

Universidad de Buenos Aires  
FACULTAD DE INGENIERIA

*Teoria de Lenguaje*

HASKELL

*1<sup>er</sup> cuatrimestre 2015*

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

15 de julio de 2015

# Índice

<b>1. Introducción</b>	<b>3</b>
1.1. Características del lenguaje . . . . .	3
1.2. Historia . . . . .	3
1.3. Compiladores e interpretes . . . . .	3
<b>2. Sintaxis básica</b>	<b>4</b>
2.1. Listas . . . . .	4
2.1.1. Rangos . . . . .	5
2.1.2. Listas por comprensión . . . . .	5
2.2. Tuplas . . . . .	5
2.3. Pattern Matching . . . . .	6
2.4. Guardas . . . . .	7
2.5. Operadores Infijos . . . . .	7
<b>3. Tipos</b>	<b>7</b>
3.1. Variables de tipo . . . . .	8
3.2. Clases de tipo . . . . .	9
<b>4. Inferencia de tipos</b>	<b>9</b>
4.1. Algoritmo de Hindley-Milner . . . . .	10
4.2. Ejemplo de aplicación del algoritmo . . . . .	11
<b>5. Recursión</b>	<b>12</b>
<b>6. Pereza</b>	<b>12</b>
<b>7. Garbage Collection</b>	<b>12</b>
<b>8. Funciones de orden superior</b>	<b>13</b>
8.1. Funciones como parametro . . . . .	13
8.2. Funciones como resultado . . . . .	14
8.3. Plieges (folds) . . . . .	14
8.4. Funciones Anónimas (lambdas) . . . . .	15
<b>9. Creación de tipos</b>	<b>15</b>
9.1. Agregando instancia de clase . . . . .	16
<b>10. Entrada y Salida</b>	<b>16</b>
10.1. Acciones IO . . . . .	16
10.1.1. Ejemplos de acciones IO . . . . .	17
10.2. Aleatoriedad . . . . .	18
10.3. Excepciones . . . . .	18
<b>11. Módulos</b>	<b>20</b>
11.1. Sintaxis . . . . .	20
11.2. Creación de módulos . . . . .	21
<b>12. Functores</b>	<b>21</b>
12.1. Leyes de los Functores . . . . .	22
12.2. Functores Aplicativos . . . . .	22

<b>13. Mónadas</b>	<b>23</b>
13.1. Las Leyes de las Mónadas . . . . .	23
13.2. Mónadas comunes . . . . .	24
13.3. Manejo de errores . . . . .	24
13.3.1. Mónada Maybe . . . . .	24
13.3.2. Either . . . . .	25
13.3.3. Mónada Error . . . . .	25
13.4. Mónada List . . . . .	26
13.5. Mónada IO . . . . .	26
13.6. Mónada State . . . . .	26
13.7. Mónada Reader . . . . .	27
13.8. Mónada Writer . . . . .	28
<b>14. Zippers</b>	<b>28</b>
<b>15. Concurrencia</b>	<b>29</b>
15.1. Threads . . . . .	29
15.2. Comunicación entre hilos . . . . .	30
15.3. Comunicación por canales . . . . .	30
15.4. Aclaraciones y problemas . . . . .	30
<b>16. Paralelismo</b>	<b>31</b>
<b>17. Software Transactional Memory</b>	<b>31</b>
<b>18. Aplicaciones</b>	<b>32</b>
18.1. Haskell en el mundo real . . . . .	32
18.2. Programas escritos en Haskell . . . . .	33

## 1. Introducción

Este informe busca reunir, explicar y ejemplificar las características principales del lenguaje **Haskell**, desde lo básico de la sintaxis hasta técnicas avanzadas y aplicaciones.

### 1.1. Características del lenguaje

Haskell es un lenguaje de programación puramente funcional. Decimos que es funcional porque enfatiza la aplicación de funciones y no maneja datos mutables o de estado. Decimos que es puro porque emplea las funciones puras, es decir, que no tienen efectos secundarios, y para una misma entrada siempre se obtiene el mismo resultado. Vamos a ver más adelante que tiene mecanismos para simular funciones impuras (sin ellas no podría comunicarse con el mundo real).

Es nombrado en homenaje al lógico Haskell Curry.

Otras características destacables:

- Tipado fuerte (datos de tipo concreto)
- Tipado estático (comprobación de tipos durante la compilación)
- Inferencia de tipos
- Muy alto nivel
- Perezoso

### 1.2. Historia

Los lenguajes funcionales no son algo nuevo: existían hace mucho tiempo. *Lisp* es considerado el primer lenguaje funcional y fue creado a fines de los 50. Pero el interés hacia ellos creció en los 80 con el lanzamiento de *Miranda*, que fue uno de los primeros lenguajes funcionales que apuntaban al uso comercial en vez del uso académico. El problema fue que *Miranda* era propietario y pago. En una conferencia de Lenguajes Funcionales en el 87 la conferencia llegó a un consenso para definir un estándar abierto para estos lenguajes.

La primera versión de Haskell se definió en 1990. Siguieron varias definiciones de especificaciones, que se fueron enumerando (1.0, 1.1, etc.) y a fines del 1997 culminaron en **Haskell 98**. A partir de ese momento continuó evolucionando. La última especificación usada hasta hoy en día es la del **Haskell 2010**.

### 1.3. Compiladores e interpretes

El compilador de código Haskell más utilizado es GHC (Glasgow Haskell Compiler). Se utilizó para las pruebas del código para este informe. Algunas de sus características:

- Es escrito en Haskell, C y C++.
- Es multiplataforma (Se puede utilizar en Windows, Mac OS y la mayoría de sistemas de Unix).
- Funciona con la mayoría de arquitecturas de procesador.
- Compila a código nativo.
- Posee un interprete (GHCi).
- Soporta concurrencia y paralelismo.
- Posee una gran cantidad de librerías, aunque algunas sólo funcionan bajo GHC.

Entre otras implementaciones se pueden destacar los compiladores **nhc98** y **HBC**, y el intérprete **Hugs**.

## 2. Sintaxis básica

Las funciones se definen de una manera similar a como son llamadas. El nombre de la función es seguido por los parámetros. Una definición se escribe agregando la definición de lo que hace la función con un `=` adelante.

```
1 duplicar x = x + x
```

Podemos utilizar la función que acabamos de definir utilizando el interprete de ghci:

```
1 ghci> duplicar 10
2 20
3 ghci> duplicar 3.3
4 6.6
```

Haskell posee de las sentencias *if .. then .. else*. La diferencia de los lenguajes imperativos es que la parte *else* es obligatoria:

```
1 duplicarNumeroChico x = if x > 100 then x else x * 2
```

### 2.1. Listas

Las listas son una estructura de datos muy usada. En Haskell las listas son homogéneas: almacena elementos del mismo tipo.

```
1 ghci> let numbers = [4,8,15,16,23,42]
2 ghci> numbers
3 [4,8,15,16,23,42]
```

Las cadenas de texto también son listas: ¡hola! es solo un azúcar sintáctico de `['H', 'o', 'l', 'a']`. Para concatenar listas en Haskell hacemos:

```
1 ghci> [1,2,3,4] ++ [9,10,11,12]
2 [1,2,3,4,9,10,11,12]
3 ghci> "Hola" ++ " " ++ "mundo"
4 "Hola mundo"
```

Hay que tener cuidado con el operador `++` ya que para concatenar tiene que recorrer el argumento izquierdo entero. Para concatenar algo al principio de la lista se usa el operador `⋈`, es instantáneo:

```
1 ghci> 'H' ⋈ "ola mundo"
2 "Hola mundo"
```

Algunas funciones básicas para operar con los elementos de las listas:

```
1 ghci> head [1,2,3]
2 1
3 ghci> tail [1,2,3]
4 [2,3]
5 ghci> last [1,2,3]
6 3
7 ghci> reverse [1,2,3]
8 [3,2,1]
9 ghci> length [1,2,3]
10 3
11 ghci> take 2 [1,2,3]
12 [1,2]
13 ghci> drop 2 [1,2,3]
14 [3]
```

### 2.1.1. Rangos

¿Qué pasa si queremos todos los números entre 1 y 15? Podríamos escribirlos a mano, pero por suerte Haskell nos simplifica el trabajo con *rangos*. Se utilizan de la siguiente manera:

```
1 ghci> [1..15]
2 [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
3 ghci> ['a'..'n']
4 "abcdefghijklmn"
5 ghci> [2,4..20]
6 [2,4,6,8,10,12,14,16,18,20]
```

Como se puede ver, funciona con cualquier elemento que tenga alguna orden. En el último ejemplo le estamos especificando al rango un paso.

Los rangos también se pueden utilizar para crear listas infinitas. Como ya mencionamos, Haskell es un lenguaje perezoso y solo va a evaluar los elementos de la lista infinita que sean necesarios.

```
1 ghci> take 5 [1,2..]
2 [1,2,3,4,5]
```

Algunas funciones útiles para la generación de listas infinitas:

```
1 ghci> take 10 (cycle [1,2,3])
2 [1,2,3,1,2,3,1,2,3,1]
3 ghci> take 4 (repeat 'a')
4 "aaaa"
5 ghci> replicate 'a' 4
6 "aaaa"
```

### 2.1.2. Listas por comprensión

Algo muy común en las matemáticas es definir conjuntos describiendo como son sus elementos, por ejemplo todos los  $x$ , *tales que*. En matemáticas podemos definir un conjunto de  $x$  naturales, pares y menores a 20 de la siguiente manera:  $S = \{x * 2 | x \in N, x < 10\}$ . En Haskell lo podríamos escribir como `take 10 [2,4..]`, pero, ¿y si queremos algo más complejo? Para ello podemos usar las *listas por comprensión*.

```
1 ghci> [x*2 | x <- [1..10]]
2 [2,4,6,8,10,12,14,16,18,20]
```

Podemos armar listas de comprensión más complejas también:

```
1 ghci> [x | x <- [50..100], x `mod` 7 == 3]
2 [52,59,66,73,80,87,94]
3 ghci> let a = ["foo", "bar"]
4 ghci> let b = ["123", "456"]
5 ghci> [x ++ " " ++ y | x <- a, y <- b]
6 ["foo 123", "foo 456", "bar 123", "bar 456"]
```

## 2.2. Tuplas

Tuplas son parecidas a las listas, pero con unas diferencias fundamentales. Las tuplas tienen un número de elementos definido, pero no necesariamente tienen que ser del mismo tipo. A continuación se pueden apreciar algunos ejemplos de uso de tuplas, con las funciones **fst** y **snd**, que devuelven el primer y el segundo elemento respectivamente.

```
1 ghci> fst (1,80)
2 1
3 ghci> fst ("Hola", [1,2,3],50)
```

```

4 "Hola"
5 ghci> snd (1,80)
6 80
7 ghci> snd ("Hola", [1,2,3],50)
8 [1,2,3]

```

### 2.3. Pattern Matching

A continuación se especificarán algunas construcciones sintácticas denominadas como ajuste de patrones (pattern matching). Una de las ventajas del ajuste de patrones es que se puede simplificar el código de manera que quede más elegante, mejorando la legibilidad.

Un ejemplo simple es crear una función que al pasarle un número verifica si es el número que buscamos:

```

1 lucky :: (Integral a) => a -> String
2 lucky 7 = "¡El siete de la suerte!"
3 lucky x = "Lo siento, ¡no es tu día de suerte!"

```

Los patrones son verificados de arriba a abajo, de manera que lucky primero verifica si se le pasó el 7 como parámetro a la función, de ser así devuelve una cadena, o de lo contrario devuelve otra.

Otro ejemplo:

```

1 sayMe :: (Integral a) => a -> String
2 sayMe 1 = "Uno!"
3 sayMe 2 = "Dos!"
4 sayMe 3 = "Tres!"
5 sayMe x = "No entre uno 1 y 3"

```

Otra vez se verifican los patrones de arriba a abajo. Esto es útil, por ejemplo, para no tener que anidar if, then, else en el cuerpo de la función

Si no se cubre algún patrón, al intentar matchear un patrón nuevo, Haskell lanza un error:

```

1 charName :: Char -> String
2 charName 'a' = "Alejandro"
3 charName 'b' = "Bernardo"
4 charName 'c' = "Carlos"

```

Utilizando el intérprete veremos:

```

1 ghci> charName 'a'
2 "Alejandro"
3 ghci> charName 'b'
4 "Bernardo"
5 ghci> charName 'c'
6 "Carlos"
7 ghci> charName 'z'
8 :(2,1)-(4,23): Non-exhaustive patterns in function charName

```

El patrón `_` indica que no es de importancia y no se utilizará el valor:

```

1 first :: (a, b, c) -> a
2 first (x, _, _) = x
3
4 second :: (a, b, c) -> b
5 second (_, y, _) = y
6
7 third :: (a, b, c) -> c
8 third (_, _, z) = z

```

El patrón `x:xs` es utilizado para listas. Diferencian la cabeza (`x`) del resto de la lista (`xs`). La sintaxis de este patrón es definida por el caracter `:`, el cual divide a la cabeza del resto de la lista.

Ejemplos:

Para identificar la cabeza de una lista:

```
1 head' :: [a] -> a
2 head' [] = error "¡Hey, no puedes utilizar head con una lista vacía!"
3 head' (x:_) = x
```

(El patrón [] representa una lista vacía)

Para determinar la longitud de una lista:

```
1 length' :: (Num b) => [a] -> b
2 length' [] = 0
3 length' (_:xs) = 1 + length' xs
```

## 2.4. Guardas

Son expresiones que consisten básicamente en booleanos. Si se evalúa a True, se utiliza el cuerpo de la función correspondiente. Son otra buena alternativa para reemplazar un árbol de if, then, else. Se indican con barras verticales, normalmente tienen sangrías y están alineadas.

Ejemplo:

```
1 max' :: (Ord a) => a -> a -> a
2 max' a b
3   | a > b = a
4   | otherwise = b
```

También pueden ser escritas en una sola línea:

```
1 max' :: (Ord a) => a -> a -> a
2 max' a b | a > b = a | otherwise = b
```

## 2.5. Operadores Infijos

Los operadores infijos también son funciones en Haskell. Por ejemplo la función de concatenación de listas se define de la siguiente forma:

```
1 (++) :: [a] -> [a] -> [a]
2 [] ++ ys = ys
3 (x:xs) ++ ys = x : (xs++ys)
```

Las funciones comunes también pueden ser llamadas de forma infija.

```
1 add x y = x + y
```

```
1 gchi> 10 'add' 5
2 15
```

## 3. Tipos

Ya hemos mencionado que Haskell tiene tipado estático y fuerte. Esto significa que por ejemplo si queremos dividir un Boolean por un Int no sólo va a dar error sino que además no va a llegar a compilar. Al contrario que en Java o en C, donde debemos especificar tipos para que se pueda compilar, en Haskell no siempre es necesario, ya que los tipos, si no se especifican, son inferidos por el compilador. Para empezar, vemos unos ejemplos simples de tipos, preguntándole al intérprete de GHC:



```

1 ghci> :t 'a'
2 'a' :: Char
3 ghci> :t True
4 True :: Bool
5 ghci> :t "HOLA!"
6 "HOLA!" :: [Char]
7 ghci> :t (True, 'a')
8 (True, 'a') :: (Bool, Char)
9 ghci> :t 4 == 5
10 4 == 5 :: Bool

```

De esta manera vemos que cuando se ejecuta el comando `:t` en una expresión devuelve esa expresión seguida de `::` y su tipo. En Haskell, las funciones también tienen tipos. Para ver esto vamos a implementar la función declarando primero su tipos. Aquí Haskell no infiere el tipo porque ya está explícito:

```

1 sumarDos :: Int -> Int
2 sumarDos a = 2 + a

```

Otro ejemplo con 3 entradas:

```

1 addThree :: Int -> Int -> Int -> Int
2 addThree x y z = x + y + z

```

**Int** representa enteros. Se utiliza para representar número enteros, por lo que 7 puede ser un **Int** pero 7.2 no puede. **Int** está acotado, lo que significa que tiene un valor máximo y un valor mínimo. Normalmente en máquinas de 32bits el valor máximo de **Int** es 2147483647 y el mínimo -2147483648. **Integer** representa enteros también. La diferencia es que no están acotados así que pueden representar números muy grandes. Sin embargo, **Int** es más eficiente.

```

1 factorial :: Integer -> Integer
2 factorial n = product [1..n]

1 ghci> factorial 50
2 30414093201713378043612608166064768844377641568960512000000000000

```

**Float** es un número real en coma flotante de simple precisión.

```

1 circumference :: Float -> Float
2 circumference r = 2 * pi * r

```

```

1 ghci> circumference 4.0
2 25.132742

```

**Double** es un número real en coma flotante de... ¡Doble precisión!.

```

1 circumference' :: Double -> Double
2 circumference' r = 2 * pi * r

```

```

1 ghci> circumference' 4.0
2 25.132741228718345

```

**Bool** es el tipo booleano. Solo puede tener dos valores: **True** o **False**.

**Char** representa un caracter. Se define rodeado por comillas simples. Una lista de caracteres es una cadena.

### 3.1. Variables de tipo

Una variable de tipo equivale a un tipo genérico. Las funciones que contienen variables de tipo son funciones polimórficas. Esto se explica en el siguiente ejemplo, preguntándole el tipo a la función `head`. Ejemplo 1:

```

1 ghci> :t head
2 head :: forall a. [a] -> a

```

**forall** *a* indica que *a* es una variable de tipo, o sea que para cualquier tipo *a*, la declaración de tipos de la función **head** va a ser `[a] -> a` (recibe una lista que contiene elementos de cualquier tipo y devuelve un elemento de tal tipo).

La siguiente declaración de tipos indica que la función **fst** recibe una tupla que contiene elementos de cualquier tipo (pueden ser iguales o no), y devuelve un elemento del tipo del primer elemento de la tupla.

```

1 ghci> fst (1, 2)
2 1
3 ghci> :t fst
4 fst :: forall a b. (a, b) -> a

```

### 3.2. Clases de tipo

Las clases de tipo tienen un comportamiento parecido a lo que son las interfaces en Java, y en general son utilizadas para generar restricciones para los tipos de las funciones que se declaran. Esto se ve de manera más fácil en el siguiente ejemplo:

Preguntamos el tipo a la función de comparación por igualdad:

```

1 ghci> :t (==)
2 (==) :: forall a. Eq a => a -> a -> Bool

```

**(Eq a)** es una restricción en la declaración de tipos. **Eq** es una clase de tipo que indica que el tipo genérico *a* es comparable por igualdad.

**Ord**: Para tipos que poseen algún orden.

```

1 ghci> :t (>)
2 (>) :: forall a. Ord a => a -> a -> Bool

```

**(>)** recibe dos parámetros y devuelve *True* si el primero es mayor al segundo. De lo contrario, devuelve *False*.

**Show**: Para tipos que pueden ser representados en forma de texto.

```

1 ghci> show 3
2 "3"
3 ghci> show False
4 "False"
5 ghci> :t show
6 show :: forall a. Show a => a -> String

```

**show** recibe elementos dentro de la clase de tipo **Show** y devuelve una cadena de caracteres.

## 4. 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. Uno puede escribir una función que suma dos números como:

```

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

```

Donde la primera línea 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 función **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.

#### 4.1. Algoritmo de Hindley-Milner

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:

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

$$\frac{\Gamma \vdash e_0 : \tau \rightarrow \tau' \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0 e_1 : \tau'} \quad [\text{App}]$$

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

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

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

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

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 estuviesen indicando una operación and lógica. Lo siguiente son los dos puntos ":", estos indican de que tipo es una expresión, por ejemplo  $x : \text{Int}$  indica que x es del tipo int. El contexto se representa con  $\gamma$ . 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 ",", es 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  $e_0$  que toma un elemento del tipo  $\tau$  y devuelve otro que es del tipo  $\tau'$ , y en ese mismo entorno tenemos una segunda expresión que es del tipo  $\tau$ , entonces

podemos decir que en ese mismo entorno que la expresión  $e_0$   $e_1$  es del tipo  $\tau'$ . La quinta nos dice que si en un entorno la expresión  $e$  es del tipo  $\sigma'$  y que  $\sigma'$  es un subtipo de  $\sigma$  entonces la expresión  $e$  es del tipo  $\sigma$ .

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 cumplan, 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.

## 4.2. Ejemplo de aplicación del algoritmo

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

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

que es lo mismo que

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

para ajustar el patrón tomamos

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

esto encaja en el segundo patrón, donde se realiza  $e_0$   $e_1$  que es del tipo  $\tau'$ , para esto se induce que  $e_0$  es del tipo  $\tau \rightarrow \tau'$  y que  $e_1$  es tipo  $\tau$ . Hasta ahora solo sabemos que  $y$  es del tipo  $\tau$ . Luego hay que ajustar el patrón  $e_0$ . Donde volvemos a caer en el segundo caso. Entonces tenemos que :

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

Con  $e'0$   $e'1$  es del tipo  $\tau \rightarrow \tau'$  Concluimos que, dado que  $e'0$  es del tipo  $\tau_1 \rightarrow \tau_1'$  y que  $e'1$  es del tipo  $\tau_1$ , como  $e'0$   $e'1 = e_0$  entonces  $e_0 : \tau_1' = \tau \rightarrow \tau'$  Hasta aquí tenemos que

```
1  y : τ
2  x : τ1
3  (+) : τ1 → τ → τ'
```

Además, como  $(+) : (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a$  Por la primera regla tenemos que  $\tau_1 \rightarrow \tau \rightarrow \tau' = a \rightarrow a \rightarrow a$  O sea

```
1  τ1 = a
2  τ = a
3  τ' = a
```

Y, finalmente, usando la quinta regla tenemos que

```
1  τ1 ⊆ a
2  τ ⊆ a
3  τ' ⊆ a
```

Finalmente solo resta recomponer todo, usando que  $\text{sumar} = e'0$   $e'1$   $e_1$  El tipo de sumar es  $(\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a$ .

## 5. 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.

```
1 factorial 0 = 1
2 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:

```
1 length [] = 0
2 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 abstraída por las funciones de las librerías 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:

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

## 6. 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 pasaje de parámetros a las funciones se realiza por necesidad.

El siguiente es un ejemplo de la pereza de Haskell:

```
1 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. También pueden no ser necesarios por completo, o ser *parcialmente* requeridos:

```
1 let (a, b) = (length [1..5], reverse "hola mundo")
2   '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 pasándole **undefined** y viendo si su ejecución falla. Esto se debe a que en Haskell la evaluación forzada del **undefined** siempre termina en un error.

```
1 let (x, y) = undefined in x -- Error!
2 length [undefined, undefined, undefined]
3 -- 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 difícil de predecir, por ejemplo las expresiones `3+2 :: Int` y `5 :: Int` denotan el mismo valor pero pueden tener diferentes tamaños en memoria.

## 7. 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 haciéndolo inalcanzable, de este modo las referencias almacenadas son solo 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.

## 8. 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.

### 8.1. Funciones como parametro

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)]
7   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.

## 8.2. 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:

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

Para cada valor de  $x$ , `multiplicarPor` da como resultado una función que recibe un número y devuelve el producto del número 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 Curricadas. 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 Curricación 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`.

```
1 sumar3 x y z = x + y + z

1 ghci> :t sumar3
2 sumar3 :: forall a. Num a => a -> a -> a -> a
3 ghci> :t sumar3 1
4 sumar3 1 :: forall a. Num a => a -> a -> a
5 ghci> :t sumar3 1 2
6 sumar3 1 2 :: forall a. Num a => a -> a
7 ghci> :t sumar3 1 2 3
8 sumar3 1 2 3 :: forall a. Num a => a
9 ghci> :t 6
10 6 :: forall a. Num a => a
```

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.

## 8.3. Plieges (folds)

Unas funciones particularmente útiles y cómodas son los plieques (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 querriamos implementar la función `elem` utilizando plieques podríamos hacerlo como

```
1 elem' :: (Eq a) => a -> [a] -> Bool
2 elem' y ys = foldl f False ys
3   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, pasárselo 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:

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

## 8.4. 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:

```
1 (\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

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

## 9. Creación de tipos

Un tipo se define usando la palabra clave **data**, y luego el nombre del tipo y los constructores. Esto lo vemos en el siguiente ejemplo, donde se muestra cómo está definido el tipo **Bool** en la librería estándar:

```
1 data Bool = False | True
```

**data** significa que se define un nuevo tipo de dato, la parte izquierda del `=` indica el nombre del tipo y en la parte derecha están los constructores. La barra `|` se puede leer como un operador *or*, por lo que se puede interpretar que `Bool` puede tomar los valores *False* o *True*.

En el siguiente ejemplo definiremos una figura que puede ser sólo un círculo o un rectángulo:

```
1 data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

Acá se está indicando que el tipo `Shape` puede tomar el valor de alguno de los constructores (`Circle` o `Rectangle`), los cuales son funciones que toman como parámetros los tipos indicados a la derecha del nombre del constructores (en el caso de `Circle` tres valores flotantes y en el caso de `Rectangle` serán cuatro). Esto significa que si le preguntamos el tipo a los constructores nos encontraremos con lo siguiente:

```
1 ghci> :t Circle
2 Circle :: Float -> Float -> Float -> Shape
3 ghci> :t Rectangle
4 Rectangle :: Float -> Float -> Float -> Float -> Shape
```

Entonces, si queremos crear una función que nos diga la superficie o área de una figura, se hará de la siguiente manera dependiendo de si es un círculo o rectángulo:

```
1 surface :: Shape -> Float
2 surface (Circle _ _ r) = pi * r ^ 2
3 surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```



Acá destacamos que `surface` no puede tomar un rectángulo o un círculo porque estos no son tipos, a diferencia de `Shape`.

Entonces vemos en el intérprete cómo funciona esta función:

```
1 ghci> surface $ Circle 10 20 10
2 314.15927
3 ghci> surface $ Rectangle 0 0 100 100
4 10000.0
```

### 9.1. Agregando instancia de clase

Si intentamos mostrar por pantalla **Circle 10 20 5** en una sesión de GHCi obtendremos un error. Esto sucede porque Haskell aún no sabe cómo representar nuestro tipo con una cadena. Cuando intentamos mostrar un valor por pantalla, primero Haskell ejecuta la función `show` para obtener la representación en texto de un dato y luego lo muestra en la terminal. Para hacer que el tipo `Shape` forme parte de la clase de tipo **Show**, añadimos **deriving (Show)** al final de la declaración de tipo, y automáticamente Haskell hace que ese tipo forme parte de la clase de tipo **Show**:

```
1 data Shape = Circle Float Float Float | Rectangle Float Float Float Float deriving (Show)
```

Entonces, al imprimir por pantalla mostrará lo siguiente:

```
1 ghci> Circle 10 20 5
2 Circle 10.0 20.0 5.0
3 ghci> Rectangle 50 230 60 90
4 Rectangle 50.0 230.0 60.0 90.0
```

Los constructores de datos son funciones, por lo que pueden ser mapeados, se los puede aplicar parcialmente, etc. Por ejemplo, si queremos una lista de círculos concéntricos con diferente radio podemos escribir:

```
1 ghci> map (Circle 10 20) [4,5,6]
2 [Circle 10.0 20.0 4.0, Circle 10.0 20.0 5.0, Circle 10.0 20.0 6.0]
```

Otro ejemplo, es crear el tipo de dato `figura` con otro tipo de dato auxiliar que represente un punto en el espacio:

```
1 data Point = Point Float Float deriving (Show)
2 data Shape = Circle Point Float | Rectangle Point Point deriving (Show)
3
4 surface :: Shape -> Float
5 surface (Circle _ r) = pi * r ^ 2
6 surface (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

Y si queremos exportarlo a un módulo, éste se declara en el encabezado de la siguiente manera:

```
1 module Shapes
2 ( Point(..)
3 , Shape(..)
4 , surface
5 ) where
6
7 -- ... declaración de los constructores y las funciones.
```

## 10. Entrada y Salida

### 10.1. Acciones IO

Para mostrar por pantalla se necesita llamar a una función que devuelva una acción IO.

```
1 putStrLn "Hola mundo"
```

Si le preguntamos el tipo a la función `putStrLn`, veremos que recibe una cadena y devuelve una acción IO que no devuelve valores.

```
1 ghci> :t putStrLn
2 putStrLn :: IO ()
```

Ahora bien, si lo que queremos es pedir una cadena de caracteres utilizando la función `getLine`, sabremos que `getLine` utiliza otra acción IO que esta vez devuelve un valor de tipo `String`, el cual es resultado de lo que se ingresó por teclado en la ejecución del programa:

```
1 :t getLine
2 getLine :: IO String
```

Lo que falta ahora es saber cómo introducir acciones IO en el código de un programa Haskell. Las acciones IO son código impuro, ya que produce efectos laterales en contraposición a la programación pura. En Haskell debe separarse lo que es código puro de código impuro. Por esto al tener varias acciones IO (dos o más), deben hacerse dentro de un mismo bloque `do`:

```
1 main = do
2   putStrLn "Ingrese su nombre"
3   nombre <- getLine
4   putStrLn ("Hola " ++ nombre)
```

Por lo tanto, podemos ver que la siguiente acción es inválida:

```
1 ghci> "Hola, mi nombre es" ++ getLine
2 Couldn't match expected type 'String' with actual type 'IO String'
3 In the second argument of '(++)', namely 'getLine'
4 In the expression: "Hola, mi nombre es" ++ getLine
```

### 10.1.1. Ejemplos de acciones IO

`putStr` toma una cadena y la imprime en pantalla sin el salto de línea.

```
1 mostrarString = do
2   putStr "Hola, "
3   putStr "programo "
4   putStr "en "
5   putStrLn "Haskell!"
```

```
1 ghci> mostrarString
2 Hola, programo en Haskell!
```

`print` muestra en pantalla miembros de la clase `Show`.

```
1 ghci> print True
2 True
3 ghci> print 2
4 2
5 ghci> print "Hola"
6 "Hola"
7 ghci> print 3.2
8 3.2
9 ghci> print [3,4,3]
10 [3,4,3]
```

`when` se encuentra en el módulo `Control.Monad`. Toma un valor booleano y una acción IO de forma que si el valor booleano es `True`, devolverá la misma acción que se le suministre. Sin embargo, si es falso, devolverá una acción `return ()`, acción que no hace absolutamente nada.

```

1 import Control.Monad
2
3 main = do
4   c <- getChar
5   when (c /= ' ') $ do
6     putChar c
7     main

```

También existe la función `unless` que es exactamente igual a `when` solo que devuelve la acción original cuando encuentra *False* en lugar de *True*.

`sequence` toma una lista de acciones IO y devuelve una acción que realizará todas esas acciones una detrás de otra. El resultado contenido en la acción IO será una lista con todos los resultados de todas las acciones IO que fueron ejecutadas. Su tipo es `sequence :: [IO a] -> IO [a]`. Hacer esto:

```

1 main = do
2   a <- getLine
3   b <- getLine
4   c <- getLine
5   print [a,b,c]

```

Produce el mismo resultado que:

```

1 main = do
2   rs <- sequence [getLine, getLine, getLine]
3   print rs

```

## 10.2. Aleatoriedad

La función `random` toma un generador, el cual es creado por la función `mkStdGen` y devuelve una dupla que contiene un número real y otro valor generador. Si invoco dos veces a la función `random` con el mismo valor generador, devolverá los mismos valores. Ejemplo:

```

1 ghci> random (mkStdGen 100)
2 (-3650871090684229393,693699796 2103410263)
3 ghci>random (mkStdGen 100)
4 (-3650871090684229393,693699796 2103410263)
5 ghci>random (mkStdGen 2432434)
6 (708227736329069275,242756692 2103410263)

```

También podemos modificar el tipo de valor que queremos que nos devuelva:

```

1 ghci>random (mkStdGen 949488) :: (Float, StdGen)
2 (0.8241101,1597344447 1655838864)
3 ghci>random (mkStdGen 949488) :: (Bool, StdGen)
4 (False,1485632275 40692)

```

## 10.3. Excepciones

Tanto el código puro como el código impuro puede lanzar excepciones, pero estas excepciones siempre son capturadas en la parte I/O del código. Esto sucede porque no se sabe si algo será evaluado en el código puro ya que se evalúa de forma perezosa y no tiene definido un orden de ejecución concreto, mientras que las partes de E/S sí lo tienen.

En esta sección solo se verá como utilizar las excepciones de E/S. Las excepciones de E/S ocurren cuando algo va mal a la hora de comunicarnos con el mundo exterior. Por ejemplo, podemos tratar de abrir un fichero y luego puede ocurrir que ese fichero ha sido eliminado o algo parecido.

El siguiente programa abre un fichero que ha sido obtenido como parámetro y e indica cuántas líneas contiene:

```

1 import System.Environment
2 import System.IO
3
4 main = do (fileName:_) <- getArgs
5   contents <- readFile fileName
6   putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

```

Vemos qué sucede si se le da el nombre de un fichero que no existe:

```

1 $ runhaskell linecount.hs i_dont_exist.txt
2 linecount.hs: i_dont_exist.txt: openFile: does not exist (No such file or directory)

```

Se obtuvo un error que indica que el fichero no existe. A continuación se ve cómo capturar el error y modificar el mensaje a gusto del programador, utilizando la función **doesFileExist** de **System.Directory**:

```

1 import System.Environment
2 import System.IO
3 import System.Directory
4
5 main = do (fileName:_) <- getArgs
6   fileExists <- doesFileExist fileName
7   if fileExists
8     then do contents <- readFile fileName
9       putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"
10    else do putStrLn "The file doesn't exist!"

```

Se hizo `fileExists <- doesFileExist fileName` porque **doesFileExist** tiene como declaración de tipo `doesFileExist :: FilePath -> IO Bool`, lo que significa que devuelve una acción IO que tiene como resultado un valor booleano que indica si el fichero existe o no. No se puede utilizar **doesFileExist** directamente en una expresión *if*. Otra solución sería utilizando excepciones. Un fichero que no existe es una excepción que se lanza desde I/O, así que capturarla en I/O es totalmente aceptable.

Para esto se utiliza la sentencia **catch**.

```

1 import System.Environment
2 import System.IO
3 import System.IO.Error
4
5 main = catch toTry handler
6
7 toTry :: IO ()
8 toTry = do (fileName:_) <- getArgs
9   contents <- readFile fileName
10  putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"
11
12 handler :: IOError -> IO ()
13 handler e
14   | isDoesNotExistError e = putStrLn "The file doesn't exist!"
15   | otherwise = ioError e

```

**toTry** es una acción de E/S que intentaremos ejecutar y **handler** es la función que toma un **IOError** y devuelve una acción que será ejecutada en caso de que suceda una excepción.

Existen varios predicados que trabajan con **IOError** que se pueden utilizar junto las guardas, ya que, si una guarda no se evalúa a *True*, se seguirá evaluando la siguiente guarda. Los predicados que trabajan con **IOError** son:

- `isAlreadyExistsError`
- `isDoesNotExistError`

- `isAlreadyInUseError`
- `isFullError`
- `isEOFError`
- `isIllegalOperation`
- `isPermissionError`
- `isUserError`

Por ejemplo, podríamos tener un manipulador como el siguiente:

```

1 handler :: IOError -> IO ()
2 handler e
3   | isDoesNotExistError e = putStrLn "The file doesn't exist!"
4   | isFullError e = freeSomeSpace
5   | isIllegalOperation e = notifyCops
6   | otherwise = ioError e

```

## 11. Módulos

Los módulos son colecciones de tipos, clases de tipos y funciones relacionadas entre sí. En un programa hecho en Haskell, el módulo principal carga otros módulos y utiliza las funciones de los mismos para realizar lo que el programa fuere a hacer. Es una buena práctica programar dividiendo el código en módulos lo mas independiente posibles para tener un mejor mantenimiento del programa y aprovechar código que puede ser reutilizado.

### 11.1. Sintaxis

La sintaxis para importar módulos en Haskell es **import** <modulename>, y las importaciones de los módulos suelen estar al principio de los ficheros ya que deben realizarse antes de definir cualquier función.

Una vez que se importan los módulos, pasan a estar definidos en el espacio global. Hay varias maneras de importar un módulo. A continuación veremos ejemplos importando el módulo **Data.List**, el cual contiene funciones para el manejo de listas:

Si se quiere importar todo el módulo:

```

1 import Data.List

```

Para importar algunas funciones del módulo (en este caso las funciones `intercalate` y `sort` del módulo **Data.List**):

```

1 import Data.List (intercalate, sort)

```

Para importar todo el módulo excepto una o más funciones (en este caso todo el módulo **Data.List** excepto sus funciones `intercalate` y `sort`):

```

1 import Data.List hiding (intercalate, sort)

```

Importaciones cualificadas (para evitar colisiones de nombres):

```

1 import qualified Data.Map

```

En este caso, si queremos referirnos, por ejemplo, a la función `filter` de **Data.Map**, debemos utilizar **Data.Map.filter**, mientras que si usamos simplemente `filter` nos referimos al filtro normal que utiliza Haskell.

Una manera más fácil de hacer llamadas a módulos importados de manera cualificada es renombrar al módulo de manera que la llamada sea más abreviada:

```
1 import qualified Data.Map as M
```

De esta manera, si yo quiero llamar a la función `filter` de **Data.Map**, debo utilizar simplemente **M.filter**. Algunos módulos que vienen con Haskell:

- Para listas: **Data.List**
- Para caracteres: **Data.Char**
- Para listas de asociación (listas de duplas (clave; valor)): **Data.Map**
- Para conjuntos: **Data.Set**

## 11.2. Creación de módulos

Para crear un módulo, lo definimos en el script y dentro del módulo definimos sus funciones (utilizando la construcción **where**):

```
1 module Areas
2 (   areaCirculo
3   , areaCuadrado
4   , areaRectangulo
5 ) where
6
7 areaCirculo :: Float -> Float
8 areaCirculo radio = pi * (radio ^ 2)
9
10 areaCuadrado :: Float -> Float
11 areaCuadrado lado = lado ^ 2
12
13 areaRectangulo :: Float -> Float -> Float
14 areaRectangulo a b = a * b
```

Para importarlo en otro script, se hace como se mencionó anteriormente (Areas está en el mismo directorio donde estamos trabajando):

```
1 import Areas
2
3 areaCirculo 1.0
4 areaCuadrado 2.0
5 areaRectangulo 3.0 1.0
```

## 12. Functores

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

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

La idea de `fmap` es que toma una función 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 devuelve una caja con el resultado de aplicar la función.

Un término más correcto para definir lo que es un functor 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 funtores básicos son las listas `[]`, `Maybe`, `Either a` (en este último caso, como necesitamos que nuestro tipo tome un único tipo concreto y `Either` toma dos debe estar parcialmente evaluada), si se definiere una clase `Tree a` también podría ser un functor. 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:

```
1 instance Functor IO where
2   fmap f action = do
3     result <- action
4     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`.

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

Aplicar `fmap` a una función nos da una nueva función, si pensamos que aplicamos esta nueva función 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.

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

## 12.1. Leyes de los Functores

Ahora vamos a ver las leyes de los funtores. Para que algo sea una functor, debe satisfacer una serie de leyes. Se espera que todos los funtores 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 functor, nada más. Este comportamiento se describe en las leyes de los funtores. Hay dos de ellas que todas las instancias de `Functor` 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 functor debe devolver lo mismo que si mapeamos una de estas funciones sobre el functor inicial y luego mapeamos la otra función (`((f . g) = fmap f)`).

## 12.2. Funtores 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

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

obtenemos

```
1 Just (* 7)
```

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

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

La definición de esta clase de tipos nos dice que si alguien es un `Applicative` 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 función es una ampliación 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 siguiente forma:

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

## 13. 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 términos del lenguaje una mónada es un tipo parametrizado que es instancia de la clase `Monad`. Su definición es la siguiente:

```
1 class Monad m where
2   return :: a -> m a
3   (>>=) :: m a -> (a -> m b) -> m b
4   (>>) :: 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 `(>>=)`, también 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:

```
1 putStrLn "Como te llamas?"
2 >>= (\_ -> getLine)
3 >>= (\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*:

```
1 do
2   putStrLn "Como te llamas?"
3   name <- getLine
4   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 más 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.

### 13.1. Las Leyes de las Mónadas

Las mónadas por convención deben cumplir las siguientes leyes:

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



```

3
4  -- Identidad por la derecha
5  m >>= return = m
6
7  -- Asociatividad
8  (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:

```

1  (>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
2  (m >=> n) x = do
3      y <- m x
4      n y

```

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

```

1  -- Identidad por la izquierda
2  return >=> f = f
3
4  -- Identidad por la derecha
5  f >=> return = f
6
7  -- Asociatividad
8  (f >=> g) >=> h = f >=> (g >=> h)

```

## 13.2. 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.

## 13.3. Manejo de errores

### 13.3.1. Mónada Maybe

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

```

1  data Maybe a = Nothing | Just a
2
3  instance Monad Maybe where
4      return      = Just
5      fail        = Nothing
6      Nothing >>= f = Nothing
7      (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:

```

1 maybeHalf :: Int -> Maybe Int
2 maybeHalf a
3     | even a = Just (div a 2)
4     | 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:

```

1 ghc> Just 10 >>= maybeHalf
2 Just 5
3 ghc> Just 10 >>= maybeHalf >>= maybeHalf
4 Nothing
5 ghc> Just 10 >>= maybeHalf >>= maybeHalf >>= maybeHalf
6 Nothing

```

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

### 13.3.2. Either

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

```

1 data Either a b = Left a | Right b

```

Su propósito es muy parecido al de *Maybe*, *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**:

```

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

```

Y la podemos utilizar de la siguiente forma:

```

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

```

### 13.3.3. 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**:

```

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

```

Un ejemplo de uso es:

```

1 example :: (Error e, MonadError e m) => m String
2 example = throwError (strMsg "Esto es un error")
3         `catchError` const (return "Atrapo el error")
4
5 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.

### 13.4. 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:

```
1 instance Monad [] where
2   m >>= f = concat (map f m)
3   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 definición 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:

```
1 foo = do
2   x <- [1 .. 10]
3   y <- [2, 3, 5, 7]
4   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:

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

### 13.5. Mónada IO

Dado que Haskell es un lenguaje funcional y perezoso, no podemos expresar los efectos reales de las operaciones de entrada y salida con funciones puras. De hecho, estas operaciones no se pueden ejecutar perezosamente, ya que esto haría 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.

### 13.6. 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:

```
1 newtype State s a = State { runState :: (s -> (a,s)) }
2
3 instance Monad (State s) where
4   return x = State $ \s -> (x,s)
5   (State h) >>= f = State $ \s -> let (a, newState) = h s
6                                   in (State g) = f a
7                                   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 tendríamos 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:

```
1 type Stack = [Int]
2
3 pop :: State Stack Int
4 pop = State $ \(x:xs) -> (x,xs)
5
6 push :: Int -> State Stack ()
7 push a = State $ \xs -> ((),a:xs)
```

Un ejemplo de uso de la pila sería:

```
1 stackManip = do
2   push 3
3   a <- pop
4   pop
```

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

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

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

### 13.7. 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:

```
1 newtype Reader r a = Reader { runReader :: r -> a }
2
3 instance Monad (Reader e) where
4   return a      = Reader $ \e -> a
5   (Reader r) >>= f = Reader $ \e -> runReader (f (r e)) e
```

En el siguiente ejemplo se puede ver fácilmente su utilidad:

```
1 import Control.Monad.Reader
2
3 greeter :: Reader String String
4 greeter = do
5   name <- ask
6   return ("Hola, " ++ name ++ "!")
```

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

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

```
1 class MonadReader m | m -> e where
2   ask    :: m e
3   local :: (e -> e) -> m a -> m a
4
5 instance MonadReader (Reader e) where
6   ask      = Reader id
7   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.

### 13.8. 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:

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

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

```
1 import Control.Monad.Writer
2
3 doubleWithLog :: Int -> Writer String Int
4 doubleWithLog x = do
5     tell ("Acabo de duplicar " ++ (show x) ++ "!")
6     return (x * 2)
```

Si ejecutamos esta función, obtendremos lo siguiente:

```
1 ghci> runWriter $ doubleWithLog 8
2 (16, "Acabo de duplicar 8!")
```

## 14. 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:

```
1 data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)
2 data Crumb a = LeftCrumb a (Tree a) | RightCrumb a (Tree a) deriving (Show)
3 type Breadcrumbs a = [Crumb a]
4
5 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 último 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.

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

```

3 goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
4
5 goRight :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
6 goRight (Node x l r, bs) = (r, RightCrumb x l:bs)
7
8 goLeft :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
9 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

```

1 modify :: (a -> a) -> Zipper a -> Zipper a
2 modify f (Node x l r, bs) = (Node (f x) l r, bs)
3 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:

```

1 type ListZipper a = ([a],[a])
2
3 goForward :: ListZipper a -> ListZipper a
4 goForward (x:xs, bs) = (xs, x:bs)
5
6 goBack :: ListZipper a -> ListZipper a
7 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.

## 15. Concurrency

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.

### 15.1. 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:

```

1 import Control.Concurrent (forkIO)
2 import qualified Data.ByteString.Lazy as L
3 import Codec.Compression.GZip (compress)
4

```

```

5 compressFile = L.writeFile "files/foo.gz" . compress
6
7 do
8   content <- L.readFile "files/foo.txt"
9   forkIO (compressFile content)
10  putStrLn "Gracias por comprimir!"
11  return ()

```

## 15.2. 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 vacía 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 vacíe. De la misma manera, si intentamos sacar el valor de una **MVar** vacía, el hilo se suspenderá hasta que algún otro thread le agregue algo. En el siguiente ejemplo se puede ver el uso de una **MVar**:

```

1 import Control.Concurrent
2
3 do
4   m <- newEmptyMVar
5   forkIO $ do
6     v <- takeMVar m
7     putStrLn ("Recibido " ++ show v)
8   putStrLn "Enviando"
9   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`.

## 15.3. Comunicación por canales

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

```

1 import Control.Concurrent
2 import Control.Concurrent.Chan
3
4 do
5   ch <- newChan
6   forkIO $ writeChan ch "Hola!"
7   forkIO $ writeChan ch "Hola desde otro hilo!"
8   readChan ch >>= print
9   readChan ch >>= print

```

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

## 15.4. 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.

## 16. 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:

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

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

```
1 import Control.Parallel (par, pseq)
2
3 parSort :: (Ord a) => [a] -> [a]
4 parSort (x:xs) = force greater `par` (force lesser `pseq`
5                                     (lesser ++ x:greater))
6   where lesser = parSort [y | y <- xs, y < x]
7         greater = parSort [y | y <- xs, y >= x]
8 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:

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

La función **par** no promete que una expresión será evaluada en paralelo con otra, pero si promete hacerlo si tiene 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 expresión es muy liviana, como para que la ejecución paralela tenga sentido. O puede notar que todos los núcleos están ocupados en el momento.

## 17. 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 15).



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:

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

```

1 import Control.Concurrent.STM
2
3 type Account = TVar Int
4
5 withdraw :: Account -> Int -> STM ()
6 withdraw acc amount = do
7     bal <- readTVar acc
8     writeTVar acc (bal - amount)
9
10 deposit :: Account -> Int -> STM ()
11 deposit acc amount = withdraw acc (- amount)
12
13 atomically $ do
14     acc <- newTVar (20 :: Int)
15     deposit acc 20
16     withdraw acc 10
17     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 podríamos escribir de la siguiente manera:

```

1 limitedWithdraw :: Account -> Int -> STM ()
2 limitedWithdraw acc amount = do
3     bal <- readTVar acc
4     if amount > 0 && amount > bal
5     then retry
6     else writeTVar acc (bal - amount)

```

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

```

1 limitedWithdraw2 :: Account -> Account -> Int -> STM ()
2 limitedWithdraw2 acc1 acc2 amt
3     = orElse (limitedWithdraw acc1 amt) (limitedWithdraw acc2 amt)

```

## 18. Aplicaciones

### 18.1. Haskell en el mundo real

A lo largo del informe describimos las principales características de este lenguaje, las fuimos detallando y dando ejemplos. Ahora nos podemos basar en lo descrito y decir para qué y por qué sirve este lenguaje.

Siendo funcional puro y de tipado estático, el lenguaje es atractivo para los programadores: permite escribir bases de código expresivas y seguras, fáciles de mantener. El compilador GHC es una gran ventaja también. El código compilado resultante termina siendo comparable en lo performante con lenguajes de tipado estático imperativos, como C o C++.

Es ideal para sistemas concurrentes: su origen funcional facilita la escritura de código concurrente y con ayuda de herramientas, como STM y con un paralelismo sencillo de programar se vuelve seguro y performante, sin la complejidad habitual para ese tipo de aplicaciones y fácil de escalar.

Dado que se basa mucho en la teoría y tiene una muy fuerte base matemática, es muy usado en las aplicaciones científicas. Se utiliza para inteligencia artificial y machine learning, protocolos criptográficos, lingüística computacional, modelado bayesiano. Tiene aplicaciones en electrónica, robótica y hasta bioinformática [15].

## 18.2. Programas escritos en Haskell

Adelante listaremos algunos de los casos de aplicaciones y librerías grandes y productivas escritas en Haskell.

- **GHC** mismo es uno de los mejores ejemplos de aplicación escrita en Haskell con aproximadamente 150000 líneas de código Haskell en el compilador y otras tantas en librerías del núcleo. Es libre y de código abierto.  
<https://www.haskell.org/ghc/>
- **Cabal**, sistema de estructurado y empaquetado de librerías y programas escritos en Haskell. Es usado en los sistemas unix para la instalación de paquetes de Haskell.  
<https://www.haskell.org/cabal/>
- **Hackage**, registro central de paquetes de código abierto escritos en Haskell.  
<https://www.haskell.org/cabal/>  
Es interesante por varios motivos: aquí podemos encontrar muchísimos paquetes en Haskell para todo tipo de necesidades, que se pueden instalar con la herramienta *cabal-install* (una de las partes del proyecto mencionado en el ítem anterior). Y otro motivo es que todo el backend del servidor de Hackage está escrito en Haskell. Su código se puede apreciar acá:  
<https://github.com/haskell/hackage-server>
- **Chordify**, servicio web que transforma música, en forma de archivo o links a servicios más conocidos, como youtube o soundcloud, en acordes.  
<http://chordify.net/>
- **XMonad**, es un manejador de ventanas para **X** que permite organizar ventanas dinámicamente en *tiles*.  
<http://xmonad.org/>
- **Snap** y **Yesod** son frameworks para desarrollo web bastante sólidos.  
<http://snapframework.com/>  
<http://www.yesodweb.com/>
- **FPCComplete** son una serie de herramientas online para desarrollo y análisis de algoritmos.  
<urlhttps://www.fpcomplete.com/>
- **Darcs**, una herramienta de versionado de código.  
<urlhttp://darcs.net/>

## 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](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](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](https://wiki.haskell.org/Monad_laws)
- [12] All about monads, [https://wiki.haskell.org/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/>
- [15] Scientific applications of Haskell, <http://www.quora.com/What-are-some-scientific-applications-of-Haskell>
- [16] Haskell in industry, [https://wiki.haskell.org/Haskell\\_in\\_industry](https://wiki.haskell.org/Haskell_in_industry)