

Taller 1: Funciones y Procesos

Fundamentos de Programación Funcional y Concurrente

Veronica Lorena Mujica Gavidia - 2359406
Jeidy Nicol Murillo Murillo - 2359310
Juan Manuel Perez Cruz - 2266033

Septiembre 2024

1 Informe de Procesos

En esta sección se analizarán los procesos generados por los programas recursivos implementados. Tal como se ha visto en clase, los programas recursivos generan una pila de llamadas que se despliega y se resuelve de manera controlada a medida que se ejecutan las operaciones. Para ilustrar este comportamiento, se presentarán ejemplos específicos para cada ejercicio.

1.1 Máximo de una lista de enteros: Ejemplo

1.1.1 Recursión Lineal

Primero, establecemos el test que utilizaremos para depurar la función `maxLin`. Para ello, proporcionamos una lista de valores y comenzamos el proceso de depuración. Esto nos permitirá observar detalladamente la ejecución de la función y verificar su comportamiento.

```
// Lista de varios valores mezclados: Recursion Lineal
test("Numero maximo de la Lista (10, 7, 8, 2) es 10") {
  assert(objMaxlist.maxLin(List(10, 7, 8, 2)) == 10)
}

// Función recursión lineal
def maxLin(l: List[Int]): Int = {
  if (l.isEmpty) throw new IllegalArgumentException("La lista no debe estar vacía")
  if (l.exists(_ < 0)) throw new IllegalArgumentException("La lista no debe contener números negativos")
  if (l.tail.isEmpty) l.head
  else math.max(l.head, maxLin(l.tail))
}
```

Paso 1: `maxLin(List(10, 7, 8, 2))`

- La lista no está vacía y no contiene números negativos.
- La lista no tiene un solo elemento, así que se llama a la recursión: `math.max(10, maxLin(List(7, 8, 2)))`. Teniendo dos llamados de pila iniciales, uno para el número entero que está almacenando y el otro para el `assert` del Test.

```
EjercicioMaxlist.maxLin(List): int EjercicioMaxlist.scala 12:1
EjercicioMaxlistTest.$anonfun$new$13(EjercicioMaxlistTest): Assertion EjercicioMaxlistTest.sc...
```

```
Local
> l = $colon$colon@1955 "List(10, 7, 8, 2)"
> this = EjercicioMaxlist@1956
```

Paso 2: `maxLin(List(7, 8, 2))`

- La lista no está vacía y no contiene números negativos.
- La lista no tiene un solo elemento, así que se llama a la recursión: `math.max(7, maxLin(List(8, 2)))`. Los llamados de la pila se incrementan en uno para guardar el nuevo número entero.

```
VARIABLES
Local
> l = $colon$colon@1959 "List(7, 8, 2)"
  serialVersionUID = 3
> head = Integer@1968 "7"
> next = $colon$colon@1969 "List(8, 2)"
> this = EjercicioMaxlist@1956
```

```
EjercicioMaxlist.maxLin(List): int      EjercicioMaxlist.scala 12:1
EjercicioMaxlist.maxLin(List): int      EjercicioMaxlist.scala 12:1
EjercicioMaxlistTest.$anonfun$new$13(EjercicioMaxlistTest): Assertion Ejerc...
```

Paso 3: maxLin(List(8, 2))

- La lista no está vacía y no contiene números negativos.
- La lista no tiene un solo elemento, así que se llama a la recursión: math.max(8, maxLin(List(2))).

```

VARIABLES
  Local
    l = $colon$colon@1969 "List(8, 2)"
      serialVersionUID = 3
      > head = Integer@1978 "8"
      > next = $colon$colon@1979 "List(2)"
      > this = EjercicioMaxlist@1956

```

```
EjercicioMaxlist.maxLin(List): int      EjercicioMaxlist.scala 12:1
EjercicioMaxlist.maxLin(List): int      EjercicioMaxlist.scala 12:1
EjercicioMaxlist.maxLin(List): int      EjercicioMaxlist.scala 12:1
EjercicioMaxlistTest.$anonfun$new$13(EjercicioMaxlistTest): Assertion Ejerc...
```

Paso 4: maxLin(List(2))

- La lista no está vacía y no contiene números negativos.
- La lista tiene un solo elemento, así que devuelve 2.

Paso de regreso:

- En el paso 3, math.max(8, 2) devuelve 8.
- En el paso 2, math.max(7, 8) devuelve 8.
- En el paso 1, math.max(10, 8) devuelve 10. El resultado final para maxLin(List(10, 7, 8, 2)) es 10.

```

  Local
    ->maxLin() = 8
    l = $colon$colon@1962 "List(7, 8, 2)"
      serialVersionUID = 3
      > head = Integer@1967 "7"
      > next = $colon$colon@1964 "List(8, 2)"
      > this = EjercicioMaxlist@1956

```

```

  Local
    ->maxLin() = 8
    l = $colon$colon@1955 "List(10, 7, 8, 2)"
      serialVersionUID = 3
      > head = Integer@1973 "10"
      > next = $colon$colon@1962 "List(7, 8, 2)"
      > this = EjercicioMaxlist@1956

```

```

  Local
    ->maxLin() = 10
    > $this = EjercicioMaxlistTest@1976 "EjercicioMaxlistTest"

```

1.1.2 Recursión de Cola

Para ilustrar cómo funciona la función, consideremos la misma lista del ejemplo anterior (10, 7, 8, 2) y en este caso objMaxlist.maxIt:

```
// Lista de varios valores mezclados: Recursion de Cola
test("Numero maximo de la Lista (10, 7, 8, 2) es 10") {
  |   assert(objMaxlist.maxIt(List(10, 7, 8, 2)) == 10)
  |
}
```

Paso 1: maxIt(List(10, 7, 8, 2))

-La lista no está vacía y no contiene números negativos.

-Se llama a maxItAux con maxItAux(List(10, 7, 8, 2), Int.MinValue).

```
Local
  l = $colon$colon@1955 "List(10, 7, 8, 2)"
    serialVersionUID = 3
    > head = Integer@1962 "10"
    > next = $colon$colon@1963 "List(7, 8, 2)"
    > this = EjercicioMaxlist@1956
```

```
EjercicioMaxlist.maxIt(List): int          EjercicioMaxlist.scala 25:1
EjercicioMaxlistTest.$anonfun$new$13(EjercicioMaxlistTest): Assertion
```

Paso 2: maxItAux(List(10, 7, 8, 2), Int.MinValue)

-l.head es 10, max es Int.MinValue.

-math.max(10, Int.MinValue) es 10.

-Se llama a maxItAux con maxItAux(List(7, 8, 2), 10).

```
Local
  l = $colon$colon@1955 "List(10, 7, 8, 2)"
    serialVersionUID = 3
    > head = Integer@1962 "10"
    > next = $colon$colon@1963 "List(7, 8, 2)"
    max = -2147483648
```

```
EjercicioMaxlist.maxItAux$l(List,int): int  EjercicioMaxlist.scala 20:1
EjercicioMaxlist.maxIt(List): int          EjercicioMaxlist.scala 25:1
EjercicioMaxlistTest.$anonfun$new$13(EjercicioMaxlistTest): Assertion
```

Paso 3: maxItAux(List(7, 8, 2), 10)

-l.head es 7, max es 10.

-math.max(7, 10) es 10.

-Se llama a maxItAux con maxItAux(List(8, 2), 10).

```
Local
  l = $colon$colon@1963 "List(7, 8, 2)"
    serialVersionUID = 3
    > head = Integer@1972 "7"
    > next = $colon$colon@1973 "List(8, 2)"
    max = 10
    > this = EjercicioMaxlist@1956
```

```
EjercicioMaxlist.maxItAux$l(List,int): int  EjercicioMaxlist.scala 19:1
EjercicioMaxlist.maxIt(List): int          EjercicioMaxlist.scala 25:1
EjercicioMaxlistTest.$anonfun$new$13(EjercicioMaxlistTest): Assertion
```

Paso 4: `maxItAux(List(8, 2), 10)`

-`l.head` es 8, `max` es 10.

-`math.max(8, 10)` es 10.

-Se llama a `maxItAux` con `maxItAux(List(2), 10)`.

```
Local
  l = $colon$colon@1973 "List(8, 2)"
    serialVersionUID = 3
  > head = Integer@1980 "8"
  > next = $colon$colon@1981 "List(2)"
  max = 10
  > this = EjercicioMaxlist@1956
```

```
EjercicioMaxlist.maxItAux$(List,int): int      EjercicioMaxlist.scala 19:1
EjercicioMaxlist.maxIt(List): int             EjercicioMaxlist.scala 25:1
EjercicioMaxlistTest.$anonfun$new$13(EjercicioMaxlistTest): Assertion 25:1
```

Paso 5: `maxItAux(List(2), 10)`

-`l.head` es 2, `max` es 10.

-`math.max(2, 10)` es 10.

-Se llama a `maxItAux` con `maxItAux(List(), 10)`.

```
Local
  l = $colon$colon@1981 "List(2)"
    serialVersionUID = 3
  > head = Integer@1988 "2"
  > next = Nil$@1989 "List()"
  max = 10
  > this = EjercicioMaxlist@1956
```

```
EjercicioMaxlist.maxItAux$(List,int): int      EjercicioMaxlist.scala 19:1
EjercicioMaxlist.maxIt(List): int             EjercicioMaxlist.scala 25:1
EjercicioMaxlistTest.$anonfun$new$13(EjercicioMaxlistTest): Assertion 25:1
```

Paso 6: `maxItAux(List(), 10)`

-La lista está vacía.

-Devuelve 10, que es el valor máximo encontrado.

```
Local
  l = Nil$@1989 "List()"
  > EmptyUnzip = Tuple2@1996 "(List(),List())"
  > MODULE$ = Nil$@1989 "List()"
    serialVersionUID = 3
  max = 10
  > this = EjercicioMaxlist@1956
```

1.2 Torres de Hanoi: Ejemplo

1.2.1 Movimientos Torres de Hanoi

La clase `TorresHanoi` contiene una función llamada `movsTorresHanoi` que calcula el número mínimo de movimientos necesarios para resolver el problema de las Torres de Hanoi con n discos con la fórmula `BigInt(2).pow(n)` la cual calcula 2 elevado a la potencia n .

```
3 class TorresHanoi {
4     def movsTorresHanoi(n: Int): BigInt = {
5         if (n == 0) BigInt(0)
6     else BigInt(2).pow(n) - 1
7     }
```

Para realizar la depuración tomamos el siguiente test de prueba:

```
// Pruebas para movsTorresHanoi
test("movsTorresHanoi con 3 discos debe devolver 7") {
    assert(torresHanoi.movsTorresHanoi(3) == BigInt(7))
}
```

Paso 1: `movsTorresHanoi(3)`

- $n = 3$, por lo que no se entra en el caso base.

-Calculamos `BigInt(2).pow(3) - 1 = 8 - 1 = 7`.

-Así, el número mínimo de movimientos con 3 discos es 7.

The screenshot shows the IDE's variables pane. At the top, under 'Local', the variable `n` is set to `3`, and `this` is `TorresHanoi@1952`. Below this, another section shows the state of the function `movsTorresHanoi()` at line 1957, which has returned the value `"7"`. Other variables shown include `serialVersionUID` (a long number), `_bigInteger` (null), and `_long` (7).

1.2.2 Torres de Hanoi

Esta función calcula la secuencia de movimientos para resolver el problema de las Torres de Hanoi.

Parámetros:

- n : El número de discos.

- $t1$: La torre de origen.

- $t2$: La torre de destino intermedia.

- $t3$: La torre de destino final.

Vamos a depurar la función con el test:

```
// Pruebas para torresHanoi
test("Torre Hanoi con 2 discos debe devolver los movimientos correctos") {
    assert(torresHanoi.torresHanoi(2, 1, 2, 3) == List((1, 2), (1, 3), (2, 3)))
}
```

Vamos a ejecutar la función con $n = 2$, $t1 = 1$, $t2 = 2$, y $t3 = 3$:

```
▼ Local
  n = 2
  t1 = 1
  t2 = 2
  t3 = 3
```

```
TorresHanoi.torresHanoi(int,int,int,int): List TorresHa...
TorresHanoiTest.$anonfun$new$8(TorresHanoiTest): Assertion
```

Paso 1: torresHanoi(2, 1, 2, 3)

-No es el caso base, así que pasamos al else.

Paso 2 (movimientosPrevios):

a) Llamamos a torresHanoi(1, 1, 3, 2):

```
▼ VARIABLES
  ▼ Local
    n = 1
    t1 = 1
    t2 = 3
    t3 = 2
```

```
TorresHanoi.torresHanoi(int,int,int,int): List TorresHa...
TorresHanoi.torresHanoi(int,int,int,int): List TorresHa...
TorresHanoiTest.$anonfun$new$8(TorresHanoiTest): Assertion
```

b) Movemos el disco 1 de la torre 1 a la torre 2 $\rightarrow (1, 2)$.

```
▼ Local
  n = 1
  t1 = 1
  t2 = 3
  t3 = 2
  > movimientosPrevios = Nil@1957 "List()"
  > moverDisco = Tuple2$mcII$sp@1960 "(1,2)"
  > this = TorresHanoi@1956
```

```
▼ Local
  > ->torresHanoi() = $colon$colon@1967 "List((1,2))"
  n = 2
  t1 = 1
  t2 = 2
  t3 = 3
  > this = TorresHanoi@1956
```

```
TorresHanoi.torresHanoi(int,int,int,int): List TorresHa...
TorresHanoiTest.$anonfun$new$8(TorresHanoiTest): Assertion
```

c) movimientosPrevios = List((1, 2)).

```
▼ Local
  n = 2
  t1 = 1
  t2 = 2
  t3 = 3
  > movimientosPrevios = $colon$colon@1967 "List((1,2))"
  > this = TorresHanoi@1956
```

Paso 3 (moverDisco):

-Movemos el disco 2 de la torre 1 a la torre 3 $\rightarrow (1, 3)$.

```
▼ Local
  n = 2
  t1 = 1
  t2 = 2
  t3 = 3
  > movimientosPrevios = $colon$colon@1967 "List((1,2))"
  > moverDisco = Tuple2$mcII$sp@1970 "(1,3)"
  > this = TorresHanoi@1956
```

Paso 4 (movimientosPosteriores):

a) Llamamos a torresHanoi(1, 2, 1, 3):

```
▼ Local
  n = 1
  t1 = 2
  t2 = 1
  t3 = 3
  > this = TorresHanoi@1952
```

```
TorresHanoi.torresHanoi(int,int,int,int): List TorresHa...
TorresHanoi.torresHanoi(int,int,int,int): List TorresHa...
TorresHanoiTest.$anonfun$new$8(TorresHanoiTest): Assertion
```

b) Movemos el disco 1 de la torre 2 a la torre 3 $\rightarrow (2, 3)$.

```
> movimientosPrevios = Nil$@1957 "List()"
> moverDisco = Tuple2$mcII$sp@1975 "(2,3)"
> this = TorresHanoi@1952
```

Paso 5: Combinar los movimientos:

-Concatenamos movimientosPrevios ::: (moverDisco :: movimientosPosteriores):

-List((1, 2)) ::: ((1, 3) :: List((2, 3))) = List((1, 2), (1, 3), (2, 3)).

```
▼ Local
  > ->torresHanoi() = $colon$colon@1989 "List((1,2), (1,3), (2,3))"
  > $this = TorresHanoiTest@1990 "TorresHanoiTest"
```


2 Informe de Corrección

2.1 Análisis de las correcciones en maxLin y maxIt

Queremos calcular el máximo de una lista no vacía de enteros positivos usando recursión. En este caso, tenemos dos versiones: una con recursión directa (maxLin) y otra con recursión de cola (maxIt)

2.1.1 Código de MaxLin

```
// Función recursión lineal
def maxLin(l: List[Int]): Int = {
  if (l.isEmpty) throw new IllegalArgumentException("La lista no debe estar vacía")
  if (l.exists(_ < 0)) throw new IllegalArgumentException("La lista no debe contener números negativos")
  if (l.tail.isEmpty) l.head
  else math.max(l.head, maxLin(l.tail))
}
```

Definimos la función de manera matemática:

$$f : list[N] \rightarrow N \quad (1)$$

donde $f(l)$ toma una lista de enteros $l=[a_1, a_2, \dots, a_n]$ y devuelve el máximo elemento de la lista.

Revisión: Se comprueba que la lista no esté vacía y que no contenga elementos negativos, el caso base está bien implementado puesto que si la lista tiene un solo elemento, simplemente se devuelve ese elemento. La recursividad está bien implementada, donde en cada llamada se evalúa el primer elemento y se compara con el máximo de los elementos restantes.

Corrección Matemática (Inducción Estructural):

Caso Base: Cuando la lista tiene un solo elemento:

$$f([a_1]) = a_1 \quad (2)$$

La función en Scala se evalúa como: `if (l.tail.isEmpty) l.head` Esto implica que cuando la lista tiene un solo elemento, la función devuelve el primer (y único) elemento.

Paso inductivo: para n mayor que 1, se define como:

$$f([a_1, a_2, \dots, a_n]) = \max(a_1, f([a_2, \dots, a_n])) \quad (3)$$

El código que utiliza `math.max` para comparar el primer elemento con el máximo de los elementos restantes. Esto significa que la función toma el primer elemento a_1 y lo compara con el máximo de los elementos restantes $f([a_2, \dots, a_{n+1}])$.

2.1.2 Código de MaxIt

```
// Función recursión de cola
def maxIt(l: List[Int]): Int = {
  @tailrec
  def maxItAux(l: List[Int], max: Int): Int = {
    if (l.isEmpty) max
    else maxItAux(l.tail, math.max(l.head, max))
  }

  if (l.isEmpty) throw new IllegalArgumentException("La lista no debe estar vacía")
  if (l.exists(_ < 0)) throw new IllegalArgumentException("La lista no debe contener números negativos")
  else maxItAux(l, Int.MinValue)
}
```

Revisión: El código hace uso de la anotación `@tailrec`, que asegura que la función sea optimizada por el compilador para evitar desbordamiento de pila. Esto cumple con el requerimiento de recursión de cola, se inicializa el acumulador con `Int.MinValue`, que es un valor adecuado para comenzar a buscar el máximo en una lista de enteros, en cada paso, la función `maxItAux` compara el acumulador con el siguiente elemento de la lista y actualiza el máximo.

Corrección Matemática:

Estado inicial:

$$max0 = a1 \quad (4)$$

Invariante del ciclo:

$$maxi = \max(a1, a2, \dots, ai) \quad (5)$$

Estado final: Cuando la lista está vacía, el acumulador contiene el valor máximo de la lista.

2.2 Análisis de las correcciones en `movsTorresHanoi` y `torresHanoi`

2.2.1 Código de `movsTorresHanoi`

```

class TorresHanoi {
  def movsTorresHanoi(n: Int): BigInt = {
    if (n == 0) BigInt(0)
    else BigInt(2).pow(n) - 1
  }
}

```

Revisión: La fórmula correcta para el número mínimo de movimientos en las Torres de Hanoi está bien implementada:

$$f(n) = 2^n - 1 \quad (6)$$

Esto cumple con los requisitos matemáticos y el código es eficiente al utilizar `BigInt` para evitar problemas con valores grandes cuando `n` es grande.

Corrección Matemática:

Caso Base: Cuando `n=1`, se requiere un solo movimiento:

$$f(1) = 2^1 - 1 = 1 \quad (7)$$

Paso Inductivo: para `n = k + 1`:

$$f(k+1) = 2^k - 1 + 1 + 2^k - 1 = 2^k + 1 - 1 \quad (8)$$

2.2.2 Código de torresHanoi

```
def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {  
  if (n == 0) Nil  
  else {  
    // Mover n-1 discos de t1 a t2 usando t3 como intermediaria  
    val movimientosPrevios = torresHanoi(n - 1, t1, t3, t2)  
  
    // Mover el disco n de t1 a t3  
    val moverDisco = (t1, t3)  
  
    // Mover n-1 discos de t2 a t3 usando t1 como intermediaria  
    val movimientosPosteriores = torresHanoi(n - 1, t2, t1, t3)  
  
    movimientosPrevios ::: (moverDisco :: movimientosPosteriores)  
  }  
}
```

Revisión: La función implementa el algoritmo recursivo para resolver las Torres de Hanoi, en cada paso: Se mueve $n-1$ discos de la torre t_1 a la torre t_2 , se mueve el disco n de t_1 a t_3 , se mueven los $n - 1$ discos de t_2 a t_3 , el uso de listas y la concatenación con $::$ es eficiente para construir la lista de movimientos.

Corrección Matemática:

Caso Base: Cuando $n=1$, el movimiento es simple:

$$\text{Movimientos} = [(t_1, t_3)] \quad (9)$$

Paso Recursivo: para n mayor que 1, se descompone en 3 pasos:

$$\text{torresHanoi}(n, t_1, t_2, t_3) = \text{torresHanoi}(n-1, t_1, t_3, t_2) [(t_1, t_3)] \text{torresHanoi}(n-1, t_2, t_1, t_3) \quad (10)$$

3 Conclusiones

Este taller nos permitió comprender mejor el tema de recursividad en programación funcional a través del análisis de dos problemas específicos: encontrar el máximo de una lista de enteros positivos no vacía y la resolución del enigma de las Torres de Hanoi.

Hemos contemplado cómo los algoritmos recursivos en programación funcional descomponen los problemas complejos en subproblemas más pequeños, y cómo la recursión de cola ofrece una solución más eficiente al evitar la acumulación de llamadas en la pila.

Además, el uso de herramientas formales como la inducción y la inducción estructural, junto con los invariantes de ciclo, nos permitió garantizar que nuestras soluciones son correctas para todos los casos posibles.

Estas técnicas no solo son importantes en programación funcional, sino que también son fundamentales para escribir programas robustos y confiables en cualquier paradigma. Finalmente, hemos aprendido que la recursión en programación es una herramienta necesaria para resolver problemas de manera ágil. A través de la evaluación y el análisis de los procesos, podemos abordar problemas complejos de manera sistemática y eficiente.