Sergio Couto
s.couto@indizen.com

Introducción a Scala y Programación Funcional

Mayo 2018

indizen
Technology & Business Confluence

ÍNDICE

01 | Funciones estrictas

indizen
Technology & Business Confluence

- Función estricta => Evalúa todos sus parámetros al recibirlos
- Función no estricta => Escoge si evaluar sus parámetros

- ¿Cómo conseguirla?
  Parámetros por referencia (no se evalúan hasta que se usan)

Ejemplos:

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

01 | **Funciones estrictas** - ejercicio

indizen
Technology & Business Confluence

- Define una función if que sea no-estricta en ambos resultados. Debe recibir tres parámetros, la condición y los resultados para cuando se cumpla la condición y para cuando no
  - def myIf[A](cond: Boolean, onTrue, onFalse): A

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

```
def duplicateNonStrict(x: => Double): Double = {
  println(x*2)
  x*2
}
```

02 | Laziness

indizen
Technology & Business Confluence

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

```
def duplicateNonStrictLazy(x: => Double): Double = {
  lazy val j = x
  println(j+j)
  j+j
}
```

```
def duplicateNonStrict(x: => Double): Double = {
  println(x*2)
  x*2
}
```

- Ejemplos de ambas llamadas incluyendo un print en la variable

```
scala>   duplicateNonStrict({println("duplicateNonStrict"); 10+11})
duplicateNonStrict
42.0
duplicateNonStrict
res3: Double = 42.0
```

```
scala>   duplicateNonStrictLazy({println("duplicateNonStrictLazy"); 10+11})
duplicateNonStrictLazy
42.0
res5: Double = 42.0
```

02 | Laziness

indizen
Technology & Business Confluence

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

02 | Laziness

indizen
Technology & Business Confluence

- 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

03 | Streams en la API general

indizen
Technology & Business Confluence

- Es similar a una Lista pero lazy
- Sólo conocemos el primer elemento
- El equivalente a cons (::) es #::
- El equivalente a Nil es Stream.empty
- Tiene la misma API practicamente que lista
  - Transformaciones: filter, map, flatMap, take, drop, dropWhile, exists, forAll...
  - Cálculos: max, min, sum, size
- Los cálculos pueden causar problemas de memoria si el Stream es muy grande
- Las transformaciones pueden causar errores si el Stream es infinito. Ojo

- Constructor
  - val s = Stream(1,2,3)
    - s: scala.collection.immutable.Stream[Int] = Stream(1, ?)
- Obtener cabecera:
  - s.head
- Obtener cola
  - s.tail
- Accediendo a elementos:
  - S(2), s(3)

indizen
Technology & Business Confluence

Listing 5.3 Program trace for `Stream`

```
Stream(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(11, Stream(2,3,4).map(_ + 10)).filter(_ % 2 == 0).toList
Stream(2,3,4).map(_ + 10).filter(_ % 2 == 0).toList
cons(12, Stream(3,4).map(_ + 10)).filter(_ % 2 == 0).toList
12 :: Stream(3,4).map(_ + 10).filter(_ % 2 == 0).toList
12 :: cons(13, Stream(4).map(_ + 10)).filter(_ % 2 == 0).toList
12 :: Stream(4).map(_ + 10).filter(_ % 2 == 0).toList
12 :: cons(14, Stream().map(_ + 10)).filter(_ % 2 == 0).toList
12 :: 14 :: Stream().map(_ + 10).filter(_ % 2 == 0).toList
12 :: 14 :: List()
```

Apply `filter` to the first element.

Apply `map` to the first element.

Apply `map` to the second element.

Apply `filter` to the second element. Produce the first element of the result.

Apply `filter` to the fourth element and produce the final element of the result.

`map` and `filter` have no more work to do, and the empty stream becomes the empty list.

04 | Creando Streams

indizen
Technology & Business Confluence

- 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 streams vacíos 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

05 | Ejercicios

indizen
Technology & Business Confluence

- 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
- Define la función exists que reciba una función de entrada y devuelva true si se cumple para alguno de los elementos.

06 | Fold

indizen
Technology & Business Confluence

> Implementa la función foldRight sobre Stream
  - def foldRight[B](z: => B)(f:(A, =>B) => B): B

> Implementa la función foldLeft sobre Stream
  - def foldLeft[B](z: => B)(f:(=> B, A) => B): B

> Define las funciones exists, forAll y headOption usando fold

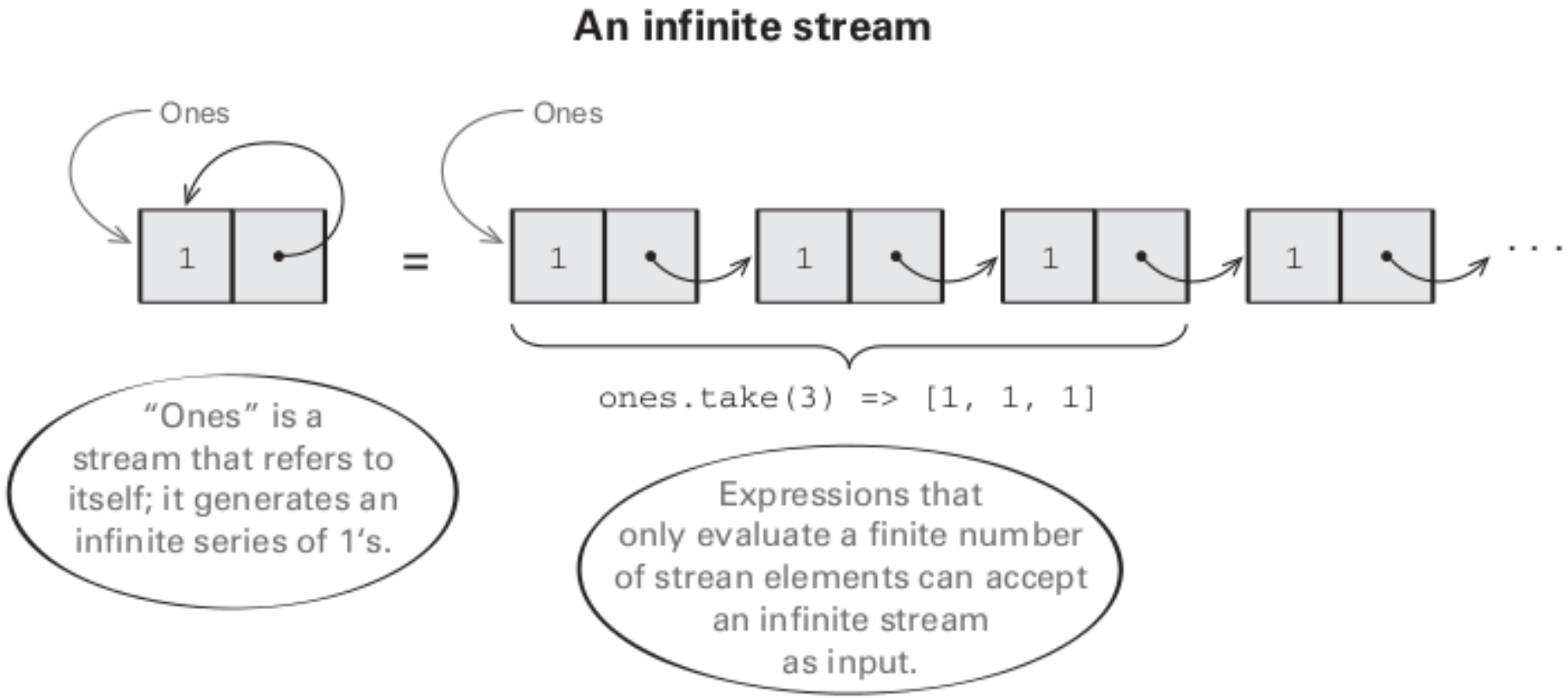
> Define map usando fold

> Define las funciones drop, take, dropWhile, takeWhile y filter usando fold

07 | Streams infinitos

indizen
Technology & Business Confluence

- Gracias a que no se computa la cola del Stream hasta que es necesario, podríamos definir Streams infinitos
- Aunque sean infinitos, las funciones evalúan unicamente la parte que necesitan:
  - ones.take(5)
  - ones.exists(_ % 2 != 0)
  - ones.map(_ + 1).exists(_ % 2 == 0)
  - ones.takeWhile(_ == 1)
  - ones.forAll(_!= 1)



Many functions can be evaluated using finite resources even if their inputs generate infinite sequences.

07 | Streams infinitos

indizen
Technology & Business Confluence

- Genera una función ones que devuelva un Stream infinito de unos
- Generaliza ones en una función constant que devuelva un Stream infinito del valor indicado

- Define una función from que genere un stream infinito 1 incremental de enteros, empezando en n. (n,n+1,n+2,n+3…)

- Define una función fibs que genere un stream infinito con la secuencia de fibonacci
    - 0,1,1,2,3,5,8...

07 | Streams infinitos

indizen
Technology & Business Confluence

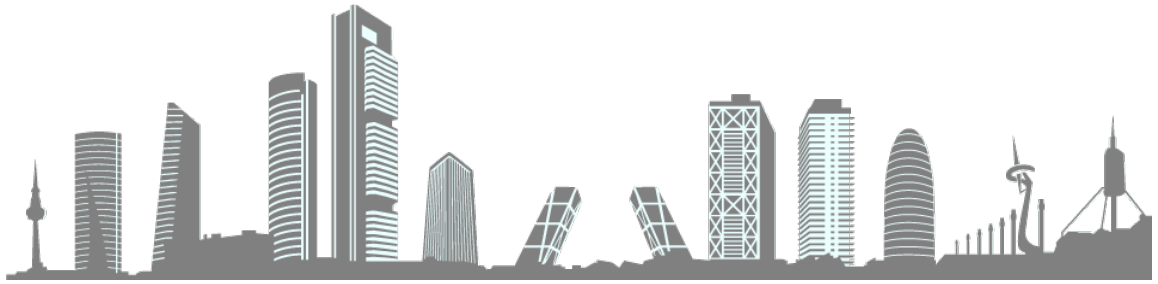
- Escribe una función unfold, que generalice la creación de streams. Tomará un estado inicial y una función para generarl el siguiente estado
  - def unfold[A,S](z: S)(f: S => Option[(A,S)]: Stream[A]

- Unfold es una función corecursiva (produce datos) (también guarded recursion)
- Función productiva o coterminativa => Genera datos hasta que f deje de devolver datos
- Ejercicio: Escribe las funciones ones, constant, from y fibs en base a unfold


"Con
Indizen
nunca caminarás solo"
indizen
Technology & Business Confluence

indizen

Technology & Business Confluence



GRACIAS

www.indizen.com