Universidad del Valle

Facultad de ingeniería

Ingeniería en sistemas

Cristian David Pacheco Torres
2227437
Juan Sebastian Molina Cuellar
2224491

28 de Septiembre del 2022

Taller 2: Simular un sumador de n digitos a partir de compuertas lógicas sencillas.

Contents

1 Creación de las compuertas sencillas.

1.1 Creando compuertas unarias.

A partir de una función de tipo $Int => Int$ se crea la función **crearChipUnario**, que devuelve un $Chip$ correspondiente al procesar el único bit de la lista de entrada.
A continuación se muestra el código de la función implementada:

```
1 def crearChipUnario( f: Int => Int ) : Chip = (arg: List[Int]) => {
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   crearChipUnarioHelper(f, List(), arg)
7 }
```

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

1.1.1 Informe de procesos.

Tipo de proceso.
El algoritmo *crearChipUnario* utiliza un un proceso **recursivo lineal**, debido a que a pesar de que no se expande, se contrae cada vez que se llama recursivamente (en este caso a su helper *crearChipUnarioHelper*) y entra a su *else* (ver linea 5 del Listing 1).

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

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

1.1.2 Informe de corrección.

Dada una función $f : \mathbb{Z} \rightarrow \mathbb{Z}$,y una lista L de longitud n, la función $crearChipUnario$ devuelve una lista L' que aplica f a cada elemento de la lista de entrada.
Y sea P_f el anterior programa realizado en Scala, que implementa a f y al cual se quiere demostrar su correctitud.
Hipótesis:

$$\forall L \in \mathbb{Z}^n, \forall i \in \{1, \ldots, n\}, P_f(f, L)_i = f(L_i)$$

- **Caso base.**
  La lista de entrada es vacía, por lo tanto $n = 0$.
  Puesto que no hay elementos que procesar, la lista de salida es vacía.
  Esto satisface la definición de la función.

- **Paso inductivo.**
  Supongamos que P_f es correcto para una lista de longitud k. Ahora, queremos probar que también es correcto para una lista de longitud $k + 1$.

  Hipótesis:

  $$\forall L \in \mathbb{Z}^k, \forall i \in \{1, \ldots, k\}, P_f(f, L)_i = f(L_i)$$

  Paso a demostrar:

  $$\forall L' \in \mathbb{Z}^{k+1}, \forall j \in \{1, \ldots, k + 1\}, P_f(f, L')_j = f(L'_j)$$

- **Demostración del Paso Inductivo.**
  Consideremos una lista L' de longitud $k+1$ donde $L' = L \cup \{x\}$, siendo x el elemento adicional.

Al aplicar P_f a L', el primer elemento x se transforma usando f y el resultado se añade al principio de la lista transformada. Luego, P_f se aplica al resto de la lista L.

Por la hipótesis de inducción, sabemos que $P_f(f, L)_i = f(L_i)$ para todos los i en L. Por lo tanto, cada elemento en L se transformará correctamente.

El elemento adicional x también se transformará correctamente, ya que simplemente se aplica f a x.

Por lo tanto, $P_f(f, L')_j = f(L'_j)$ para todos los j en L', lo que completa la demostración del paso inductivo.

1.1.3 Casos de pruebas.

```scala
val chip_not = crearChipUnario((x: Int) => (1 - x))
chip_not(List(0)) // Deberia imprimir [1]
chip_not(List(1)) // Deberia imprimir [0]
chip_not(List(-1))  // Deberia imprimir [2]
chip_not(List(3))  // Deberia imprimir [-2]
chip_not(List(99))  // Deberia imprimir [-98]
```

Listing 2: Casos de prueba para la función crearChipUnario.

Anotaciones:

- La función *crearChipUnario* es correcta, puesto que cumple con la definición de la función, la forma idonea de llegar a esta conclusion independiente de los casos de prueba que se hagan es usando un metodo de demostración formal.

- A pesar de que la función *crearChipUnario* se pretende usar para negar un bit, se puedodria usar para cualquier función $f : \mathbb{Z} \to \mathbb{Z}$ y a su vez para cualquier entero, por eso se hacen pruebas con otros enteros y no solo bits para probar su funcionamiento.

- Gracias a los casos de prueba, se pudo caer en cuenta de que la funcion que teniamos implementada inicialmente, si cumplia con su deber, pero retornaba la lista de forma invertida si fuera el caso de que le entrara una lista con un tamaño mayor a 1. Si este tipo de caso estuviese en cuestión, se podria usar la función *reverse* de Scala para invertir la lista de salida.

1.2 Creando compuertas binarias.

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

```
1   def crearChipBinario ( f: (Int, Int) => Int ) : Chip = (arg: List[Int]) => {
2     def crearChipBinarioHelper(f: (Int, Int) => Int, transformedList: List[Int], currentList: List[Int]): List[Int] =
3       if( currentList.isEmpty || currentList.tail.isEmpty) transformedList
4       else crearChipBinarioHelper(f, f(currentList.head, currentList.tail.head)::transformedList, currentList.tail.tail)
5   crearChipBinarioHelper(f, List(), arg)
6 }
```

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

1.2.1 Informe de procesos.

Tipo de proceso.
Debido a que en la llamada recursiva (ver linea 4 del Listing 3), el problema se reduce al pasar por *currentList.tail.tail* como el nuevo argumento de la lista. Esto asegura que, la lista estara en este caso en un proceso de contracción y este tipo de procesos son caracteristicos de los procesos **recursivos lineales**.

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

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:
$chip\_or((List(0, 1)))$
$\rightarrow crearChipBinarioHelper((x : Int, y : Int) => (x+y)-(x*y), [\,],\ List(0,\ 1))$

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

1.2.2 Informe de corrección.

Sea P_f la función ‘crearChipBinario‘ que aplica una función binaria f a pares consecutivos de elementos en una lista L.
Sea L es una lista de números enteros de longitud n.
Sea $f : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ es una función binaria.
Y sea $P_f(L)$ denota el resultado de aplicar P_f a la lista L.

Hipótesis:

$$\forall L \in \mathbb{Z}^{2n}, P_f(L) = [f(L_1, L_2), f(L_3, L_4), \ldots, f(L_{2n-1}, L_{2n})]$$

- **Caso Base.**
  $n = 1$ (la lista tiene dos elementos) Para una lista $L = [a, b]$:

  $$P_f(L) = [f(a, b)]$$

- **Paso Inductivo.**
  Supongamos que la hipótesis es verdadera para alguna lista L de longitud $2k$. Queremos demostrar que es verdadera para una lista de longitud $2(k + 1)$.

  Hipótesis:

  $$\forall L \in \mathbb{Z}^{2k}, P_f(L) = [f(L_1, L_2), f(L_3, L_4), \ldots, f(L_{2k-1}, L_{2k})]$$

  Paso a demostrar:

  $$\forall L' \in \mathbb{Z}^{2(k+1)}, P_f(L') = [f(L'_1, L'_2), f(L'_3, L'_4), \ldots, f(L'_{2k+1}, L'_{2k+2})]$$

- **Demostración del Paso Inductivo.**

  Consideremos una lista L' de longitud $2(k + 1)$ donde $L' = L \cup \{x, y\}$, siendo x y y los elementos adicionales.

  Al aplicar P_f a L', los primeros elementos x y y se transforman usando f y el resultado se añade al principio de la lista transformada. Luego, P_f se aplica al resto de la lista L.

Por la hipótesis de inducción, sabemos que $P_f(L)$ transforma correctamente los elementos en L. El par adicional (x, y) también se transformará correctamente, ya que simplemente se aplica f a x y y.

Por lo tanto, todos los elementos en L' se transformarán correctamente, lo que completa la demostración del paso inductivo.

1.2.3 Casos de pruebas.

```
val chip_add = crearChipBinario((x: Int, y: Int) => (x * y))
chip_add(List(0, 1))  // Deberia imprimir [0]
chip_add(List(1, 1))  // Deberia imprimir [1]
chip_add(List(1, 0))  // Deberia imprimir [0]
chip_add(List(0,0))  // Deberia imprimir [0]
chip_add(List(-1, 1)) // Deberia imprimir [-1]
val chip_or = crearChipBinario((x: Int, y: Int) => (x + y) - (x * y ))
chip_or(List(0, 0))   // Deberia imprimir [0]
chip_or(List(0, 1))   // Deberia imprimir [1]
chip_or(List(1, 0))   // Deberia imprimir [1]
chip_or(List(1, 1))   // Deberia imprimir [1]
chip_or(List(-1, 0))  // Deberia imprimir [-1]
```

Listing 4: Casos de prueba para la función crearChipBinario.

Anotaciones:

- Evaluando todos los casos de las posibles combinaciones de 2 bits, se puede concluir que la función *crearChipBinario* es correcta.
- La demostración formal nos lleva a pensar de que se pueden hacer casos de prueba para diferentes valores, que no solo sean un par de bits.
- Debido a que el algoritmo es similar al anterior, si se usara una lista de bits o de enteros de mayor tamaño, el proceso terminaria con una $List[n/2]$ y estaria invertida.

2 Creando *semisumadores*.

Construye un *Chip* correspondiente a un *semisumador* a partir de compuertas lógicas sencillas.

```
def half_adder : Chip = ( operands: List[Int]) => {
  val add = chip_add(operands)
  val or = chip_or(operands)
  val not_add = chip_not(add)
  val s = chip_add(or ++ not_add)
  val c = add
  c ++ s
}
```

Listing 5: SemiSumador implementado

2.1 Informe de procesos

Tipo de proceso.
Puesto a que el algoritmo $half\_adder$ no se expande ni se contrae, ni se llama recursivamente a si mismo, la deducción es que el proceso depende de las funciones de las que esta compuesto, en este caso *chip_add*, *chip_or* y *chip_not*. Por lo tanto se considera el algoritmo como un proceso **recursivo**.

$$
\begin{aligned}
&\quad half\_adder(List(0,\ 0)) \\
&\rightarrow val\ add = chip\_add(List(0,\ 0)) \\
&\twoheadrightarrow val\ add = List[0] \\
&\rightarrow val\ or = chip\_or(List(0,\ 0)) \\
&\twoheadrightarrow val\ or = List[0] \\
&\rightarrow val\ not\_add = chip\_not(add) \\
&\twoheadrightarrow val\ not\_add = List[1] \\
&\rightarrow val\ s = chip\_add(or + +not\_add) \\
&\twoheadrightarrow s = List[0] \\
&\rightarrow val\ c = List[0] \\
&\twoheadrightarrow List(0)\quad + +\quad List(0) \\
&\rightarrow [0,\ 0]
\end{aligned}
$$

2.2 Informe de corrección

Puesto que el programa $half\_adder(chip)$ (ver Listing 5) retorna $c++s$. Teniendo demostrados anteriormente todos los subprocesos que $half\_adder(chip)$ ocupa, el retorno de esta operación es $c++s$.
Para demostrar esto tenemos:
consideremos a f como la funcion que retorna un semi sumador, tal que $f(chip(a,b)) = c++s$.

Sean A y B, dos bits (en cualquier combinación) la operación S se define como:

$$A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$$

Y C como la función que retorna un sumador, tal que:

$$C = chip\_add(chip(A,B)) = A \wedge B$$

Si $half\_adder(chip)$ retorna $c++s$, entonces $c = C$ y $s = S$.

Demostración.

$$half\_adder(List(A,\ B))$$

$$\rightarrow val\ add = chip\_add(List(A,\ B))$$

$$\twoheadrightarrow val\ add = A \wedge B$$

$$\rightarrow val\ or = chip\_or(List(A,\ B))$$

$$\twoheadrightarrow val\ or = A \vee B$$

$$\rightarrow val\ not\_add = chip\_not(add)$$

$$\twoheadrightarrow val\ not\_add = \urcorner(A\ \wedge\ B)$$

$$\rightarrow val\ s = chip\_add(or++not\_add)$$

$$\twoheadrightarrow s = (A \vee B) \wedge (\urcorner(A\ \wedge\ B))$$

$$\text{Alpicando distribucion: } (A \wedge \urcorner(A \wedge B)) \vee (B \wedge \urcorner(A \wedge B))$$

$$\text{Expandiendo la forma: } \urcorner(A \wedge B) \therefore\ \urcorner A \vee \urcorner B$$

$$(A \wedge (\urcorner A \vee \urcorner B)) \vee (B \wedge (\urcorner A \vee \urcorner B))$$

$$\text{Por distribucion nuevamente: } (A \wedge \urcorner A) \vee (A \wedge \urcorner B) \vee (B \wedge \urcorner A) \vee (B \wedge \urcorner B)$$

Pero tenemos dos casos que siempre son falsos (se contradicen): $(A \wedge \urcorner A) \vee (B \wedge \urcorner B)$

Por lo tanto nos queda: $S = (A \land \neg B) \lor (\neg A \land B)$

$$\rightarrow val\ c = A \land B$$

Puesto que desde $half\_adder$ se llego a la misma expresión que S y C, se puede concluir que $half\_adder$ es correcto.

2.3 Casos de pruebas

```
val ha = half_adder
ha(List(0, 0))  // Deberia imprimir [0, 0]
ha(List(0, 1))  // Deberia imprimir [0, 1]
ha(List(1, 0))  // Deberia imprimir [0, 1]
ha(List(1, 1))  // Deberia imprimir [1, 0]
ha(List(1, 1))  // Deberia imprimir [1, 0]
```

Listing 6: Casos de prueba para la función half_adder.

Anotaciones:

- Evaluando todos los casos de las posibles combinaciones de 2 bits, se puede concluir que la función half_adder hace su proceso de manera correcta.
- A través de la lógica proposicional es posible demostrar a través de equivalencias una función de alto nivel.
- La función half_adder no solo calcula la suma, sino que también determina correctamente el acarreo para cada combinación de entrada..

3 Creando *sumadores completos*.

Construye un *chip* correspondiente a un *sumador completo* a partir de compuertas lógicas sencillas.

```
def full_adder : Chip = (operands: List[Int]) => {
    val halfAdder_1 = half_adder(operands.head::
       operands.tail.head::Nil)
    val halfAdder_2 = half_adder(halfAdder_1.tail.head
       ::operands.tail.tail.head::Nil)
    val C_out = chip_or(halfAdder_1.head::halfAdder_2.
       head::Nil).head
    C_out::halfAdder_2.tail.head::Nil
}
```

Listing 7: Sumador completo implementado

3.1 Informe de procesos

Tipo de proceso.
El algoritmo $full\_adder$ se puede considerar como un proceso **recursivo**, debido a que a pesar de que no se llama recursivamente a si mismo y no tiene proceso iterativo, llama funciones que ya hemos desarrollado anteriormente que fueron catalogadas como **recursivas**.

$full\_adder(List(0,\ 0,\ 0))$
$\rightarrow val\ half\_Adder\_1 = half\_adder(List(0,\ 0))$
$\twoheadrightarrow half\_Adder\_1 = [0,\ 0]$
$\rightarrow val\ half\_Adder\_2 = half\_adder(List(0,\ 0))$
$\twoheadrightarrow half\_Adder\_2 = [0,\ 0]$
$\rightarrow val\ or\_op = chip\_or(List(0,\ 0))$
$\twoheadrightarrow or\_op = List[0]$
$\rightarrow List(0)\ \ ++\ \ List(0)$
$\rightarrow [0,\ 0]$

3.2 Informe de corrección

Dado que $half\_adder$ ya ha sido demostrado, podemos usarlo para construir el $full\_adder$.

$$
\begin{gathered}
full\_adder(List(A, B, C_{in})) \\
\rightarrow val\ half\_Adder\_1 = half\_adder(List(B, C_{in})) \\
\twoheadrightarrow half\_Adder\_1 = [S_1, C_1] \\
\rightarrow val\ half\_Adder\_2 = half\_adder(List(A, S_1)) \\
\twoheadrightarrow half\_Adder\_2 = [S, C_2] \\
\rightarrow val\ or\_op = chip\_or(List(C_1, C_2)) \\
\twoheadrightarrow or\_op = [C_{out}] \\
\rightarrow [C_{out}, S]
\end{gathered}
$$

Donde: S_1 es la suma de B y C_{in} usando el primer $half\_adder$. C_1 es el acarreo de la suma de B y C_{in}. S es la suma final que es el resultado de sumar A con S_1 usando el segundo $half\_adder$. C_2 es el acarreo de la suma de A y S_1. C_{out} es el acarreo final que es el resultado de la operación OR entre C_1 y C_2.

Por lo tanto, el $full\_adder$ produce correctamente la suma S y el acarreo C_{out} para cualquier combinación de entradas A, B, y C_{in}.

3.3 Casos de pruebas

```
val fa = full_adder

fa(List(0, 0, 0))  // Deberia imprimir [0, 0]
fa(List(0, 0, 1))  // Deberia imprimir [0, 1]
fa(List(0, 1, 0))  // Deberia imprimir [0, 1]
fa(List(0, 1, 1))  // Deberia imprimir [1, 0]
fa(List(1, 0, 0))  // Deberia imprimir [0, 1]
fa(List(1, 0, 1))  // Deberia imprimir [1, 0]
fa(List(1, 1, 0))  // Deberia imprimir [1, 0]
fa(List(1, 1, 1))  // Deberia imprimir [1, 1]
```

Listing 8: Casos de prueba para la función full_adder.

Anotaciones:

- Gracias al uso de casos de pruebas se pudo corregir un error de codigo que hacia que no retornara los valores correctos al calcular la funcion.
- Evaluando todos los casos de las posibles combinaciones de 3 bits, se puede concluir que la función full_adder hace su proceso de manera correcta.

4 Construyendo un *sumador* $- n$

Se construye un sumador que implementando la funcion adder construye un chip correspondiente a un sumador-n.

```


def adder ( n : Int ) : Chip = (operands: List[Int]) => {
  def splitList(n: Int, counter: Int, lowerList: List[Int], upperList: List[Int]): (List[Int], List[Int]) =
   { if( (n + 1)  == counter ) (lowerList, upperList)
    else splitList(n , counter + 1, upperList.head::lowerList, upperList.tail)

  val (l1, l2) = splitList(n, 1, List(), operands)
  def adderHelper( accumulatedList: List[Int], firstList:List[Int],  secondList: List[Int] ): List[Int] = {
    if(firstList.isEmpty || secondList.isEmpty) accumulatedList
      val fullAddResult = full_adder(firstList.head::secondList.head::accumulatedList.head::Nil)
      return adderHelper( fullAddResult ++ accumulatedList.tail, firstList.tail, secondList.tail )
    }
  }
  val initial_sum = half_adder(l1.head::l2.head::Nil)
  adderHelper( initial_sum.tail.head::initial_sum.head::Nil, l1.tail, l2.tail)
}
```

Listing 9: sumador-n.

4.1 Informe de procesos

Tipo de proceso.
Bajo el mismo razonamiento utilizado anteriormente, consideramos que *adder* utiliza un proceso **recursivo**.

val $fa = adder$

$fa(List(1,0,1,0) + +List(0,1,1,1))$

$\rightarrow$ splitList$(4, 1, List(), List(1,0,1,0,0,1,1,1))$

$\twoheadrightarrow$ if$(4 == 1)$ $(List(), List(1,0,1,0,0,1,1,1))$

else splitList$(4, 2, 1 :: List(), List(0,1,0,0,1,1,1))$

$\twoheadrightarrow$ if$(4 == 4)$ $(List(1,0,1,0), List(0,1,1,1))$

$\rightarrow$ val $(l1, l2) = (List(1,0,1,0), List(0,1,1,1))$

$\rightarrow$ adderHelper(half_adder$(1 :: 0 :: Nil), List(0,1,0), List(1,1,1))$

$\rightarrow$ adderHelper$(List(0,1), List(0,1,0), List(1,1,1))$

$\rightarrow$ if$(List(0,1,0).isEmpty$ or $List(1,1,1).isEmpty)List(0,1)$

else

val fullAddResult = full_adder$(0 :: 1 :: 0 :: Nil)$

return adderHelper(List(1, 0) + +List(0)$, List(1,0), List(1,1))$

$\twoheadrightarrow$ if$(List(1,0).isEmpty$ or $List(1,1).isEmpty)$ List(1,1,0,1,1)

$\rightarrow [1,1,0,1,1]$

4.2 Informe de corrección

4.3 Casos de pruebas