

Informe taller 1

Docente

DELGADO SAAVEDRA CARLOS ANDRES

Estudiantes

Juan David García Arroyave-2359450

Sebastián Zacipa Martínez-2359695

Juan José Hincapié Tascón-2359493

Informe de procesos

Análisis del Programa 1: maxList

Definición del problema

La función maxList se encarga de encontrar el valor máximo de una lista de enteros usando recursividad.

Código de la función maxList:

```

1  package taller
2
3  import scala.annotation.tailrec
4
5
6  class maxlist {
7      def maxLin(l:List[Int] ): Int = {
8          if(l.isEmpty)
9              throw new IllegalArgumentException("la lista esta vacia")
10         else if (l.tail.isEmpty)
11             l.head
12         else {
13             val maxRest = maxLin(l.tail)
14             if(l.head > maxRest ) l.head else maxRest
15         }
16     }

```

Descripción del proceso

Si la lista está vacía, la función advierte que está vacía.

Si la lista contiene solo un elemento

En el caso general, la función compara el primer elemento de la lista con el valor máximo del resto de la lista

Ejemplo de ejecución

Consideremos la lista List(3, 7, 2, 9, 5).

1. **Llamada inicial:** maxLin(List(3, 7, 2, 9, 5))
 Compara 3 con maxLin(List(7, 2, 9, 5))
2. **2 llamada:** maxLin(List(7, 2, 9, 5))

Compara 7 con maxLin(List(2, 9, 5))

3. **3 llamada:** maxLin(List(2, 9, 5))

Compara 2 con maxLin(List(9, 5))

4. **Cuarta llamada:** maxLin(List(9, 5))

Compara 9 con maxLin(List(5))

5. **Quinta llamada:** maxLin(List(5))

Como la lista tiene un solo elemento, retorna 5.

Despliegue de la pila de llamadas

Una vez que se llega a la lista de un solo elemento, la función comienza a retornar resultados:

- maxLin(List(5)) retorna 5.
- maxLin(List(9, 5)) compara 9 con 5 y retorna 9.
- maxLin(List(2, 9, 5)) compara 2 con 9 y retorna 9.
- maxLin(List(7, 2, 9, 5)) compara 7 con 9 y retorna 9.
- maxLin(List(3, 7, 2, 9, 5)) compara 3 con 9 y retorna 9.

Análisis del Programa 2: maxIt (Recursividad de Cola)

La función maxIt también se encarga de encontrar el valor máximo de una lista de enteros

```
18
19  def maxIt(l: List[Int]): Int = {
20    if (l.isEmpty)
21      throw new IllegalArgumentException("La lista no puede estar vacía")
22    else {
23      @tailrec
24      def maxItAux(l: List[Int], max: Int): Int = {
25        if (l.isEmpty)
26          max
27        else {
28          val newMax = if (l.head > max) l.head else max
29          maxItAux(l.tail, newMax)
30        }
31      }
32      maxItAux(l.tail, l.head)
33    }
34  }
```

Descripción del proceso

1. Si la lista está vacía, lanza una excepción.

2. Si la lista no está vacía, utiliza una función auxiliar, `maxItAux`, que recibe la lista restante y el valor máximo acumulado hasta el momento.
3. En cada llamada, la función compara el primer elemento de la lista con el valor máximo acumulado y actualiza el máximo si es necesario.

Ejemplo de ejecución

Usamos la misma lista de ejemplo `List(3, 7, 2, 9, 5)`.

1. **Llamada inicial:** `maxIt(List(3, 7, 2, 9, 5))`
 - Llama a `maxItAux(List(7, 2, 9, 5), 3)`.
2. **Primera llamada a `maxItAux`:** `maxItAux(List(7, 2, 9, 5), 3)`
 - Compara 7 con 3, actualiza `newMax = 7`, y llama a `maxItAux(List(2, 9, 5), 7)`.
3. **Segunda llamada a `maxItAux`:** `maxItAux(List(2, 9, 5), 7)`
 - Compara 2 con 7, `newMax` sigue siendo 7, y llama a `maxItAux(List(9, 5), 7)`.
4. **Tercera llamada a `maxItAux`:** `maxItAux(List(9, 5), 7)`
 - Compara 9 con 7, actualiza `newMax = 9`, y llama a `maxItAux(List(5), 9)`.
5. **Cuarta llamada a `maxItAux`:** `maxItAux(List(5), 9)`
 - Compara 5 con 9, `newMax` sigue siendo 9, y llama a `maxItAux(List(), 9)`.
6. **Quinta llamada a `maxItAux`:** `maxItAux(List(), 9)`
 - Como la lista está vacía, retorna 9.

Informe de Procesos: Resolución del Problema de las Torres de Hanoi

Análisis del Programa 1: `movsTorresHanoi`

Definición del Problema

La función `movsTorresHanoi` calcula el número mínimo de movimientos necesarios para resolver el problema de las Torres de Hanoi con `n` discos.

```

class Hanoi {
    def movsTorresHanoi(n : Int) : BigInt = {
        if(n <= 0)
            throw new IllegalArgumentException("el numero de discos debe de ser mayor a cero")
        else if (n == 1)
            1
        else 2 * movsTorresHanoi(n-1) + 1
    }
}

```

Descripción del código

Si $n \leq 0$, la operación no será posible.

Si n es igual obtendremos que la cantidad de movimientos será 1. Actúa como caso base

Si $n > 1$ se hace un llamado recursivo con $n-1$, a su vez, duplicando el valor y sumándole uno

Ejemplo de Ejecución

Para 3 discos:

Llamada inicial: `movsTorresHanoi(3):`

$2 * \text{movsTorresHanoi}(2) + 1$

Segunda llamada: `movsTorresHanoi(2):`

$2 * \text{movsTorresHanoi}(1) + 1$

Tercera llamada: `movsTorresHanoi(1):`

Obtenemos 1

$2 * \text{movsTorresHanoi}(1) + 1 \rightarrow 2(1) + 1 = 3$

$2 * \text{movsTorresHanoi}(2) + 1 \rightarrow 2(3) + 1 = 7$

Análisis 2: torresHanoi

Definición del problema

La función torresHanoi resuelve el problema de mover n discos desde la torre t1 hacia la torre t3, utilizando la torre t2 como torre auxiliar.

```
def torresHanoi( n : Int , t1 : Int , t2 : Int , t3 : Int) : List[(Int , Int)] = {  
  if(n <= 0)  
    Nil  
  else if(n == 1)  
    List((t1,t3))  
  else {  
    val mov1 = torresHanoi(n-1, t1, t3, t2)  
    val movn = List((t1,t3))  
    val mov2 = torresHanoi(n-1,t2, t1, t3)  
    mov1 ++ movn ++ mov2  
  }  
}
```

Descripción del Proceso

1. Si n es menor o igual a 0, la función retorna una lista vacía, ya que no hay discos que mover.
2. Si n es 1, se retorna una lista con un solo movimiento: mover el disco de la torre t1 a la torre t3.
3. Para n>1 , el proceso recursivo genera tres listas de movimientos:

mov1: los movimientos para trasladar n-1 discos de t1 a t2 usando t3 como auxiliar.

movn: el movimiento del disco más grande de t1 a t3.

mov2: los movimientos para trasladar n-1 discos de t2 a t3 usando t1 como auxiliar.

Ejemplo de Ejecución

Para n = 3, t1 = 1, t2 = 2 y t3 = 3:

1. **Llamada inicial:** torresHanoi(3, 1, 2, 3)
 - Llama a torresHanoi(2, 1, 3, 2) para mover 2 discos de la torre 1 a la torre 2.
2. **Segunda llamada:** torresHanoi(2, 1, 3, 2)
 - Llama a torresHanoi(1, 1, 2, 3) para mover un disco de la torre 1 a la torre 3.
3. **Tercera llamada:** torresHanoi(1, 1, 2, 3)

- Retorna List((1, 3)).

4. **Cuarta llamada:** Después de mover los 2 discos, llama a torresHanoi(1, 2, 1, 3) para mover el disco más grande de la torre 2 a la torre 3.

El resultado final es:

[(1, 3), (1, 2), (3, 2), (1, 3), (2, 1), (2, 3), (1, 3)]

Informe de Corrección

Sea $f: \text{List}[N] \rightarrow N$ la función que calcula el máximo de una lista de enteros no vacía.

Demostremos que $\forall l \in \text{List}[N]: \text{maxLin}(l) = f(l)$

Caso base: $l = \text{List}(a_1)$

$\text{maxLin}(\text{List}(a_1)) \rightarrow \text{if } \text{List}(a_1).\text{tail}.\text{isEmpty} \text{ then } \text{List}(a_1).\text{head} \text{ else } \dots \rightarrow a_1$

por otro lado, $f(\text{List}(a_1)) = a_1$. Entonces, $\text{maxLin}(\text{List}(a_1)) = f(\text{List}(a_1))$

Caso de inducción $l = \text{List}(a_1, a_2, \dots, a_{k+1})$

Supongamos que $\text{maxLin}(\text{List}(a_1, a_2, \dots, a_k)) = f(\text{List}(a_1, a_2, \dots, a_k))$

Queremos demostrar que $\text{maxLin}(\text{List}(a_1, a_2, \dots, a_k)) = f(\text{List}(a_1, a_2, \dots, a_k + 1))$

$\text{maxLin}(\text{List}(a_1, a_2, \dots, a_{k+1})) \rightarrow \text{math.max}(\text{maxLin}(\text{List}(a_2, \dots, a_{k+1})), a_1)$

Usando la hipótesis de inducción

$\text{math.max}(\text{maxLin}(\text{List}(a_2, \dots, a_{k+1})), a_1) = \text{math.max}(f(\text{List}(a_2, \dots, a_{k+1})), a_1) = f(\text{List}(a_1, a_2, \dots, a_{k+1}))$

Por lo tanto, $\text{maxLin}(\text{List}(a_1, a_2, \dots, a_{k+1})) = f(\text{List}(a_1, a_2, \dots, a_{k+1}))$.

Concluimos por inducción que $\forall l \in \text{List}[N]: \text{maxLin}(l) = f(l)$

Análisis del Programa 2: maxIt

Sea $f: \text{List}[N] \rightarrow N$

Este programa implementa el siguiente proceso iterativo:

Un estado $s = (\text{max}, l)$ donde $l = \text{List}(a_i, a_{i+1}, \dots, a_k)$ es una cola de L

El estado inicial $s_0 = (L.\text{head}, L.\text{tail}) = (a_1, \text{List}(a_2, \dots, a_k))$

El estado $s = (\text{max}, l)$ es final si l es vacía

La invariante de ciclo es $\text{Inv}(\text{max}, l) \equiv l = \text{List}(a_i, a_{i+1}, \dots, a_k) \wedge \text{max} = f(\text{List}(a_1, a_2, \dots, a_{i-1}))$.

La transformación de estado es transformar $(\text{max}, l) = (\text{nmax}, l.\text{tail})$ donde $\text{nmax} = \text{math.nax}(\text{max}, l.\text{head})$

Demostración de los puntos

1. $\text{Inv}(s_0)$ El estado inicial cumple la condición invariante.
 $s_0 = (a_1, \text{List}(a_2, \dots, a_k)) \rightarrow a_1 = f(\text{List}(a_1))$.
2. $(\text{Si } \neq sf \wedge \text{Inv}(s_i)) \rightarrow \text{Inv}(\text{transformar}(s_i))$:
Si $\text{Inv}(s_i)$ es verdadera y $s_i \neq$, entonces $\text{Inv}(\text{transformar}(s_i))$ es verdadera.
 $(\text{max}, l) \rightarrow (\text{nmax}, l.\text{tail})$, donde $\text{nmax} = \text{math.max}(\text{max}, l.\text{head})$ y nmax sigue siendo el máximo de $\text{List}(a_1, \dots, a_i)$.
3. $\text{Inv}(sf) \rightarrow \text{respuestas}(sf) = f(a)$:
En el estado final $(\text{max}, \text{List}())$, $\text{max} = f(\text{List}(a_1, \dots, a_k))$
4. En cada paso, la lista l se reduce, acercándose a ser vacía. Después de k iteraciones, $l = \text{List}()$.

Esto implica que $\text{maxIt}(l) = \text{maxIt}(\text{iter}(L.\text{head}, L.\text{tail})) = f(L)$

Concluimos que $\text{maxIt}(l) = f(l)$

Análisis 2: torresHanoi

La función $f(n)$ se define como:

$f(1) = 1$ (un solo disco requiere un movimiento)

Para $n > 1$, $f(n) = 2 * f(n-1) + 1$

Demostrar que:

$\forall n \in \mathbb{N}^+ : \text{Pf}(n) == f(n)$

1. Caso base $n = 1$

Para $n = 1$, la función devuelve q , que es el número mínimo de movimientos necesarios para trasladar un disco. Así mismo, $f(1) = 1$ por definición obtenemos, $\text{Pf}(1) == f(1)$

2. $n = k + 1$ con $k \geq 1$

supongamos que $\text{Pf}(k) == f(k)$. Demostrar que $f(k+1) == \text{Pf}(k+1)$

Según la función recursiva, tenemos $\text{Pf}(k+1) = 2 * \text{Pf}(k) + 1$

Sustituimos $\text{Pf}(k)$ por $f(k)$, lo que nos da:

$\text{Pf}(k+1) = 2 * f(k) + 1 = f(k+1)$

Se puede concluir que $\text{Pf}(k+1) == f(k+1)$

$\text{Pf}(n) == f(n)$ para todo $n \in \mathbb{N}^+$

Análisis del Programa 2: Secuencia de movimientos

Caso base:

Para $n=1$, la función devuelve una lista con un único movimiento, que es mover el disco de la torre t_1 a la torre t_3 . Este es el resultado esperado, ya que solo hay un disco que mover.

Paso inductivo:

Asumimos que la función es correcta para $n=k$, es decir, genera la secuencia correcta de movimientos para mover k discos de t_1 a t_3 utilizando t_2 como torre auxiliar. Ahora probamos que la función es correcta para $n=k+1$.

Para $n=k+1$, la función divide el problema en tres partes:

1. Mueve k discos de t_1 a t_2 utilizando t_3 como auxiliar. Esto está cubierto por la hipótesis inductiva.
2. Mueve el disco más grande de t_1 a t_3 . Este movimiento es correcto.
3. Mueve k discos de t_2 a t_3 utilizando t_1 como auxiliar. Esto también está cubierto por la hipótesis inductiva.

Dado que cada parte del proceso es correcta por la hipótesis inductiva, la función es correcta para $n=k+1$. Por inducción, la función es correcta para todos los $n \geq 1$.

Conclusiones

Uso eficiente de la recursividad en problemas clásicos: El código implementa de forma efectiva soluciones recursivas para problemas clásicos como las Torres de Hanoi y la búsqueda del máximo en una lista.

La función `maxIt` utiliza recursividad de cola, lo que es una mejora sobre la versión recursión lineal (`maxLin`)