

CURSO DE PROGRAMACION SCALA

Sesión 10

Sergio Couto Catoira

Índice

- › Procesamiento y optimización en estructuras
- › Funciones estrictas y no estrictas
- › Laziness

Procesamiento en listas normales

- > Problema: Retirar cartas de un palo y aquellas impares
 - Lo normal es hacer una única pasada sobre las cartas eliminando las que queramos.
- > En Scala, cada map, filter, fold etc.. hace su propia pasada sobre toda la colección.

Procesamiento en listas normales

> Ejemplo:

- Iter0: `List(1,2,3,4).map(_+10).filter(_%2==0).map(_*3)`
- Iter1: `List(11, 12, 13, 14).filter(_%2==0).map(_*3)`
- Iter2: `List(12, 14).map(_*3)`
- Iter3: `List(36, 42)`

> ¿Qué genera cada pasada? ¿Cuántas listas hay?

Opciones para mejorar el proceso

- > Escribir el código como un bucle normal a mano
 - No nos vale, perdemos el *estilo*, la composición de funciones de orden superior
- > Ordenar y agrupar nuestro proceso
 - Sí nos vale, mantenemos el *estilo*
 - Primero las funciones que reduzcan la lista (filter...)
 - Agrupar funciones que transforman la lista (map...)
 - Tiene un límite. No podemos optimizar al máximo según qué casos
- > Construir mejores estructuras => no-estrictas

Funciones estrictas: ¿Qué son?

- > Estricta: Evalúa los parámetros antes de nada
- > No estricta: Puede escoger no evaluar algún parámetro

Funciones no estrictas

> Ejemplos:

- Comparadores lógicos `&&` y `||`
 `false && {println("Hello"); true}`
 `true || {println("Hello"); false}`
- Sentencia if: `if (x>0) println("Hello") else sys.error("fail")`

¿Cómo conseguirla?

- Parámetros por referencia (no se evalúan hasta que se usan)

Ejercicio

- > Define una función `if` que sea no-estricta en ambos operadores
 - `def myIf[A](cond: Boolean, onTrue, onFalse): A`

Variables lazy

- > De este modo se evalúa el parámetro tantas veces como se use

```
def maybeDuplicate(cond: Boolean, elem: => Int): Int = {  
  if (cond) elem + elem else 0  
}
```

- > Con variables definidas como lazy, retrasa la evaluación de la parte derecha y la cachea para siguientes llamadas

```
def maybeDuplicate2(cond: Boolean, elem: => Int): Int = {  
  lazy val j = elem  
  if (cond) j + j else 0  
}
```

Variables lazy

- > Ejemplos de llamada con un print para que sea más obvio

```
scala> maybeDuplicate(true, {println("hi"); 1+41})
hi
hi
res0: Int = 84

scala> maybeDuplicate2(true, {println("hi"); 1+41})
hi
res1: Int = 84
```

Syntactic sugar detrás de los valores por referencia

- > Ambos trozos de código son lo mismo.
- > El primero tiene syntactic sugar
- > Difiere la forma de llamarle
 - `myIf2(isEarly, () => "Morning", () => "Afternoon")`
 - `myIf(isEarly, "Morning", "Afternoon")`

```
def myIf[A](cond: Boolean, onTrue: => A, onFalse: => A): A = {  
  if (cond) onTrue else onFalse  
}  
  
def myIf2[A](cond: Boolean, onTrue: () => A, onFalse: () => A): A = {  
  if (cond) onTrue() else onFalse()  
}
```

Syntactic sugar detrás de los valores por referencia

- > name: () => 5 representa una función llamada name que acepta 0 argumentos y devuelve un 5.
- > Se llama con el nombre de la función y la lista de argumentos (vacía lógicamente)
 - name()
- > En Scala la forma no-evaluada se llama *thunk*. Al llamar al thunk, se fuerza su evaluación

Listas lazy - Streams

- > Comprueba la clase `Stream.scala`, verás que es muy similar a `Lista`
 - Los argumentos de `Cons` son por referencia (como `thunks` debido a que no se pueden pasar valores por referencia en la construcción de una `case class`)
 - Hay un constructor de listas vacías tipado
 - Hay constructores inteligentes que evitan:
 - Tener que construir explícitamente el *thunk*
 - `Cons(() => head, () => tail)` vs `cons(head, tail)`
 - Que se evalúen los argumentos de `Cons` cada vez que se llamen

Ejercicio

- > Define una función `headOption` que devuelva la cabecera de la lista si existe y `None` en caso contrario
- > Define una función `toList` que convierta un `Stream` a `List`
- > Define las funciones `drop` y `dropWhile` que funcionen como en listas
- > Define las funciones `take` y `takeWhile` que

Separación definición - ejecución

- > Uno de los principios de la programación funcional
- > Por un lado la definición de las tareas a realizar
- > Por otro lado se ejecuta la tarea

- > Ejemplo: Option o Either, se usan para capturar un error, pero la decisión de qué hacer con ese error se deja para otro momento o otro punto del programa.

Separación definición - ejecución

- > Usando laziness podemos separar la definición de una tarea de su evaluación (ejecución)
- > Ejercicio: Define la función `exists` que reciba una función de entrada y devuelva `true` si se cumple para alguno de los elementos.