Informe Taller 2: Funciones de alto orden

Santiago Avalo Monsalve - 2359442
Daniel Gómez Cano - 2359396
Edward Stivens Pinto - 2359431

Trabajo presentado a:
Carlos Andrés Delgado S.

Universidad del Valle
Fundamentos de Programación Funcional y Concurrente
Ingeniería de Sistemas
Tuluá - Valle del Cauca
2024

1. Informe de Procesos

1.1 Método: pertenece

```
5   type ConjDifuso = Int => Double
6
7   def pertenece ( elem : Int , s : ConjDifuso ) : Double = {
8       s(elem)
9   }
10
```

La Función recibe un “elem” del tipo int y lo evalúa en una función del tipo ConjDifuso.
Determina el grado de pertenencia de un elemento a un conjunto difuso. Toma un entero elem y un ConjDifuso “s” (una función que asocia a cada entero un valor de pertenencia en el rango [0, 1]) y devuelve el valor de pertenencia de “elem” en “s”.

1.2 Método: muchoMayorQue

```
11  def muchoMayorQue ( a : Int, m: Int ) : ConjDifuso = {
12      def mma( x :Int) : Double = {
13          if ( x <= a ) 0.0
14          else if ( x > a && x <= m) (x-a).toDouble / (m-a).toDouble
15          else 1.0
16      }
17      mma
18  }
19
```

Define un conjunto difuso que representa la relación de un número “x” con respecto a dos valores enteros “a” y “m”. Devuelve una función ConjDifuso que evalúa si un número es "mucho mayor que" a según un punto de transición m. El valor devuelto varía entre 0.0 y 1.0, donde:

- 0.0: Si “x” es menor o igual a “a”. (línea 13)
- Incrementa linealmente entre “a” y “m”. (línea 14)
- 1.0: Si “x” es mayor que “m”. (línea 15)

1.3 Método: grande

```
21      def grande(d: Int, e: Int): ConjDifuso = {
22          require(d >= 1, "d debe ser mayor o igual a 1")
23          require(e > 1, "e debe ser mayor a 1")
24          (n: Int) => Math.pow(n.toDouble / (n + d), e)
25      }
26
```

Genera un conjunto difuso que evalúa qué tan "grande" es un número “n” con respecto a los parámetros “d” y “e”. La fórmula usada es: (n / (n + d)) ^ e, donde el valor aumenta conforme n crece.

Requisitos: “d” debe ser mayor o igual a 1, y “e” mayor a 1.

1.4 Método: complemento

```
28      def complemento (c : ConjDifuso) : ConjDifuso = {
29          (x: Int) => 1.0 - c(x)
30      }
31
```

Calcula el complemento de un conjunto difuso. Devuelve una función ConjDifuso que para cada valor “x” retorna 1.0 - c(x), invirtiendo el grado de pertenencia del elemento “x” en el conjunto original “c”.

1.5 Método: union

```
32      def union( cd1 : ConjDifuso , cd2 : ConjDifuso ) : ConjDifuso = {
33          (x: Int) => math.max(cd1(x) , cd2(x))
34      }
35
```

Realiza la unión de dos conjuntos difusos “cd1” y “cd2”. Devuelve una función ConjDifuso que para cada “x” toma el máximo valor de pertenencia entre “cd1(x)” y “cd2(x)”, reflejando el grado de pertenencia más alto del elemento “x” a cualquiera de los dos conjuntos.

1.6 Método: intersección

```
36      def interseccion( cd1 : ConjDifuso , cd2 : ConjDifuso ) : ConjDifuso = {
37          (x: Int) => math.min(cd1(x) , cd2(x))
38      }
39      /*
```

Realiza la intersección de dos conjuntos difusos “cd1” y “cd2”. Devuelve una función ConjDifuso que para cada “x” toma el valor mínimo entre “cd1(x)” y “cd2(x)”, indicando el menor grado de pertenencia del elemento “x” en ambos conjuntos.

1.7 Método: inclusión

```
def inclusion ( cd1 : ConjDifuso , cd2 : ConjDifuso ) : Boolean = {
    @annotation.tailrec
    def aux( x : Int ) : Boolean = {
        if ( x > 100 ) true
        else if (cd1.pertenece(x) > cd2.pertenece(x)) false
        else aux(x+1)
    }
    aux(0)
}
```

“aux” es una función auxiliar interna que recibe el índice “x” y verifica si el grado de pertenencia de “x” en cd1 es menor o igual al de cd2. Si encontramos algún elemento donde el grado de pertenencia de “cd1” es mayor que en “cd2”, se retorna **false**. Si llegamos al final del intervalo [0,1000] sin encontrar ninguna violación, se retorna **true**, indicando que “cd1” está incluido en “cd2”. Iniciamos la recursión desde el elemento 0.

1.8 Método: igualdad

```
def igualdad ( cd1 : ConjDifuso , cd2 : ConjDifuso ) : Boolean = {
    inclusion(cd1, cd2) && inclusion(cd2, cd1)
}
```

Se evalúa la igualdad de dos conjuntos difusos “cd1” y “cd2” se verifica si “**cd1** $\subseteq$ **cd2”** y “**cd2** $\subseteq$ **cd1”**. Se retornará entonces **false** o **true** dependiendo del resultado de esta verificación.

2. Informe de Corrección

2.1 Método: pertenece

f: μ $S(x)$ = $[0, 1]$, S es un conjunto difuso, "x" es valor que tiene un grado de pertenencia en ese conjunto.

```
def pertenece ( elem : Int , s : ConjDifuso ) : Double = {
    s(elem)
}
```

P_f:

P_f(Entero, Conjunto Difuso) == f μ($Conjunto\ Difuso$) ($Entero$)

Inducción

P_f(Entero, Conjunto Difuso) == f μ($Conjunto\ Difuso$) ($Entero$)

S (elem) == μ $S(x)$

Double == [0 , 1]

Se puede interpretar que el código representa la pertenencia de conjuntos de una forma casi igual a la matemática, siendo su resultado un Double que, aunque no se especifique en esta parte del código, sólo toma valores decimales entre 1 y 0 incluidos.

2.2 Método: grande

f: $(n/n + d)^e$ = $Conjunto\ de\ numeros\ grandes$, donde n es el número que se quiere saber qué tan grande es, "d" es un número igual o mayor a 1 y "e"

```
def grande(d: Int, e: Int): ConjDifuso = {
    require(d >= 1, "d debe ser mayor o igual a 1")
    require(e > 1, "e debe ser mayor a 1")
    (n: Int) => Math.pow(n.toDouble / (n + d), e)
}
```

P_f:

Se requiere que d sea mayor o igual a 1 y que e sea mayor a 1

P_f (n: Int) => Math.pow(n.toDouble / (n + d),e)

Caso base: n = 0

P_f (n: Int) => Math.pow(0 / (0 + d),e)

P_f (n: Int) => 0

En el caso de 0, devolvería un grado de pertenencia 0 pues no es para nada un número grande.

Caso Inductivo: n > 0

En el caso de que n sea mayor a 0, y los valores de "d" y "e" cumplan las condiciones respectivas, el valor de pertenencia va a ser mayor a 0, pero menor a 1 para todos los casos, puesto que n/(n+d) nunca va a dar 1.

2.3 Método: complemento

f: $f\neg S = 1 - fS$

$(\mu S(x))^c = 1 - \mu S(x)$

```
def complemento (c : ConjDifuso) : ConjDifuso = {
    (x: Int) => 1.0 - c(x)
}
```

P_f:

$P_f(x: Int) => 1.0 - c(x)$ c: conjunto difuso

Inducción:

$P_f == f$

$1.0 - c(x) == 1 - \mu S(x)$

Se puede deducir que cumple con su interpretación matemática, puesto que el método de complemento devuelve un conjunto difuso resultado de la resta entre 1 y el valor de pertenencia del conjunto difuso original.

2.4 Método: unión

f: $f_{S1 \cup S2} = max(f_{S1}, f_{S2})$

```
def union( cd1 : ConjDifuso , cd2 : ConjDifuso ) : ConjDifuso = {
    (x: Int) => math.max(cd1(x) , cd2(x))
}
```

P_f:

$P_f(x: Int) => math.max(cd1(x), cd2(x))$ cd1 y cd2 son conjuntos difusos

Inducción:

$P_f == f$

$f_{S1 \cup S2} == P_f(x: Int)$

$max(f_{S1}, f_{S2}) == math.max(cd1(x), cd2(x))$

Por inducción, podemos deducir que se cumplen las condiciones planteadas por la parte matemática.

La unión de dos conjuntos difusos es el valor máximo de pertenencia de los conjuntos, el método cumple esto al ejecutar la función math.max en los dos conjuntos, escogiendo entre los 2 cual de los dos valores de pertenencia de x en cada conjunto es mayor.

2.5 Método: intersección

f: $f_{S1 \cap S2} = min(f_{S1}, f_{S2})$

```
def interseccion( cd1 : ConjDifuso , cd2 : ConjDifuso ) : ConjDifuso = {
    (x: Int) => math.min(cd1(x) , cd2(x))
}
```

P_f:

$P_f\,(x\colon Int) \;=>\; math.min(cd1(x),\, cd2(x))$ cd1 y cd2 son conjuntos difusos

Inducción:

$P_f == f$

$f_{S1 \cap S2} \;==\; P_f\,(x\colon Int)$

$min(f_{S1}\,,\, f_{S2}) \;==\; math.min(cd1(x),\, cd2(x))$

Por inducción, podemos deducir que se cumplen las condiciones planteadas por la parte matemática.

El razonamiento matemático nos dice que la intersección de 2 conjuntos difusos es igual al menor grado de pertenencia entre los conjuntos difusos originales. El método intersección cumple con esta condición gracias a la función math.min que así como lo especifica el razonamiento matemático selecciona entre los grados de pertenencia de los dos conjuntos el menor.

2.6 Método: inclusión

$f\colon\; \forall s \in U : fS1\,(s) \leq fS2\,(s) \;=\; Boolean\,(true\ or\ false)$

```
def inclusion ( cd1 : ConjDifuso , cd2 : ConjDifuso ) : Boolean = {
    @annotation.tailrec
    def aux( x : Int ) : Boolean = {
        if ( x > 100 ) true
        else if (cd1.pertenece(x) > cd2.pertenece(x)) false
        else aux(x+1)
    }
    aux(0)
}
```

$P_f\colon aux\,\{\,if\ (x \;>\; 100)\ true$

$else\ if\ (cd1.pertenece(x) \;>\; cd2.pertenece(x))\ false$

$else\ aux(x\ +\ 1)\ \}$

Caso base:

x = 0

$if(0\ >\ 100)\ true$ $\quad$ $\#0\ no\ mayor\ que\ 100$

Inducción:

Para x = 0 se tienen 2 posibilidades.

$if\ (cd1.pertenece(0)\ >\ cd2.pertenece(0))\ false$ $\quad$ $\#Caso\ en\ el\ que\ no\ sea\ mayor\ cd1$

$else\ aux(0\ +\ 1)$ $\quad$ $\#Caso\ en\ el\ que\ sea\ menor\ cd1$

En este caso se calcula si el conjunto cd1 está incluido en cd2. Para esto, en todos los casos x posibles, cd1 debe ser menor o igual a cd2. En caso de que cd1 sea mayor ya lo descarta. (devuelve el boolean false)

Pero en caso x = 0; si el grado de pertenencia de cd1 es menor entra en una recursión:

x = 1

$f(1\ >\ 100)\ true$

$if\ (cd1.pertenece(1)\ >\ cd2.pertenece(1))\ false$ $\quad$ $\#Caso\ en\ el\ que\ no\ sea\ mayor\ cd1$

$else\ aux(1\ +\ 1)$ $\quad$ $\#Caso\ en\ el\ que\ sea\ menor\ cd1$

Repitiendo el proceso hasta calcular el grado de pertenencia de la mayoría de valores de x para los dos conjuntos (en este caso 100, no se necesita tanto para trabajar un programa simple). Si el grado de pertenencia de cd1 es menor al de cd2 para todos los casos retorna el boolean true.

2.7 Método: igualdad

$f\colon S1\ =\ S2,\ \ si\ \ S1 \subseteq S2 \wedge S2 \subseteq S1\ \ =\ Boolean\ (true\ or\ false)$

```
def igualdad ( cd1 : ConjDifuso , cd2 : ConjDifuso ) : Boolean = {
    inclusion(cd1, cd2) && inclusion(cd2, cd1)
}
```

P_f: $inclusion(cd1,\ cd2)$ && $inclusion(cd2, cd1)$

Caso inductivo:

$f\ ==\ P_f$

$S1 \subseteq S2 \wedge S2 \subseteq S1\ ==\ inclusion(cd1,\ cd2)$ && $inclusion(cd2, cd1)$

Para este caso, requerimos del método anterior pues este nos permite calcular la inclusión de un conjunto en otro. Para el caso de la igualdad, dos conjuntos difusos son iguales si $S1 \subseteq S2 \wedge S2 \subseteq S1$, para la parte de código es cambiar la semántica.

Podemos ver que ambas funciones son iguales ya que ambas calculan la igualdad de dos conjuntos difusos a partir del operador "&&" (AND) en la inclusión de un conjunto en otro y viceversa.

3. Conclusiones

3.1 Sobre abstracción y reusabilidad de las funciones de alto orden

En el desarrollo de este taller, el uso de funciones de alto orden resultó esencial para abstraer la lógica de los conjuntos difusos de una manera que permite construir operaciones complejas y reusables, como el complemento, la unión y la intersección. Las funciones de alto orden, al aceptar otras funciones como parámetros o retornar funciones, facilitan la creación de operaciones flexibles y genéricas, adaptables a diversas necesidades.

Esta capacidad para generalizar y modularizar las operaciones sobre conjuntos difusos enriquece el diseño del código y reduce la repetición de lógica, haciendo que cada función individual sea más comprensible y reusable en otros contextos. Así, el taller demostró cómo la programación funcional promueve la construcción de software modular y adaptable, características fundamentales en sistemas complejos y en áreas donde la precisión matemática, como en el manejo de conjuntos difusos, es crítica.

3.2 Sobre la aplicabilidad de la P.F en el manejo de sistemas complejos.

La programación funcional, aplicada en este taller al contexto de conjuntos difusos, demostró su utilidad para resolver problemas que requieren operaciones matemáticas complejas y precisas. En el ámbito industrial, especialmente en

áreas como la inteligencia artificial, el procesamiento de datos y la ingeniería de sistemas, la capacidad de construir sistemas fiables y eficientes es clave. La programación funcional permite desarrollar aplicaciones que son más fáciles de probar, optimizar y mantener, lo cual reduce los riesgos asociados con sistemas complejos. Al implementar conceptos de programación funcional como funciones de alto orden en problemas matemáticos, en este taller se nos brinda una visión práctica de cómo las técnicas de P.F pueden aportar soluciones confiables y escalables en un contexto real.

3.3 Sobre pruebas matemáticas y de software

Este proceso de tests de software y pruebas a través de las matemáticas discretas es particularmente relevante en programación funcional, ya que la composición de funciones de alto orden puede generar lógicas complejas que deben ser rigurosamente verificadas para evitar errores. Además, aplicar pruebas específicas para cada función fomenta una cultura de desarrollo confiable y estructurado, especialmente en áreas que requieren precisión, como los sistemas de lógica difusa.

Así, el taller ilustra cómo la programación funcional, al ser fácilmente verificable mediante pruebas matemáticas y de software, refuerza la creación de software fiable y eficiente.