

Informe Taller 1

Jaime Andrés Noreña 2359523

Dilan Muricio lemos 2359416

Juam Jose Restrepo 2359517

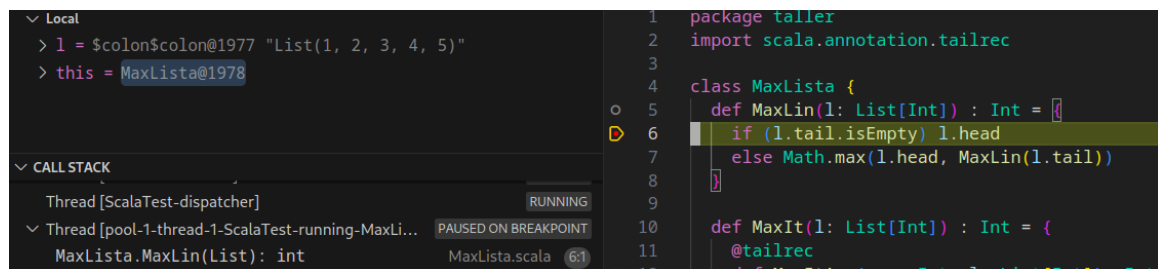
September 2024

1 Informe de Procesos

Para cada función implementada en nuestro código entregado se le realizará un ejemplo sencillo para mostrar su comportamiento a lo largo de la ejecución del código y comprobar su correcto funcionamiento.

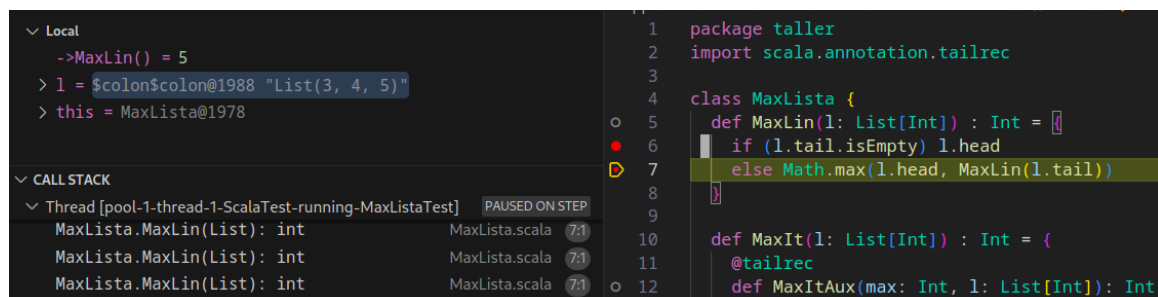
1.1 Funcion de Maxlin

para la función `MaxLin(List(1, 2, 3, 4, 5)) = 5` al ejecutar el test “devolver 6 en `List(1, 2, 3, 4, 5)`” el primer llamado a la función de ve así:



```
1 package taller
2 import scala.annotation.tailrec
3
4 class MaxLista {
5   def MaxLin(l: List[Int]) : Int = {
6     if (l.tail.isEmpty) l.head
7     else Math.max(l.head, MaxLin(l.tail))
8   }
9
10  def MaxIt(l: List[Int]) : Int = {
11    @tailrec
12    def MaxItAux(max: Int, l: List[Int]): Int = {
13      if (l.isEmpty) max
14      else MaxItAux(Math.max(max, l.head), l.tail)
15    }
16  }
17 }
```

habiendo solo un llamado a la función `MaxLin`, a continuación se harán llamadas recursivas hasta que `l.tail` este vacío lo que significa que solo queda un valor en la lista `l` que al retornar a cada llamado anterior se comparará al `l.head` correspondiente finalmente retornando datos más grande de la lista. A continuación fotos del proceso



```
1 package taller
2 import scala.annotation.tailrec
3
4 class MaxLista {
5   def MaxLin(l: List[Int]) : Int = {
6     if (l.tail.isEmpty) l.head
7     else Math.max(l.head, MaxLin(l.tail))
8   }
9
10  def MaxIt(l: List[Int]) : Int = {
11    @tailrec
12    def MaxItAux(max: Int, l: List[Int]): Int = {
13      if (l.isEmpty) max
14      else MaxItAux(Math.max(max, l.head), l.tail)
15    }
16  }
17 }
```

```

1 package taller
2 import scala.annotation.tailrec
3
4 class MaxLista {
5   def MaxLin(l: List[Int]) : Int = {
6     if (l.tail.isEmpty) l.head
7     else Math.max(l.head, MaxLin(l.tail))
8   }
9
10  def MaxIt(l: List[Int]) : Int = {
11    @tailrec
12    def MaxItAux(max: Int, l: List[Int]): Int = {
13      if (l.isEmpty) max
14      else MaxItAux(Math.max(max, l.head), l.tail)
15    }
16  }

```

```

1 package taller
2 import scala.annotation.tailrec
3
4 class MaxLista {
5   def MaxLin(l: List[Int]) : Int = {
6     if (l.tail.isEmpty) l.head
7     else Math.max(l.head, MaxLin(l.tail))
8   }
9
10  def MaxIt(l: List[Int]) : Int = {
11    @tailrec
12    def MaxItAux(max: Int, l: List[Int]): Int = {
13      if (l.isEmpty) max
14      else MaxItAux(Math.max(max, l.head), l.tail)
15    }
16  }

```

En este punto la función retornara l.head ya que l.tail está vacío y el call stack empezará a reducirse ya que todas las llamadas empezaran a retornar.

```

1 package taller
2 import scala.annotation.tailrec
3
4 class MaxLista {
5   def MaxLin(l: List[Int]) : Int = {
6     if (l.tail.isEmpty) l.head
7     else Math.max(l.head, MaxLin(l.tail))
8   }
9
10  def MaxIt(l: List[Int]) : Int = {
11    @tailrec
12    def MaxItAux(max: Int, l: List[Int]): Int = {
13      if (l.isEmpty) max
14      else MaxItAux(Math.max(max, l.head), l.tail)
15    }
16  }

```

una llamada menos

```

1 package taller
2 import scala.annotation.tailrec
3
4 class MaxLista {
5   def MaxLin(l: List[Int]) : Int = {
6     if (l.tail.isEmpty) l.head
7     else Math.max(l.head, MaxLin(l.tail))
8   }
9
10  def MaxIt(l: List[Int]) : Int = {
11    @tailrec
12    def MaxItAux(max: Int, l: List[Int]): Int = {
13      if (l.isEmpty) max
14      else MaxItAux(Math.max(max, l.head), l.tail)
15    }
16  }

```

The first screenshot shows the local variables and call stack for the initial call to `MaxLin`. The list is `List(2, 3, 4, 5)` and the function is paused on step 7 of the `MaxLin` method.

The second screenshot shows the recursive call to `MaxLin` with the list `List(1, 2, 3, 4, 5)`. The function is paused on step 7 of the `MaxLin` method.

The third screenshot shows the test assertion passing. The function is paused on step 12 of the `MaxListaTest` class.

Al final al llegar al retorno del primer llamado vemos como se retornó el valor esperado: 5

1.2 Función de MaxIt

La función `MaxIt(List(3, 14, 5))` en si no es recursiva así que nos centraremos en `MaxItAux` que es la que se encarga de la recursión en sí.

`MaxItAux(0, List(3, 14, 5))`

The screenshot shows the local variables and call stack for the `MaxIt` function. The list is `List(3, 14, 5)` and the function is paused on step 13 of the `MaxItAux` method.

Podemos ver en la pila de llamados que solo hay dos llamados relacionados a esta función `MaxIt` y `MaxItAux`. Como la lista no está vacía el flujo del código se irá a la línea 14 donde esta el `else` que

llamara otra vez a la función con el valor del atributo max siendo el mayor entre l.head y max y l.tail siendo el nuevo l. Dicho esto procedemos a ver la próxima llamada.

```

1 package taller
4 class MaxLista {
5   def MaxLin(l: List[Int]) : Int = {
6     if (l.tail.isEmpty) l.head
7     else Math.max(l.head, MaxLin(l.tail))
8   }
9
10  def MaxIt(l: List[Int]) : Int = {
11    @tailrec
12    def MaxItAux(max: Int, l: List[Int]): Int = {
13      if (l.isEmpty) max
14      else MaxItAux(Math.max(max, l.head), l.tail)
15    }
16  }

```

Aquí podemos ver como los atributos cambiaron, siendo ahora l = List(14, 5) y max = 3, pero si miramos la pila llamados esta misma no aumentó, esto porque al estar utilizando la recursión de cola esta función se puede representar con un ciclo iterativo ya que toda la información que la función necesita para retornar siempre está en sus argumentos permitiendo simplemente reemplazar estos mismos valores una y otra vez hasta que se pueda retornar el resultado de la función sin hacer llamados adicionales. A continuación el resto de los pasos.

```

1 package taller
4 class MaxLista {
5   def MaxLin(l: List[Int]) : Int = {
6     if (l.tail.isEmpty) l.head
7     else Math.max(l.head, MaxLin(l.tail))
8   }
9
10  def MaxIt(l: List[Int]) : Int = {
11    @tailrec
12    def MaxItAux(max: Int, l: List[Int]): Int = {
13      if (l.isEmpty) max
14      else MaxItAux(Math.max(max, l.head), l.tail)
15    }
16  }

```

```

1 package taller
4 class MaxLista {
5   def MaxLin(l: List[Int]) : Int = {
6     if (l.tail.isEmpty) l.head
7     else Math.max(l.head, MaxLin(l.tail))
8   }
9
10  def MaxIt(l: List[Int]) : Int = {
11    @tailrec
12    def MaxItAux(max: Int, l: List[Int]): Int = {
13      if (l.isEmpty) max
14      else MaxItAux(Math.max(max, l.head), l.tail)
15    }
16  }

```

Aquí se llegó al punto del caso base donde vamos a ver que se va a retornar 14 a la función MaxIt que va a retornar al test.

```

1 package taller
4 class MaxLista {
5   def MaxLin(l: List[Int]) : Int = {
6     if (l.tail.isEmpty) l.head
7     else Math.max(l.head, MaxLin(l.tail))
8   }
9
10  def MaxIt(l: List[Int]) : Int = {
11    @tailrec
12    def MaxItAux(max: Int, l: List[Int]): Int = {
13      if (l.isEmpty) max
14      else MaxItAux(Math.max(max, l.head), l.tail)

```

1.3 Función movsTorresHanoi

Veamos el proceso de $\text{movsTorresHanoi}(3) = 7$

```

2 class Hanoi() {
3   def movsTorresHanoi (n : Int) : BigInt = {
4     if (n == 0) 0
5     else 2 * movsTorresHanoi(n - 1) + 1
6   }
7
8   def TorresHanoi (n : Int, t1: Int, t2: Int, t3: Int) : List[Int] = {
9     if (n == 0) List()
10    if (n == 1) List((t1, t3))
11    else ((TorresHanoi(n-1, t1, t3, t2) ++ (t1, t3)) :: List())
12  }

```

para este punto se usó una forma recursiva de la fórmula de la complejidad del ejercicio de las torres de hanoi que es $2^n - 1$ donde el 2^n se calcula recursivamente. Ahora veremos todas las llamadas recursivas hasta llegar al fondo de la recesión

```

1 package taller
2
3 class Hanoi() {
4   def movsTorresHanoi (n : Int) : BigInt = {
5     if (n == 0) 0
6     else 2 * movsTorresHanoi(n - 1) + 1
7   }
8
9   def TorresHanoi (n : Int, t1: Int, t2: Int, t3: Int) : List[Int] = {
10    if (n == 0) List()

```

```

1 package taller
2
3 class Hanoi() {
4   def movsTorresHanoi (n : Int) : BigInt = {
5     if (n == 0) 0
6     else 2 * movsTorresHanoi(n - 1) + 1
7   }
8
9   def TorresHanoi (n : Int, t1: Int, t2: Int, t3: Int) : List[Int] = {
10    if (n == 0) List()
11    if (n == 1) List((t1, t3))
12    else ((TorresHanoi(n-1, t1, t3, t2) ++ (t1, t3)) :: List())
13  }

```

aquí vemos como se llega a una llamada con $n = 0$ que retornara 1 a la llamada anterior

```

1 package taller
2
3 class Hanoi() {
4     def movsTorresHanoi (n : Int) : BigInt = {
5         if (n == 0) 0
6         else 2 * movsTorresHanoi(n - 1) + 1
7     }
8
9     def TorresHanoi (n : Int, t1: Int, t2: Int, t3: Int) : List[(Int, Int, Int)] = {
10         if (n == 0) List()
11         if (n == 1) List((t1, t3))
12         else ((TorresHanoi(n-1, t1, t3, t2) ++ (t1, t3)) :: TorresHanoi(n-1, t2, t1, t3))
13     }
14 }

```

esta llamada retornara $(2 * 0) + 1 = 1$

```

1 package taller
2
3 class Hanoi() {
4     def movsTorresHanoi (n : Int) : BigInt = {
5         if (n == 0) 0
6         else 2 * movsTorresHanoi(n - 1) + 1
7     }
8
9     def TorresHanoi (n : Int, t1: Int, t2: Int, t3: Int) : List[(Int, Int, Int)] = {
10         if (n == 0) List()
11         if (n == 1) List((t1, t3))
12         else ((TorresHanoi(n-1, t1, t3, t2) ++ (t1, t3)) :: TorresHanoi(n-1, t2, t1, t3))
13     }
14 }

```

En este punto retornara $(2 * 1) + 1 = 3$

```

1 package taller
2
3 class Hanoi() {
4     def movsTorresHanoi (n : Int) : BigInt = {
5         if (n == 0) 0
6         else 2 * movsTorresHanoi(n - 1) + 1
7     }
8
9     def TorresHanoi (n : Int, t1: Int, t2: Int, t3: Int) : List[(Int, Int, Int)] = {
10         if (n == 0) List()
11         if (n == 1) List((t1, t3))
12         else ((TorresHanoi(n-1, t1, t3, t2) ++ (t1, t3)) :: TorresHanoi(n-1, t2, t1, t3))
13     }
14 }

```

Finalmente llegamos a la primera llamada donde la función retornará $(2 * 3) + 1 = 7$, el cual es el valor final y correcto de la función para $n = 3$.

1.4 Función TorresHanoi

Veamos el preces de la función $TorresHanoi(2, 1, 2, 3) = List((1,2), (1,3), (2, 3))$

```

1 package taller
2
3 class Hanoi() {
4     def movsTorresHanoi (n : Int) : BigInt = {
5         if (n == 0) 0
6         else 2 * movsTorresHanoi(n - 1) + 1
7     }
8
9     def TorresHanoi (n : Int, t1: Int, t2: Int, t3: Int) : List[(Int, Int, Int)] = {
10         if (n == 0) List()
11         if (n == 1) List((t1, t3))
12         else ((TorresHanoi(n-1, t1, t3, t2) ++ (t1, t3)) :: TorresHanoi(n-1, t2, t1, t3))
13     }
14 }

```

primera llamada, esta función realiza dos llamadas por vez, se evaluará primero la de la derecha.

```

VARIABLES
  Local
    n = 1
    t1 = 1
    t2 = 3
    t3 = 2
    > this = Hanoi@2043

CALL STACK
  Thread [pool-1-thread-1-ScalaTest-run...] PAUSED ON BREAKPOINT
    Hanoi.TorresHanoi(int,int,int,int): List Han...
    Hanoi.TorresHanoi(int,int,int,int): List Han...

app > src > main > scala > taller > Hanoi.scala > {} taller > Hanoi > TorresHanoi
1 package taller
3 class Hanoi() {
9
10 def TorresHanoi (n : Int, t1: Int, t2: Int, t3: Int) : List[(Int, Int)] = {
11   if (n == 0) List()
12   if (n == 1) List((t1, t3))
13   else ((TorresHanoi(n-1, t1, t3, t2) :+ (t1, t3)) ::: TorresHanoi(n-1, t2, t1, t3))
14 }
15

```

Vemos como se ve reflejada la nueva llamada en el stack y los cambios en los parámetros. Esta función al ser $n = 1$ retornara $List((t1, t3))$ que en este caso equivale a $List((1, 2))$.

```

VARIABLES
  Local
    n = 2
    t1 = 1
    t2 = 2
    t3 = 3
    > this = Hanoi@2043

CALL STACK
  Thread [ScalaTest-dispatcher] RUNNING
  Thread [pool-1-thread-1-ScalaTest-running-HanoiTest] PAUSED ON STEP
    Hanoi.TorresHanoi(int,int,int,int): List Han...
    Hanoi.TorresHanoi(int,int,int,int): List Han...

app > src > main > scala > taller > Hanoi.scala > {} taller > Hanoi > TorresHanoi
1 package taller
3 class Hanoi() {
9
10 def TorresHanoi (n : Int, t1: Int, t2: Int, t3: Int) : List[(Int, Int)] = {
11   if (n == 0) List()
12   if (n == 1) List((t1, t3))
13   else ((TorresHanoi(n-1, t1, t3, t2) :+ (t1, t3)) ::: TorresHanoi(n-1, t2, t1, t3))
14 }
15

```

la llamada anterior sale del stack y volvemos a la primera donde ahora entraremos en la llamada de la izquierda

```

VARIABLES
  Local
    n = 1
    t1 = 2
    t2 = 1
    t3 = 3
    > this = Hanoi@2043

CALL STACK
  Thread [pool-1-thread-1-ScalaTest-run...] PAUSED ON BREAKPOINT
    Hanoi.TorresHanoi(int,int,int,int): List Han...
    Hanoi.TorresHanoi(int,int,int,int): List Han...

app > src > main > scala > taller > Hanoi.scala > {} taller > Hanoi > TorresHanoi
1 package taller
3 class Hanoi() {
9
10 def TorresHanoi (n : Int, t1: Int, t2: Int, t3: Int) : List[(Int, Int)] = {
11   if (n == 0) List()
12   if (n == 1) List((t1, t3))
13   else ((TorresHanoi(n-1, t1, t3, t2) :+ (t1, t3)) ::: TorresHanoi(n-1, t2, t1, t3))
14 }
15

```

ya que $n = 1$ esta llamada retornara $List((2, 3))$

```

VARIABLES
  Local
    n = 2
    t1 = 1
    t2 = 2
    t3 = 3
    > this = Hanoi@2043

CALL STACK
  Thread [pool-1-thread-1-ScalaTest-run...] PAUSED ON BREAKPOINT
    Hanoi.TorresHanoi(int,int,int,int): List Han...
    Hanoi.TorresHanoi(int,int,int,int): List Han...

app > src > main > scala > taller > Hanoi.scala > {} taller > Hanoi > TorresHanoi
1 package taller
3 class Hanoi() {
9
10 def TorresHanoi (n : Int, t1: Int, t2: Int, t3: Int) : List[(Int, Int)] = {
11   if (n == 0) List()
12   if (n == 1) List((t1, t3))
13   else ((TorresHanoi(n-1, t1, t3, t2) :+ (t1, t3)) ::: TorresHanoi(n-1, t2, t1, t3))
14 }
15

```

habiendo hecho las dos llamadas anteriores se puede retornar el resultado el cual sería $List((1, 2)) : + (1, 3) ::: List((2, 3)) = List((1, 2), (1, 3), (2, 3))$ el cual es el resultado esperado.

2 Informe de Corrección

2.1 MaxLin

Sea $f : List[N] \rightarrow N$ la función que calcula el máximo de una lista de enteros positivos, no vacía. Y sea P_f el siguiente programa en Scala:

```
def maxLin ( l: List[Int] ): Int = {
  if ( l.tail.isEmpty ) l.head
  else math.max( maxLin( l.tail ), l.head )
}
```

Demostraremos que: $\forall n \in \mathbf{N} \setminus \{0\} : P_f(\text{List}(a_1, a_2, \dots, a_n)) == f(\text{List}(a_1, a_2, \dots, a_n))$
 caso base $n = 1$

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

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

caso de inducción: $n = k + 1, k \geq 1$

Se debe demostrar:

$P_f(\text{List}(b_1, b_2, \dots, b_k)) == f(\text{List}(b_1, b_2, \dots, b_k)) \rightarrow P_f(\text{List}(a_1, a_2, \dots, a_{k+1})) == f(\text{List}(a_1, a_2, \dots, a_{k+1}))$

Empecemos por calcular que devuelve P_f usando el modelo de sustitucion:

$P_f(L) \rightarrow \text{if } L.\text{tail.isEmpty then } L.\text{head else } \text{math.max}(P_f(L.\text{tail}), L.\text{head})$
 $\rightarrow \text{math.max}(P_f(\text{List}(a_2, \dots, a_{k+1})), a_1)$

Hay dos posibilidades:

- Si $\text{math.max}(b, a_1) = b$, entonces $b \geq a_1$ y $b == f(\text{List}(a_1, a_2, \dots, a_{k+1}))$
- Si $\text{math.max}(b, a_1) = a_1$, entonces $a_1 \geq b$ y $a_1 == f(\text{List}(a_1, a_2, \dots, a_{k+1}))$

Por lo tanto, $P_f(L) == f(L)$.

Concluimos por induccion que: $\forall n \in \mathbf{N} \setminus \{0\} : P_f(\text{List}(a_1, a_2, \dots, a_n)) == f(\text{List}(a_1, a_2, \dots, a_n))$

2.2 MaxIt

Sea $f : \text{List}[\mathbf{N}] \rightarrow \mathbf{N}$ la funcion que calcula el maximo de una lista de enteros positivos. Y sea P_f el siguiente programa en Scala:

```
def maxIt(l: List[Int]): Int = {
  def maxAux(max: Int, l: List[Int]): Int = {
    if (l.isEmpty) max
    else maxAux(math.max(max, l.head), l.tail)
  }
  maxAux(l.head, l.tail)
}
```

Este programa implementa el siguiente proceso iterativo:

- Un estado $s = (max, l)$ donde $l = \text{List}(a_i, a_{i+1}, \dots, a_k)$ es una cola de L
- El estado inicial es $s_0 = (L.\text{head}, L.\text{tail}) = (a_1, \text{List}(a_2, \dots, a_k))$
- $s = (max, l)$ es final si l es vacia
- $\text{Inv}(max, l) \equiv l = \text{List}(a_i, a_{i+1}, \dots, a_k) \ \& \ max = f(\text{List}(a_1, a_2, \dots, a_i - 1))$

- $\text{transformar}((max, l)) = (nmax, l.tail)$ donde $nmax = max$ si $max \geq l.head$ y $nmax = l.head$ sino

Demostracion:

1. $\text{Inv}(s_o)$: el estado inicial cumple la condicion invariante.

$$s_0 = (a_1, \text{List}(a_2, \dots, a_k)) \implies a_1 = f(\text{List}(a_1))$$

2. $(s_i \neq s_f \wedge \text{Inv}(s_i)) \rightarrow \text{Inv}(\text{transformar}(s_i))$

$$\neg l.isEmpty \wedge = l.\text{List}(a_i, a_{i+1}, \dots, a_k) \wedge max = f(\text{List}(a_1, a_2, \dots, a_{i1}))$$

$$\rightarrow l.tail = \text{List}(a_{i+1}, \dots, a_k) \wedge nmax = f(\text{List}(a_1, \dots, a_i))$$

3. $\text{Inv}(s_f) \rightarrow \text{respuesta}(s_f) == f(a)$

$$\text{Inv}((max, \text{List}())) \rightarrow max = f(\text{List}(a_1, \dots, a_k))$$

4. En cada paso, la lista l se reduce, acercandose a ser vacia. Despues de k iteraciones, $l = \text{List}()$.

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

2.3 MovsTorresHanoi

Sea $f : \mathbf{N} \rightarrow \mathbf{N}$ la función que relaciona el numero de discos de una torre de hanoi con el numero los pasos que toma para llevarlos hacia la tercera varilla siendo esta $f(a) = 2^a - 1$, P_f un programa hecho en scala que calcula esta función y se define así:

```
def movsTorresHanoi (n : Int ) : BigInt = {
  if (n == 0) 0
  else 2 * movsTorresHanoi(n - 1) + 1
}
```

y $a \in \mathbf{N}$ se procederá a demostrar que:

$$\forall a \in \mathbf{N} : P_f(a) == 2^a - 1$$

- caso base $n = 0$

$$P_f(n) \rightarrow \text{if } (n == 0) 0 \text{ else } 2 * P_f(-1) + 1 \rightarrow 0$$

por otro lado $f(0) = 0$

- caso de inducción $n = k + 1, k \geq 0$

para demostrar demostrar que $P_f(k) == f(k) \rightarrow P_f(k + 1) == f(k + 1)$ tenemos:

$$P_f(k + 1) \rightarrow \text{if } (k + 1 == 0) 0 \text{ else } 2 * P_f(k + 1 - 1) + 1$$

usando la hipótesis de inducción

$$\rightarrow 2 * (2^k - 1) + 1 = 2^{(k+1)} - 1$$

$$\rightarrow 2^{k+1} - 1 = 2^{k+1} - 1$$

por lo tanto se concluye $P_f(k + 1) == f(k + 1)$

entonces queda demostrado por inducción que $\forall a \in \mathbf{N} : P_f(a) == 2^a - 1$

2.4 TorresHanoi

Sea $f : \mathbf{N}, \mathbf{N}, \mathbf{N}, \mathbf{N} \rightarrow \text{List}((\mathbf{N}, \mathbf{N}))$ la función que relaciona el numero de discos de una torre de hanoi, y los 3 nombres de cada varilla representados con tres números, con los pasos que toma para llevarlos hacia la tercera varilla representados en tuplas de estos valores y $k \in \{a, b, c\}$,

siendo $f: f(n, a, b, c) = \text{List}((k, k)_1, (k, k)_2, \dots, (k, k)_{2^n-1})$.

Siendo P_f un programa hecho en scala que calcula esta función y se define así:

```
def TorresHanoi (n : Int, t1: Int, t2: Int, t3: Int) : List[(Int, Int)] = {
  if (n == 0) List()
  if (n == 1) List((t1, t3))
  else ((TorresHanoi(n-1, t1, t3, t2) :+ (t1, t3)) :: TorresHanoi(n-1, t2, t1, t3))
}
```

se procede a demostrar que:

$$\forall n, \forall a, \forall b, \forall c \in \mathbf{N}, k \in \{a, b, c\} : P_f(n, a, b, c) == (k, k)_1, (k, k)_2, \dots, (k, k)_{2^n-1}$$

- caso base $n = 1$

$$P_f(1, a, b, c) \rightarrow \text{if } (n == 1) \text{ List}((a, c)) \text{ else } P_f(0, a, c, b) + \text{List}((a, c)) + P_f(0, b, a, c) \rightarrow \text{List}((a, c))$$

$$\text{por otro lado } f(1, a, b, c) = \text{List}((a, c))$$

- caso de inducción $t = n + 1, t \geq 0$

para demostrar demostrar que $P_f(t, a, b, c) == f(t, a, b, c) \rightarrow P_f(t+1, a, b, c) == f(t+1, a, b, c)$ tenemos:

$$P_f(t+1, a, b, c) \rightarrow \text{if } (t+1 == 1) \text{ List}((a, c)) \text{ else } P_f(t+1-1, a, c, b) + \text{List}((a, c)) + P_f(t+1-1, b, a, c)$$

usando la hipótesis de inducción

$$\rightarrow P_f(t, a, c, b) + \text{List}((a, c)) + P_f(t, b, a, c) = f(1, a, c, b) + \text{List}((a, c)) + f(1, b, a, c)$$

$$\rightarrow P_f(1, a, c, b) + \text{List}((a, c)) + P_f(1, b, a, c) = \text{List}((a, b)) + \text{List}((a, c)) + \text{List}((b, c))$$

$$\rightarrow P_f(1, a, c, b) + \text{List}((a, c)) + P_f(1, b, a, c) = \text{List}((a, b), (a, c), (b, c))$$

$$\rightarrow \text{List}((a, b)) + \text{List}((a, c)) + \text{List}((b, c)) = \text{List}((a, b), (a, c), (b, c))$$

$$\text{List}((a, b), (a, c), (b, c)) = \text{List}((a, b), (a, c), (b, c))$$

por lo tanto se concluye $P_f(k+1, a, b, c) == f(k+1, a, b, c)$

entonces queda demostrado por inducción que $\forall n, \forall a, \forall b, \forall c \in \mathbf{N}, k \in \{a, b, c\} : P_f(n, a, b, c) == (k, k)_1, (k, k)_2, \dots, (k, k)_{2^n-1}$

3 Conclusiones

Durante este proyecto, nos vimos obligados a adoptar una perspectiva más profunda al analizar problemas de manera recursiva para encontrar las soluciones más óptimas. La recursión es una técnica fundamental en la programación, y su correcta aplicación nos permitió abordar eficazmente problemas como el manejo de listas y la resolución de las Torres de Hanoi. Mediante la recursión,

logramos descomponer problemas complejos en subproblemas más manejables, facilitando su solución de una manera más rápida y sencilla.

Además, exploramos las diferencias y similitudes entre la recursión lineal y la recursión de cola. Aunque ambas técnicas comparten el objetivo de resolver problemas recursivos, presentan diferencias significativas en su funcionamiento y eficiencia. La recursión lineal es intuitiva y directa; en cada llamada recursiva, procesa una parte del problema antes de realizar la siguiente llamada. Sin embargo, este enfoque puede llevar a un mayor consumo de memoria, ya que todas las llamadas recursivas permanecen en la pila hasta completar la función. En contraste, la recursión de cola reutiliza el espacio de pila de las llamadas anteriores, dado que la llamada recursiva es la última operación que se ejecuta. Esto permite evitar desbordamientos de pila y reduce significativamente el uso de memoria.

El rendimiento es un factor crucial que hemos tenido que considerar. Aunque la recursión lineal es más sencilla de entender e implementar, puede resultar ineficiente para problemas a gran escala debido a su alto consumo de recursos. La recursión de cola, si se aplica correctamente, mejora la eficiencia del código al optimizar el uso de la pila, haciéndola más adecuada para algoritmos que requieren un gran número de llamadas recursivas. Esta optimización se traduce en un mejor rendimiento y en la posibilidad de resolver problemas más complejos sin comprometer la estabilidad del programa.