Fundamentos de Programación Funcional y Concurrente

Funciones de alto orden

Juan Francisco Díaz Frias

Profesor Titular (1993-hoy)
juanfco.diaz@correounivalle.edu.co
Edif. B13 - 4009


Universidad del Valle

Octubre 2022

Plan

Plan

Generalidades

- En general, los parámetros de las funciones no son solo números. Podrían ser también funciones.
- Frecuentemente el mismo patrón de programación es usado en diferentes funciones.
- Expresar esos patrones como conceptos implica poder pasar funciones como parámetros.
- Los lenguajes de programación funcional tratan las funciones como valores también. Se les denominan *valores de primera clase*
- Las funciones que reciben funciones como parámetro o que devuelven funciones como respuesta se denominan Funciones de alto orden

Generalidades

- En general, los parámetros de las funciones no son solo números. Podrían ser también funciones.
- Frecuentemente el mismo patrón de programación es usado en diferentes funciones.
- Expresar esos patrones como conceptos implica poder pasar funciones como parámetros.
- Los lenguajes de programación funcional tratan las funciones como valores también. Se les denominan *valores de primera clase*
- Las funciones que reciben funciones como parámetro o que devuelven funciones como respuesta se denominan Funciones de alto orden

Generalidades

- En general, los parámetros de las funciones no son solo números. Podrían ser también funciones.
- Frecuentemente el mismo patrón de programación es usado en diferentes funciones.
- Expresar esos patrones como conceptos implica poder pasar funciones como parámetros.
- Los lenguajes de programación funcional tratan las funciones como valores también. Se les denominan *valores de primera clase*
- Las funciones que reciben funciones como parámetro o que devuelven funciones como respuesta se denominan Funciones de alto orden

Generalidades

- En general, los parámetros de las funciones no son solo números. Podrían ser también funciones.
- Frecuentemente el mismo patrón de programación es usado en diferentes funciones.
- Expresar esos patrones como conceptos implica poder pasar funciones como parámetros.
- Los lenguajes de programación funcional tratan las funciones como valores también. Se les denominan *valores de primera clase*
- Las funciones que reciben funciones como parámetro o que devuelven funciones como respuesta se denominan Funciones de alto orden

Generalidades

- En general, los parámetros de las funciones no son solo números. Podrían ser también funciones.
- Frecuentemente el mismo patrón de programación es usado en diferentes funciones.
- Expresar esos patrones como conceptos implica poder pasar funciones como parámetros.
- Los lenguajes de programación funcional tratan las funciones como valores también. Se les denominan *valores de primera clase*
- Las funciones que reciben funciones como parámetro o que devuelven funciones como respuesta se denominan Funciones de alto orden

Ejemplo de trabajo

- Suponga que le piden calcular las siguientes tres sumas:

$$a + (a+1) + (a+2) + \ldots + b$$
$$a^2 + (a+1)^2 + (a+2)^2 + \ldots + b^2$$
$$a^2 + (a+2)^2 + (a+4)^2 + \ldots + c^2, (b-1) \leq c \leq b$$

- Y escribimos los siguientes tres programas en Scala:

```
0  def sumaEnteros (a:Int, b:Int):Int = if (a>b) 0 else a + sumaEnteros(a+1,b)
1  sumaEnteros(1,10)
2  def sumaCuadrados (a:Int, b:Int):Int = if (a>b) 0 else a*a + sumaCuadrados(a+1,b)
3  sumaCuadrados(1,10)
4  def sumaAlternada (a:Int, b:Int):Int = if (a>b) 0 else a + sumaAlternada(a+2,b)
5  sumaAlternada(1,10)
```

- Todos los casos anteriores son casos especiales para calcular

$$\sum_{i=a}^{b} f(i)$$

para diferentes casos de f.

Plan

Sumando con funciones de alto orden

- Factoricemos lo común en una función de Scala parametrizada en f:

```
0  def suma(f:Int => Int, prox:Int=> Int, a:Int, b:Int): Int =
1    if (a>b) 0
2    else f(a) + suma(f, prox, prox(a), b)
```

- Y hagamos nuestros cálculos a través de esa función, pasando los parámetros adecuados:

```
0  def ident(x:Int) = x
1  def cuadrado(x:Int) = x*x
2  def suc(x:Int) = x+1
3  def sum2(x:Int) = x+2
4  /* Sumar los enteros entre a y b */
5  suma(ident, suc, 1, 10)
6  /* Sumar los cuadrados de los enteros entre a y b */
7  suma(cuadrado, suc, 1, 10)
8  /* Sumar los enteros entre a y b de 2 en 2 */
9  suma(ident, sum2, 1, 10)
```

- O definamos y usemos cada una de las funciones solicitadas:

```
0  def sumaEnteros2 (a:Int, b:Int):Int = suma(ident, suc, a, b)
1  sumaEnteros2(1,10)
2  def sumaCuadrados2 (a:Int, b:Int):Int = suma(cuadrado, suc, a, b)
3  sumaCuadrados2(1,10)
4  def sumaAlternada2 (a:Int, b:Int):Int = suma(ident, sum2, a, b)
5  sumaAlternada2(1,10)
```

Tipos de las funciones

- Nótese que las funciones tienen tipo (como todos los datos).
- El tipo $A => B$ es el tipo de las funciones que reciben valores de tipo A como argumento y devuelven valores de tipo B como resultado
- Por ejemplo $Int => Int$ es el tipo de las funciones que reciben valores de tipo Int como argumento y devuelven valores de tipo Int como resultado
- O $(Int, Int) => Boolean$ es el tipo de las funciones que reciben dos valores de tipo Int como argumento y devuelven un valor de tipo $Boolean$ como resultado

Tipos de las funciones

- Nótese que las funciones tienen tipo (como todos los datos).
- El tipo $A => B$ es el tipo de las funciones que reciben valores de tipo A como argumento y devuelven valores de tipo B como resultado
- Por ejemplo $Int => Int$ es el tipo de las funciones que reciben valores de tipo Int como argumento y devuelven valores de tipo Int como resultado
- O $(Int, Int) => Boolean$ es el tipo de las funciones que reciben dos valores de tipo Int como argumento y devuelven un valor de tipo $Boolean$ como resultado

Tipos de las funciones

- Nótese que las funciones tienen tipo (como todos los datos).
- El tipo $A => B$ es el tipo de las funciones que reciben valores de tipo A como argumento y devuelven valores de tipo B como resultado
- Por ejemplo $Int => Int$ es el tipo de las funciones que reciben valores de tipo Int como argumento y devuelven valores de tipo Int como resultado
- O $(Int, Int) => Boolean$ es el tipo de las funciones que reciben dos valores de tipo Int como argumento y devuelven un valor de tipo $Boolean$ como resultado

Tipos de las funciones

- Nótese que las funciones tienen tipo (como todos los datos).
- El tipo $A => B$ es el tipo de las funciones que reciben valores de tipo A como argumento y devuelven valores de tipo B como resultado
- Por ejemplo $Int => Int$ es el tipo de las funciones que reciben valores de tipo *Int* como argumento y devuelven valores de tipo *Int* como resultado
- O $(Int, Int) => Boolean$ es el tipo de las funciones que reciben dos valores de tipo *Int* como argumento y devuelven un valor de tipo *Boolean* como resultado

Ejercicios

- La función *suma* genera un proceso recursivo lineal. Escribir una función *suma* que genere más bien un proceso iterativo lineal.
- Escriba una función *producto* análogo al procedimiento *suma*. Defina *factorial* en función de este nuevo procedimiento.

Plan

Funciones anónimas

- Pasar funciones como parámetro, nos hizo crear (nombrar) muchas funciones pequeñas nuevas (*suc*, *sum2*, *ident*, *cuadrado*). Eso es tedioso en algunas ocasiones.
- Miremos lo que pasa con otros datos, como por ejemplo, las cadenas. No necesitamos definir una cadena para poder imprimirla. En lugar de:

```
0  def cad="abc"; println(cad)
```

  podemos escribir directamente:

```
0  println("abc")
```

  porque las cadenas existen como *literales*
- Análogamente, uno quisiera tener *literales de funciones* que nos permitan escribirlas sin darles un nombre. Es una manera de escribir funciones anónimas.

Sintaxis de funciones anónimas

- Por ejemplo, para referirnos a una función que eleva al cubo, sin nombrarla, escribiremos:

```
0  (x:Int) => x*x*x
```

Nótese que $(x : Int)$ es el parámetro de la función y $x * x * x$ es su cuerpo. El tipo del parámetro puede ser omitido si el compilador lo puede inferir.

- Si hay varios parámetros, se separarán por comas:

```
0  (x:Int, y:Int) => (x+y)/2
```

- Una función anónima $(x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n) => E$ se puede expresar en el lenguaje de la manera siguiente:

```
0  def f(x_1:T_1, x_2:T_2, ..., x_n:T_n) => E; f
```

donde f es un nombre arbitrario, fresco, que nunca ha sido usado en el programa.

La suma con funciones anónimas

- El ejercicio de la suma lo podemos reescribir con funciones anónimas:

```
0  def sumaEnteros2 (a:Int, b:Int):Int = suma(x=>x, x=>x+1, a, b)
1  sumaEnteros2(1,10)
2  def sumaCuadrados2 (a:Int, b:Int):Int = suma(x=>x*x, x=>x+1, a, b)
3  sumaCuadrados2(1,10)
4  def sumaAlternada2 (a:Int, b:Int):Int = suma(x=>x, x=>x+2, a, b)
5  sumaAlternada2(1,10)
```

- Haga el mismo ejercicio con el producto.
- ¿Podría escribir una función, más general, que generalice las funciones *suma* y *producto* ?

Plan

El ejemplo de la suma de nuevo

- Miremos de nuevo el ejemplo de la *suma*:

```
0  def sumaEnteros2 (a:Int, b:Int):Int = suma(x=>x, x=>x+1, a, b)
1  def sumaCuadrados2 (a:Int, b:Int):Int = suma(x=>x*x, x=>x+1, a, b)
2  def sumaAlternada2 (a:Int, b:Int):Int = suma(x=>x, x=>x+2, a, b)
```

  Nótese que *a* y *b* pasan intactas de *sumaEnteros2*, *sumaCuadrados2*, *sumaAlternada2* a *suma*.

- ¿Se podría entonces escribir más corto lo mismo, teniendo en cuenta que esos parámetros no se tocan?

Funciones que devuelven funciones

- Reescribamos *suma* de la siguiente manera:

```
0  def suma2(f:Int => Int, prox:Int => Int):(Int, Int) => Int = {
1    def sumaF(a:Int, b:Int):Int = if (a>b) 0
2                                  else f(a) + sumaF(prox(a),b)
3    sumaF
4  }
```

suma ahora es una función que devuelve otra función (*sumaF*) como resultado.

- Podemos calcular las funciones *sumaEnteros2*, *sumaCuadrados2*, *sumaAlternada2* así:

```
0  def sumaEnteros2 = suma2(x=>x,x=>x+1 )
1  sumaEnteros2(1,10)
2  def sumaCuadrados2 = suma2(x=>x*x, x=>x+1)
3  sumaCuadrados2(1,10)
4  def sumaAlternada2 = suma2(x=>x, x=>x+2)
5  sumaAlternada2(1,10)
```

Currificación (del inglés *Currying*)

- Una práctica frecuente en la programación funcional es la denominada currificación
- Consiste en ver todas las funciones como funciones de un solo argumento. Mirémoslo con dos argumentos y luego generalizamos.
- Recordemos primero que en matemáticas se denota B^A como el conjunto de todas las funciones de $A \to B$:

$$B^A = \{g | g : A \to B\}$$

- Sea $f : A \times B \to C$ una función de dos argumentos, tal que $f(a, b) \in C$. Sea $fc : A \to C^B$ tal que $fc(a) = g_a$ y $g_a : B \to C$ es tal que $g_a(b) = f(a, b)$. Decimos que fc es la versión currificada de f.
- Nótese que:

$$f(a, b) = (fc(a))(b)$$

pero fc no sólo permite calcular f sino también otras funciones

$$\{fc(a) : a \in A\}$$

Currificación (del inglés *Currying*)

- Una práctica frecuente en la programación funcional es la denominada currificación
- Consiste en ver todas las funciones como funciones de un solo argumento. Mirémoslo con dos argumentos y luego generalizamos.
- Recordemos primero que en matemáticas se denota B^A como el conjunto de todas las funciones de $A \to B$:

$$B^A = \{g | g : A \to B\}$$

- Sea $f : A \times B \to C$ una función de dos argumentos, tal que $f(a, b) \in C$. Sea $fc : A \to C^B$ tal que $fc(a) = g_a$ y $g_a : B \to C$ es tal que $g_a(b) = f(a, b)$. Decimos que fc es la versión currificada de f.
- Nótese que:

$$f(a, b) = (fc(a))(b)$$

pero fc no sólo permite calcular f sino también otras funciones

$$\{fc(a) : a \in A\}$$

Currificación (del inglés *Currying*)

- Una práctica frecuente en la programación funcional es la denominada currificación
- Consiste en ver todas las funciones como funciones de un solo argumento. Mirémoslo con dos argumentos y luego generalizamos.
- Recordemos primero que en matemáticas se denota B^A como el conjunto de todas las funciones de $A \to B$:

$$B^A = \{g | g : A \to B\}$$

- Sea $f : A \times B \to C$ una función de dos argumentos, tal que $f(a, b) \in C$. Sea $fc : A \to C^B$ tal que $fc(a) = g_a$ y $g_a : B \to C$ es tal que $g_a(b) = f(a, b)$. Decimos que fc es la versión currificada de f.
- Nótese que:

$$f(a, b) = (fc(a))(b)$$

pero fc no sólo permite calcular f sino también otras funciones

$$\{fc(a) : a \in A\}$$

Currificación (del inglés *Currying*)

- Una práctica frecuente en la programación funcional es la denominada currificación
- Consiste en ver todas las funciones como funciones de un solo argumento. Mirémoslo con dos argumentos y luego generalizamos.
- Recordemos primero que en matemáticas se denota B^A como el conjunto de todas las funciones de $A \to B$:

$$B^A = \{g | g : A \to B\}$$

- Sea $f : A \times B \to C$ una función de dos argumentos, tal que $f(a, b) \in C$. Sea $fc : A \to C^B$ tal que $fc(a) = g_a$ y $g_a : B \to C$ es tal que $g_a(b) = f(a, b)$. Decimos que fc es la versión currificada de f.
- Nótese que:

$$f(a, b) = (fc(a))(b)$$

pero fc no sólo permite calcular f sino también otras funciones

$$\{fc(a) : a \in A\}$$

Currificación (del inglés *Currying*)

- Una práctica frecuente en la programación funcional es la denominada currificación
- Consiste en ver todas las funciones como funciones de un solo argumento. Mirémoslo con dos argumentos y luego generalizamos.
- Recordemos primero que en matemáticas se denota B^A como el conjunto de todas las funciones de $A \to B$:

$$B^A = \{g | g : A \to B\}$$

- Sea $f : A \times B \to C$ una función de dos argumentos, tal que $f(a, b) \in C$. Sea $fc : A \to C^B$ tal que $fc(a) = g_a$ y $g_a : B \to C$ es tal que $g_a(b) = f(a, b)$. Decimos que fc es la versión currificada de f.
- Nótese que:

$$f(a, b) = (fc(a))(b)$$

pero fc no sólo permite calcular f sino también otras funciones

$$\{fc(a) : a \in A\}$$

Ejemplo sencillo de currificación

- Sea $suma : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ la función de dos argumentos, tal que $suma(a, b) = a + b$.
- Entonces $sumac : \mathbb{N} \to \mathbb{N}^{\mathbb{N}}$ es la función tal que $sumac(a) = g_a$ y $g_a : \mathbb{N} \to \mathbb{N}$ es tal que $g_a(b) = suma(a, b) = a + b$.

  O sea, $sumac(a)$ es la función que le suma a a cualquier número natural.
- $$suma(a, b) = (sumac(a))(b)$$

  pero *sumac* no sólo permite calcular *suma* sino también otras funciones:
  - $sumac(2)$ es la función que suma 2: $sumac(2)(b) = 2 + b$
  - $sumac(5)$ es la función que suma 5: $sumac(5)(b) = 5 + b$

Ejemplo sencillo de currificación

- Sea $suma : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ la función de dos argumentos, tal que $suma(a, b) = a + b$.
- Entonces $sumac : \mathbb{N} \to \mathbb{N}^{\mathbb{N}}$ es la función tal que $sumac(a) = g_a$ y $g_a : \mathbb{N} \to \mathbb{N}$ es tal que $g_a(b) = suma(a, b) = a + b$.

  O sea, $sumac(a)$ es la función que le suma a a cualquier número natural.
- $$suma(a, b) = (sumac(a))(b)$$

  pero *sumac* no sólo permite calcular *suma* sino también otras funciones:
  - $sumac(2)$ es la función que suma 2: $sumac(2)(b) = 2 + b$
  - $sumac(5)$ es la función que suma 5: $sumac(5)(b) = 5 + b$

Ejemplo sencillo de currificación

- Sea $suma : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ la función de dos argumentos, tal que $suma(a, b) = a + b$.
- Entonces $sumac : \mathbb{N} \to \mathbb{N}^{\mathbb{N}}$ es la función tal que $sumac(a) = g_a$ y $g_a : \mathbb{N} \to \mathbb{N}$ es tal que $g_a(b) = suma(a, b) = a + b$.

  O sea, $sumac(a)$ es la función que le suma a a cualquier número natural.
- $$suma(a, b) = (sumac(a))(b)$$

  pero *sumac* no sólo permite calcular *suma* sino también otras funciones:
  - $sumac(2)$ es la función que suma 2: $sumac(2)(b) = 2 + b$
  - $sumac(5)$ es la función que suma 5: $sumac(5)(b) = 5 + b$

Ejemplo en Scala con la suma de cuadrados y de enteros

- Visitemos de nuevo el ejemplo que veníamos trabajando. La versión currificada quedaría así:

```
0  def suma4(f:Int => Int) (prox:Int => Int)(a:Int, b:Int): Int =
1    if (a>b) 0
2    else f(a) + suma4(f)(prox)(prox(a),b)
3  suma4(x=>x) _
4  suma4(x=>x)(x=>x+1) _
5  suma4(x=>x)(x=>x+1)(1,10)
6  suma4(x=>x*x)(x=>x+1)(1,10)
7  suma4(x=>x)(x=>x+2)(1,10)
```

- Obsérvese la notación especial en Scala:

```
0  def f(arg1)(arg2)...(argn) = E
```

que realmente es azúcar sintáctico de:

```
0  def f(arg1)(arg2)...(arg_(n-1)) = (argn => E)
```

y finalmente (después de repetir n veces):

```
0  def f =(arg1 (=> (arg2 => ... => (arg_(n-1) => (argn => E)) ... )))
```

Algo más sobre tipos de las funciones

- ¿Cuál es el tipo de *suma4*?

```
0  def suma4(f:Int => Int) (prox:Int => Int)(a:Int, b:Int): Int =
1    if (a>b) 0
2    else f(a) + suma4(f)(prox)(prox(a),b)
```

$$suma4 : (Int => Int) => ((Int => Int) => ((Int, Int) => Int))$$

- Nótese que los tipos de funciones son asociativos a la derecha:

$$Int => Int => Int$$

es equivalente a

$$Int => (Int => Int)$$

Funciones como respuesta

- Considere el problema de, dada una función $f : \mathbb{R} \to \mathbb{R}$ calcular su derivada f':

$$f'(x) \approx \frac{f(x + dx) - f(x)}{dx}, dx \to 0$$

- Una solución sencilla en Scala sería implementar una función *derivada* que reciba una función de entrada, y devuelva la derivada como salida:

```
0  def derivada (f:Double => Double,dx:Double) = (x:Double) => (f(x+dx) - f(x))/dx
```

- Luego se puede calcular la derivada de cualquier función:

```
0  def cube(x:Double)= x*x*x
1  def cubeD = derivada(cube, 0.0001)
2  cubeD(1)
3  cubeD(2)
4  cubeD(3)
```