Universidad del Valle

Facultad de ingeniería

Ingeniería en sistemas

Cristian David Pacheco Torres
2227437
Juan Sebastian Molina Cuellar
2224491

Septiembre 2023

Taller 2

Abstract

Your abstract goes here functional programming

Contents

Funciones de alto orden		
Función	Forma de alto orden	Expresión donde aparece
$Chip$	Retorno	Retorno de funciones crearChipunario, crearChipBinario, half_adder, full_adder, adder
$(x : Int) \Rightarrow (x - 1)$	Lambda como argumento	$crearChipUnario((x : Int) \Rightarrow (x - 1)) : Chip$
$(x : Int, y : Int) => (x * y)$	Lambda como argumento	$crearChipBinario((x : Int, y : Int) => (x * y)) : Chip$
$(x : Int, y : Int) => (x + y) - (x * y)$	Lambda como argumento	$crearChipBinario((x : Int, y : Int) => (x + y) - (x * y)) : Chip$
$half\_adder$	Variable la cual se asigna una función de retorno	val $half\_adder = (operands : List[Int]) =>$ { ... }
$full\_adder$	Variable la cual se asigna una función de retorno	val $full\_adder = (operands : List[Int]) =>$ { ... }
$adder$	Variable la cual se asigna una función de retorno	val $adder = (operands : List[Int]) =>$ { ... }

Table 1: Funciones de alto orden realizadas en la implementación del circuito lógico.

1 Introduction

A introduction a ver asdf

2 Taller 1 : Funciones de alto orden:

Para el desarrollo de este taller, se utilizaron las siguientes funciones en scala:

3 Funciones de alto orden implementadas

A continuación, se presenta la funciones implementadas de alto orden, las cuales fueron utilizadas para instanciar otras funciones (funciones generadoras), a través de su paso como parámetro, ya sea referenciada (nominada) o como funcion anónima(inline), o como valor retorno de la misma.

4 Crear chip unario

4.1 Informe de procesos

Realiza una operación logica sobre un solo valor de entrada. A continuación, se presenta su implementación en *Scala*

```
1   def crearChipUnario( f: Int => Int ) : Chip = (arg:
       List[Int]) => { // Apply the f function on the head
        of current list and call recursively the
       crearChipUnarioHelper with function f, a
       accumulated list with new transformed value as its
       head, and the current list tail, until the empty
       list condition is reached.
2      @tailrec
3      def crearChipUnarioHelper(f: Int => Int,
          transformedList: List[Int],  currentList: List[
          Int]): List[Int] =
4        if (currentList.isEmpty) transformedList
5        else crearChipUnarioHelper(f, f(currentList.head)
            ::transformedList, currentList.tail)
6
7      // The initial state of the iteration
8      crearChipUnarioHelper(f, List(), arg)
9   }
```

Listing 1: Aplica una operación binaria sobre una valor de entrada.

4.2 Informe de corrección

$val\ chip\_not = crearchipUnario(x => 1 - x)$

Caso 1:
$chip\_not((List(0)))$
$\rightarrow crearChipUnarioHelper(x => 1 - x, [\,],\ List(0))$
$\rightarrow if(List(0).isEmpty)\ [\,]$
$\quad else\ crearChipUnarioHelper(x => 1 - x, (1 - 0) :: [\,], [\,])$
$\rightarrow if(List().isEmpty)\ [1]$
$\quad else\ crearChipUnarioHelper(x => 1 - x, (1 - 1) :: [1], [\,])$
$\rightarrow [1]$

Caso 2:
$chip\_not((List(1)))$
$\rightarrow crearChipUnarioHelper(x => 1 - x, [\,], List(1))$
$\rightarrow if(List(1).isEmpty)\ [\,]$
$\quad else\ crearChipUnarioHelper(x => 1 - x, (1 - 1) :: [\,], [\,])$
$\rightarrow if(List().isEmpty)\ [0]$
$\quad else\ crearChipUnarioHelper(x => 1 - x, (1 - 0) :: [0], [\,])$
$\rightarrow [0]$

5 Crear chip binario

5.1 Informe de procesos

Realiza una operación logica sobre un solo valor de entrada. A continuación, se presenta su implementación en *Scala*

```
1  def crearChipUnario( f: Int => Int ) : Chip = (arg:
       List[Int]) => { // Apply the f function on the head
        of current list and call recursively the
       crearChipUnarioHelper with function f, a
       accumulated list with new transformed value as its
       head, and the current list tail, until the empty
       list condition is reached.
2    @tailrec
3    def crearChipUnarioHelper(f: Int => Int,
         transformedList: List[Int], currentList: List[
         Int]): List[Int] =
4      if (currentList.isEmpty) transformedList
5      else crearChipUnarioHelper(f, f(currentList.head)
           ::transformedList, currentList.tail)
6
7    // The initial state of the iteration
8    crearChipUnarioHelper(f, List(), arg)
9  }
```

Listing 2: Aplica una operación binaria sobre una valor de entrada.

5.2 Informe de corrección

$val\ chip\_and = crearChipBinario((x : Int, y : Int) => x * y\ )$

Caso 1:
$chip\_and((List(0, 0)))$
$\rightarrow crearChipBinarioHelper((x : Int, y : Int) => x * y, [\ ],\ List(0,\ 0))$
$\rightarrow if(List(0,\ 0).isEmpty | List(0,\ 0).tail.isEmpty)\ [\ ]$
$\quad else\ crearChipBinarioHelper((x : Int, y : Int) => x * y, (0 * 0) :: [\ ], [\ ])$
$\rightarrow if(List().isEmpty)\ [0]$
$\rightarrow [0]$

Caso 2:
$chip\_and((List(0, 1)))$
$\rightarrow crearChipBinarioHelper((x : Int, y : Int) => x * y, [\ ],\ List(0,\ 1))$
$\rightarrow if(List(0,\ 1).isEmpty | List(0,\ 1).tail.isEmpty)\ [\ ]$
$\quad else\ crearChipBinarioHelper((x : Int, y : Int) => x * y, (0 * 1) :: [\ ], [\ ])$
$\rightarrow if(List().isEmpty)\ [0]$
$\rightarrow [0]$

Caso 2:
$chip\_not((List(0)))$
$\rightarrow crearChipUnarioHelper(x => 1 - x, [\ ],\ List(0))$
$\rightarrow if(List(0).isEmpty)\ [\ ]$
$\quad else\ crearChipUnarioHelper(x => 1 - x, (1 - 0) :: [\ ], [\ ])$
$\rightarrow if(List().isEmpty)\ [1]$
$\quad else\ crearChipUnarioHelper(x => 1 - x, (1 - 1) :: [1], [\ ])$
$\rightarrow [1]$

Caso 3:
$chip\_and((List(1,\ 1)))$
$\rightarrow crearChipBinarioHelper((x : Int, y : Int) => x * y, [\ ],\ List(1,\ 1))$
$\rightarrow if(List(1,\ 1).isEmpty | List(1,\ 1).tail.isEmpty)\ [\ ]$
$\quad else\ crearChipBinarioHelper((x : Int, y : Int) => x * y, (1 * 1) :: [\ ], [\ ])$
$\rightarrow if(List().isEmpty)\ [1]$
$\rightarrow [1]$

5.3 Informe de corrección

$val\ chip\_or = crearChipBinario((x : Int, y : Int) => x * y\ )$

Caso 1:
$chip\_or((List(0, 0)))$
$\rightarrow crearChipBinarioHelper((x : Int, y : Int) => (x+y)-(x*y), [\,],\ List(0,\ 0))$
$\rightarrow if(List(0,\ 0).isEmpty | List(0,\ 0).tail.isEmpty)\ [\,]$
$\quad else\ crearChipBinarioHelper((x : Int, y : Int) => (x+y)-(x*y), ((0+0)-(0*0)) :: [\,], [\,])$
$\rightarrow if(List().isEmpty)\ [0]$
$\rightarrow [0]$

Caso 2:
$chip\_or((List(0, 1)))$
$\rightarrow crearChipBinarioHelper((x : Int, y : Int) => x * y, [\,],\ List(0,\ 1))$
$\rightarrow if(List(0,\ 1).isEmpty | List(0,\ 1).tail.isEmpty)\ [\,]$
$\quad else\ crearChipBinarioHelper((x : Int, y : Int) => (x+y)-(x*y), ((0+1)-(0*1)) :: [\,], [\,])$
$\rightarrow if(List().isEmpty)\ [1]$
$\rightarrow [1]$

Caso 3:
$chip\_or((List(1, 0)))$
$\rightarrow crearChipBinarioHelper((x : Int, y : Int) => x * y, [\,],\ List(1,\ 0))$
$\rightarrow if(List(0,\ 1).isEmpty | List(1,\ 0).tail.isEmpty)\ [\,]$
$\quad else\ crearChipBinarioHelper((x : Int, y : Int) => (x+y)-(x*y), ((1+0)-(1*0)) :: [\,], [\,])$
$\rightarrow if(List().isEmpty)\ [1]$
$\rightarrow [1]$

Caso 4:
$chip\_or((List(1,\ 1)))$
$\rightarrow crearChipBinarioHelper((x : Int, y : Int) => x * y, [\,],\ List(1,\ 1))$
$\rightarrow if(List(1,\ 1).isEmpty | List(1,\ 1).tail.isEmpty)\ [\,]$
$\quad else\ crearChipBinarioHelper((x : Int, y : Int) => x*y, ((1+1)-(1*1)) :: [\,], [\,])$
$\rightarrow if(List().isEmpty)\ [1]$
$\rightarrow [1]$

5.4 Informe de corrección

$half\_adder(List(0,\ 0))$
$\rightarrow val\ and\_op\_in = chip\_add(List(0,\ 0))$
$\quad val\ or\_op = chip\_or(List(0,\ 0))$
$\quad val\ and\_op\_out = chip\_add(List(0)\ \ ++\ \ chip\_not(List(0)))$
$\quad List(0)\ \ ++\ \ List(0)$
$\rightarrow [0,\ 0]$

$half\_adder(List(0,\ 1))$
$\rightarrow val\ and\_op\_in = chip\_add(List(0,\ 1))$
$\quad val\ or\_op = chip\_or(List(0,\ 1))$
$\quad val\ and\_op\_out = chip\_add(List(1)\ \ ++\ \ chip\_not(List(0)))$
$\quad List(1)\ \ ++\ \ List(0)$
$\rightarrow [1,\ 0]$

$half\_adder(List(1,\ 0))$
$\rightarrow val\ and\_op\_in = chip\_add(List(1,\ 0))$
$\quad val\ or\_op = chip\_or(List(1,\ 0))$
$\quad val\ and\_op\_out = chip\_add(List(1)\ \ ++\ \ chip\_not(List(0)))$
$\quad List(1)\ \ ++\ \ List(0)$
$\rightarrow [1,\ 0]$

$half\_adder(List(1,\ 1))$
$\rightarrow val\ and\_op\_in = chip\_add(List(1,\ 1))$
$\quad val\ or\_op = chip\_or(List(1,\ 1))$
$\quad val\ and\_op\_out = chip\_add(List(1)\ \ ++\ \ chip\_not(List(1)))$
$\quad List(0)\ \ ++\ \ List(1)$
$\rightarrow [0,\ 1]$

5.5 Informe de corrección

Caso 0 + 0, Carriage:=0:
$full\_adder(List(0,\ 0,\ 0))$
$\rightarrow\ \ val\ halfAdder\_1 = half\_adder(0 :: 0 :: Nil)$
$\quad val\ halfAdder\_2 = half\_adder(0 :: 0 :: Nil)$
$\quad val\ or\_op = chip\_or(List(0)\ \ ++\ \ List(0))$
$\quad List(0)\ \ ++\ \ List(0)$
$\rightarrow [0,\ 0]$

Caso 0 + 0, Carriage:=0:
$full\_adder(List(0,\ 0,\ 1))$
$\rightarrow\ val\ halfAdder\_1 = half\_adder(0 :: 1 :: Nil)$
$val\ halfAdder\_2 = half\_adder(0 :: 1 :: Nil)$
$val\ or\_op = chip\_or(List(0)\ \ ++\ \ List(0))$
$List(0)\ \ ++\ \ List(1)$
$\rightarrow [0,\ 1]$

Caso 1 + 0:
$full\_adder(List(0,\ 1,\ 1))$
$\rightarrow\ val\ halfAdder\_1 = half\_adder(1 :: 1 :: Nil)$
$val\ halfAdder\_2 = half\_adder(0 :: 1 :: Nil)$
$val\ or\_op = chip\_or(List(0)\ \ ++\ \ List(0))$
$List(0)\ \ ++\ \ List(1)$
$\rightarrow [0,\ 1]$

Caso 0 + 1, Carriage:=0:
$full\_adder(List(0,\ 1,\ 0))$
$\rightarrow\ val\ halfAdder\_1 = half\_adder(1 :: 0 :: Nil)$
$val\ halfAdder\_2 = half\_adder(0 :: 1 :: Nil)$
$val\ or\_op = chip\_or(List(0)\ \ ++\ \ List(0))$
$List(0)\ \ ++\ \ List(1)$
$\rightarrow [0,\ 1]$

Caso 1 + 1, Carriage:=0:
$full\_adder(List(1,\ 1,\ 0))$
$\rightarrow\ val\ halfAdder\_1 = half\_adder(1 :: 0 :: Nil)$
$val\ halfAdder\_2 = half\_adder(1 :: 1 :: Nil)$
$val\ or\_op = chip\_or(List(0)\ \ ++\ \ List(1))$
$List(1)\ \ ++\ \ List(0)$
$\rightarrow [1,\ 0]$

Caso 1 + 1, Carriage:=1:
$full\_adder(List(1,\ 1,\ 1))$
$\rightarrow\ val\ halfAdder\_1 = half\_adder(1 :: 1 :: Nil)$
$val\ halfAdder\_2 = half\_adder(1 :: 0 :: Nil)$
$val\ or\_op = chip\_or(List(1)\ \ ++\ \ List(0))$
$List(1)\ \ ++\ \ List(0)$

$\rightarrow [1,\ 0]$

6 Implementnación de adder

6.1 Informe de corrección

En la siguiente figura se muestra la implementación algorítmica de un sumador de n dígitos, construido a partir de las funciones previamente desarrolladas y utilizadas de forma declarativa:

```
1   def adder ( n : Int ) : Chip = (operands: List[Int])
      => {
2    /**
3    * Return a tuple of two list splited in n elements
4        * @param n Number to split the original list
5        * @param lowerList A list that accumlates the
           first n elements
6        * @param upperList A list that ends with last n
           elements
7        * @return Tuple(List[Int], List[Int])  Two list
           of the length n
8      */
9    @tailrec
10   def splitList(n: Int, counter: Int, lowerList: List[
       Int], upperList: List[Int]): (List[Int], List[Int
       ]) =
11     if( (n + 1)  == counter ) (lowerList, upperList)
12     else splitList(n , counter + 1,   upperList.head::
         lowerList, upperList.tail)
13
14   // Reverse a list
15   def reverse(l: List[Int]): List[Int] =  if(l.isEmpty
       ) l else reverse(l.tail) ++ List(l.head)
16
17   val (l1, l2) = splitList(n, 1, List(), operands)  //
       Call the splitList function and save its result
      in a tuple of Int List
18   val l2rv = reverse(l2) // Reverse the list for set
      the lower significant digits to front of the list
```

```
        for later bcause the other list is so
       connfigured
19
20      /**
21     * Calculate recursively the sum of the two number of
        n digits
22         * @param accumulatedList The parcial accumulated
            result list; The tail of this list is the
           caariage of the operation in every step
23         * @param firstList A list that represents the
           first number
24         * @param lastList A list that represents the
           second number
25         * @return List[Int] The number resulted of the
           sum operation
26     */
27     @tailrec
28     def adderHelper( accumulatedList: List[Int],
          firstList:List[Int],  secondList: List[Int] ):
          List[Int] = {
29       if(firstList.isEmpty || secondList.isEmpty)
           accumulatedList     // Check if the some list
          is empty to end the recursion proccess
30       else {
31         val fullAddResult = full_adder(firstList.head::
             secondList.head::accumulatedList.head::Nil)
             // Pass in to the fulladder: the head of the
             first list, the head of second list and the
             head of the accumulated list which has the
             carriage of previous full_adder operation
32         return adderHelper( fullAddResult ++
             accumulatedList.tail, firstList.tail,
             secondList.tail ) // Const he partial result
             list with previous adder result, and pass in
             the rest(tail) of both list.
33       }
34     }
35     val initial_sum = half_adder(l1.head::l2rv.head::Nil
          ) // Determine the first digiit sum without
          carriage input
36     adderHelper( initial_sum.tail.head::initial_sum.head
          ::Nil, l1.tail, l2rv.tail) // Arrange the
          accumulated list so the carriage is in the head
```

```
37    }
38  }
```

Listing 3: Aplica una operación binaria sobre una valor de entrada.

Lo que hace adder, sucitamente, es calcular primeramente la suma del digito menos significativo utilizando semi-sumador y, a posteriori, resultado lo anterior como el estado inicial de iteración, entra en un proceso iterativo donde en cada paso iterativo aplica el sumador completo con la cabeza de las dos listas implicadas como operandos; y asi, sucesivamente hasta exhaustar, o llegar a consumir, los elementos de alguna de las listas. Tambien se observa, como parte de la solución, un función splitList que divide la lista de entrada al adder en dos sublista de n elementos cada una. Por último, para alinear potencias en sus posiciones correspondientes, como se definió en en la lista argumento de la función, se reversa la lista de la parte superior de la lista original. En consecuencia, las operaciones quedan primero en aplicadas en los dígitos menos significativos y luego acumula hasta los más significativos.

$$
\begin{aligned}
&\quad val\ add\_4 = adder(4)\\
&add\_4(List(0,0,1,1)\ \ ++\ \ List(1,1,1,1))\\
&\rightarrow splitList(4,0,List(),List(0,0,1,1,1,1,1,1))\\
&\rightarrow\ if(4==0)\ (List(),List(0,0,1,1,1,1,1,1))\\
&\quad else\ spplitList(4,1,0::List(),List(0,1,1,1,1,1,1))\\
&\twoheadrightarrow\ if(4==4)\ (List(0,0,1,1),\ List(1,1,1,1))\\
&\rightarrow\ val\ (l1,\ l2) = (List(0,0,1,1),\ List(1,1,1,1))\\
&\rightarrow\ adderHelper(half\_adder(0::1::Nil),List(0,1,1),List(1,1,1))\\
&\rightarrow\ adderHelper(List(1,0),List(0,1,1),List(1,1,1))\\
&\rightarrow\ if(List(0,1,1).isEmpty||List(1,1,1).isEmpty)List(1,\ 0)\\
&\quad val\ fullAddResult = full\_adder(0::1::1::Nil)\\
&\quad adderHelper(List(1,0)++List(0),List(1,1),List(1,1))\\
&\twoheadrightarrow\ if(List().isEmpty\ ||\ List().isEmpty)\ List(1,0,0,1,0)\\
&\rightarrow\ [1,0,0,1,0]
\end{aligned}
$$

6.2 Corrección de procesos

Sea f una función $B \rightarrow B$ donde $B = \{b | b =< b_n, ...b_i, ..., b_0... > \ con\ b_i \in 0,\ 1\}$, es decir, una funcion con dominio e imagen en los números binarios, y sea Pf una implementación de f con una aplicación $List[B] \rightarrow List[B]$ realizada en el leguaje de programación *Scala*. Se tiene:

- Un estado inicial $s_0 = (l_0, l_{n-1\ inf}, l_{n-1\ sup})$ donde los subindices connotan un lista de tamaño n - 1, y la parte superior e inferior de la lista original, respectivamente.
- *head*: Int (devuelve si una lista l esta vacia).
- *tail*: List[Int] (devuelve la lista sin el primer elemento l).
- $x :: l$: devuelve la lista que representa la secuencia $< x, x_1, x_2, ..., x_n >$ si l es la lista que representa la secuencia $< x_1, x_2, ..., x_n >$.
- $l1 + +l2$ devuelve la lista que representa la concatenacion de las secuencias representadas por $l1$ y $l2$.

7 Conclusion

La conclusion

$$a = \sum F\dot{m} = \frac{dv}{dt}$$