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

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

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

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

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

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

6 Conclusion

La conclusion

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