Programación Funcional y Concurrente

Alejandra Osorio Giraldo - 2266128
Luis Manuel Cardona Trochez - 2059942
Jose David Marmol Otero - 2266370

Carlos Andres Delgado Saavedra

Universidad del Valle
Sede Tuluá
2024

INFORME DE PROCESO

En este informe se presenta la implementación y análisis de varias funciones fundamentales para trabajar con conjuntos difusos en Scala, como la funciones que veremos a continuación, grande, complemento, unión, intersección, inclusión e igualdad.

1) **Def grande**

```scala
def grande(d: Int, e: Int): ConjuntoDifuso = {
    def auxGrande(x: Int): Double = {
        math.pow(x.toDouble / (x + d).toDouble, e.toDouble)
    }
    auxGrande
}
```

TEST:

```scala
24      test("Pertenece 1 en grande(2, 2)") {
25      assert(c.Pertenece(1, c.grande(2, 2)) == math.pow(1.0 / (1 + 2), 2)) // (1 / 3)^2
26      }
27
28      test("Pertenece 2 en grande(2, 2)") {
29          assert(c.Pertenece(2, c.grande(2, 2)) == math.pow(2.0 / (2 + 2), 2)) // (2 / 4)^2
30      }
31
32      test("Pertenece 3 en grande(2, 3)") {
33          assert(c.Pertenece(3, c.grande(2, 3)) == math.pow(3.0 / (3 + 2), 3)) // (3 / 5)^3
34      }
35
36      test("Pertenece 4 en grande(3, 1)") {
37          assert(c.Pertenece(4, c.grande(3, 1)) == math.pow(4.0 / (4 + 3), 1)) // 4 / 7
38      }
39
40      test("Pertenece 5 en grande(4, 2)") {
41          assert(c.Pertenece(5, c.grande(4, 2)) == math.pow(5.0 / (5 + 4), 2)) // (5 / 9)^2
42      }
```

PASSED:

```
TestConjuntoDifusos:
- Pertenece 1 en grande(2, 2)
- Pertenece 2 en grande(2, 2)
- Pertenece 3 en grande(2, 3)
- Pertenece 4 en grande(3, 1)
- Pertenece 5 en grande(4, 2)
Execution took 59ms
5 tests, 5 passed
All tests in taller.TestConjuntoDifusos passed
```

2) **Def complemento:**

```
def complemento(s: ConjuntoDifuso): ConjuntoDifuso = {
    def Auxcomp(x: Int): Double = {
        1.0 - s(x)
    }
    Auxcomp
}
```

TEST:

```
val conjuntoEjemplo = c.ConjuntoDifuso(x => if (x <= 5) 0.8 else 0.3)
val complementoEjemplo = c.complemento(conjuntoEjemplo)

test("Pertenece 3 en complemento de conjuntoEjemplo") {
assert(c.Pertenece(3, complementoEjemplo) == 1.0 - 0.8) // Debe devolver 0.2
}

test("Pertenece 6 en complemento de conjuntoEjemplo") {
    assert(c.Pertenece(6, complementoEjemplo) == 1.0 - 0.3) // Debe devolver 0.7
}

test("Pertenece 5 en complemento de conjuntoEjemplo") {
    assert(c.Pertenece(5, complementoEjemplo) == 1.0 - 0.8) // Debe devolver 0.2
}

test("Pertenece 0 en complemento de conjuntoEjemplo") {
    assert(c.Pertenece(0, complementoEjemplo) == 1.0 - 0.8) // Debe devolver 0.2
}
```

PASSED:

```
TestConjuntoDifusos:
- Pertenece 3 en complemento de conjuntoEjemplo
- Pertenece 6 en complemento de conjuntoEjemplo
- Pertenece 5 en complemento de conjuntoEjemplo
- Pertenece 0 en complemento de conjuntoEjemplo
- Pertenece 10 en complemento de conjuntoEjemplo
Execution took 27ms
5 tests, 5 passed
All tests in taller.TestConjuntoDifusos passed
```

3) **Def unión**

```
def Union(cd1: ConjuntoDifuso, cd2: ConjuntoDifuso): ConjuntoDifuso = {
    def auxUnion(x: Int): Double = {
        math.max(cd1(x), cd2(x))
    }
    auxUnion
}
```

TEST:

```
71      val conjuntoDifuso1 = c.ConjuntoDifuso(x => if (x <= 3) 0.4 else 0.7)
72      val conjuntoDifuso2 = c.ConjuntoDifuso(x => if (x <= 3) 0.8 else 0.2)
73      val unionConjuntos = c.Union(conjuntoDifuso1, conjuntoDifuso2)
74
75      test("Pertenece 2 en union de conjuntoDifuso1 y conjuntoDifuso2") {
76          assert(c.Pertenece(2, unionConjuntos) == math.max(0.4, 0.8)) // Debe devolver 0.8
77      }
78
79      test("Pertenece 4 en union de conjuntoDifuso1 y conjuntoDifuso2") {
80          assert(c.Pertenece(4, unionConjuntos) == math.max(0.7, 0.2)) // Debe devolver 0.7
81      }
82
83      test("Pertenece 3 en union de conjuntoDifuso1 y conjuntoDifuso2") {
84          assert(c.Pertenece(3, unionConjuntos) == math.max(0.4, 0.8)) // Debe devolver 0.8
85      }
86
87      test("Pertenece 1 en union de conjuntoDifuso1 y conjuntoDifuso2") {
88          assert(c.Pertenece(1, unionConjuntos) == math.max(0.4, 0.8)) // Debe devolver 0.8
89      }
```

PASSED:

```
TestConjuntoDifusos:
- Pertenece 2 en union de conjuntoDifuso1 y conjuntoDifuso2
- Pertenece 4 en union de conjuntoDifuso1 y conjuntoDifuso2
- Pertenece 3 en union de conjuntoDifuso1 y conjuntoDifuso2
- Pertenece 1 en union de conjuntoDifuso1 y conjuntoDifuso2
- Pertenece 5 en union de conjuntoDifuso1 y conjuntoDifuso2
Execution took 25ms
5 tests, 5 passed
All tests in taller.TestConjuntoDifusos passed
```

4) **Def union**

```
def Union(cd1: ConjuntoDifuso, cd2: ConjuntoDifuso): ConjuntoDifuso = {
    def auxUnion(x: Int): Double = {
        math.max(cd1(x), cd2(x))
    }
    auxUnion
}
```

TEST:

```
val conjuntoDifuso3 = c.ConjuntoDifuso(x => if (x % 2 == 0) 0.5 else 0.3)
val conjuntoDifuso4 = c.ConjuntoDifuso(x => if (x > 5) 0.6 else 0.4)
val unionConjuntos2 = c.Union(conjuntoDifuso3, conjuntoDifuso4)

test("Pertenece 2 en union de conjuntoDifuso3 y conjuntoDifuso4") {
    assert(c.Pertenece(2, unionConjuntos2) == math.max(0.5, 0.4)) // Debe devolver 0.5
}

test("Pertenece 7 en union de conjuntoDifuso3 y conjuntoDifuso4") {
    assert(c.Pertenece(7, unionConjuntos2) == math.max(0.3, 0.6)) // Debe devolver 0.6
}

test("Pertenece 4 en union de conjuntoDifuso3 y conjuntoDifuso4") {
    assert(c.Pertenece(4, unionConjuntos2) == math.max(0.5, 0.4)) // Debe devolver 0.5
}

test("Pertenece 6 en union de conjuntoDifuso3 y conjuntoDifuso4") {
    assert(c.Pertenece(6, unionConjuntos2) == math.max(0.5, 0.6)) // Debe devolver 0.6
```

PASSED:

```
TestConjuntoDifusos:
- Pertenece 2 en union de conjuntoDifuso3 y conjuntoDifuso4
- Pertenece 7 en union de conjuntoDifuso3 y conjuntoDifuso4
- Pertenece 4 en union de conjuntoDifuso3 y conjuntoDifuso4
- Pertenece 6 en union de conjuntoDifuso3 y conjuntoDifuso4
- Pertenece 5 en union de conjuntoDifuso3 y conjuntoDifuso4
Execution took 31ms
5 tests, 5 passed
All tests in taller.TestConjuntoDifusos passed
```

5) **Def interseccion**

```
def Interseccion(cd1: ConjuntoDifuso, cd2: ConjuntoDifuso): ConjuntoDifuso = {
    def auxInterseccion(x: Int): Double = {
        math.min(cd1(x), cd2(x))
    }
    auxInterseccion
}
```

TEST:

```
125     val conjuntoDifuso5 = c.ConjuntoDifuso(x => if (x <= 4) 0.9 else 0.2)
126     val conjuntoDifuso6 = c.ConjuntoDifuso(x => if (x % 2 == 0) 0.7 else 0.4)
127     val interseccionConjuntos = c.Interseccion(conjuntoDifuso5, conjuntoDifuso6)
128
129     test("Pertenece 3 en interseccion de conjuntoDifuso5 y conjuntoDifuso6") {
130         assert(c.Pertenece(3, interseccionConjuntos) == math.min(0.9, 0.4)) // Debe devolver 0.4
131     }
132
133     test("Pertenece 2 en interseccion de conjuntoDifuso5 y conjuntoDifuso6") {
134         assert(c.Pertenece(2, interseccionConjuntos) == math.min(0.9, 0.7)) // Debe devolver 0.7
135     }
136
137     test("Pertenece 5 en interseccion de conjuntoDifuso5 y conjuntoDifuso6") {
138         assert(c.Pertenece(5, interseccionConjuntos) == math.min(0.2, 0.4)) // Debe devolver 0.2
139     }
140
141     test("Pertenece 6 en interseccion de conjuntoDifuso5 y conjuntoDifuso6") {
142         assert(c.Pertenece(6, interseccionConjuntos) == math.min(0.2, 0.7)) // Debe devolver 0.2
143     }
```

PASSED:

```
TestConjuntoDifusos:
- Pertenece 3 en interseccion de conjuntoDifuso5 y conjuntoDifuso6
- Pertenece 2 en interseccion de conjuntoDifuso5 y conjuntoDifuso6
- Pertenece 5 en interseccion de conjuntoDifuso5 y conjuntoDifuso6
- Pertenece 6 en interseccion de conjuntoDifuso5 y conjuntoDifuso6
- Pertenece 4 en interseccion de conjuntoDifuso5 y conjuntoDifuso6
Execution took 25ms
5 tests, 5 passed
All tests in taller.TestConjuntoDifusos passed
```

6) **Def inclusion**

```
def Inclusion (cd1: ConjuntoDifuso, cd2: ConjuntoDifuso): Boolean = {
    def AuxInclusion(x: Int): Boolean = {
        if (x>1000) true
        else if (cd1(x) > cd2(x)) false//0,2 esta incluido 0,199999
        else AuxInclusion(x+1)

    }
    AuxInclusion(0)//hasta que mi iterador llegue a 1001
}
```

TEST:

```
150     val conjuntoDifuso7 = c.ConjuntoDifuso(x => if (x <= 500) 0.2 else 0.5)
151     val conjuntoDifuso8 = c.ConjuntoDifuso(x => if (x <= 500) 0.3 else 0.5)
152     val conjuntoDifuso9 = c.ConjuntoDifuso(x => if (x <= 300) 0.6 else 0.3)
153
154     // Pruebas para Inclusion
155     test("Inclusion de conjuntoDifuso7 en conjuntoDifuso8") {
156         assert(c.Inclusion(conjuntoDifuso7, conjuntoDifuso8) == true) // Debe devolver true
157     }
158
159     test("Inclusion de conjuntoDifuso8 en conjuntoDifuso7") {
160         assert(c.Inclusion(conjuntoDifuso8, conjuntoDifuso7) == false) // Debe devolver false
161     }
162
163     test("Inclusion de conjuntoDifuso9 en conjuntoDifuso8") {
164         assert(c.Inclusion(conjuntoDifuso9, conjuntoDifuso8) == false) // Debe devolver false
165     }
```

PASSED:

```
TestConjuntoDifusos:
- Inclusion de conjuntoDifuso7 en conjuntoDifuso8
- Inclusion de conjuntoDifuso8 en conjuntoDifuso7
- Inclusion de conjuntoDifuso9 en conjuntoDifuso8
- Inclusion de conjuntoDifuso7 en conjuntoDifuso9
- Inclusion de conjuntoDifuso8 en sí mismo
Execution took 33ms
5 tests, 5 passed
All tests in taller.TestConjuntoDifusos passed
```

7) **Def igualdad**

```
def Igualdad(cd1: ConjuntoDifuso, cd2: ConjuntoDifuso): Boolean = {
    def AuxIgualdad(x: Int): Boolean = {
        if (x>1000) true
        else if (cd1(x) != cd2(x)) false
        else AuxIgualdad(x+1)
    }
    AuxIgualdad(0)
}
```

TEST:

```
176     test("Igualdad entre conjuntoDifuso7 y conjuntoDifuso7") {
177     assert(c.Igualdad(conjuntoDifuso7, conjuntoDifuso7) == true) // Debe devolver true
178     }
179
180     test("Igualdad entre conjuntoDifuso7 y conjuntoDifuso8") {
181         assert(c.Igualdad(conjuntoDifuso7, conjuntoDifuso8) == false) // Debe devolver false
182     }
183
184     test("Igualdad entre conjuntoDifuso8 y conjuntoDifuso9") {
185         assert(c.Igualdad(conjuntoDifuso8, conjuntoDifuso9) == false) // Debe devolver false
186     }
187
188     test("Igualdad entre dos funciones idénticas") {
189         val conjuntoDifuso10 = c.ConjuntoDifuso(x => 0.5)
190         val conjuntoDifuso11 = c.ConjuntoDifuso(x => 0.5)
191         assert(c.Igualdad(conjuntoDifuso10, conjuntoDifuso11) == true) // Debe devolver true
192     }
```

PASSED:

```
TestConjuntoDifusos:
- Igualdad entre conjuntoDifuso7 y conjuntoDifuso7
- Igualdad entre conjuntoDifuso7 y conjuntoDifuso8
- Igualdad entre conjuntoDifuso8 y conjuntoDifuso9
- Igualdad entre dos funciones idénticas
- Igualdad entre conjuntos difusos con patrones distintos
Execution took 32ms
5 tests, 5 passed
All tests in taller.TestConjuntoDifusos passed
```

WILL BE DELETED

Tests

```
4   class TestConjuntoDifusos extends AnyFunSuite {
5       val c = new ConjDifusos()
6
7       test("Pertenece 3 en muchoMayorQue(3, 7)"){
8           assert(c.Pertenece(3, c.muchoMayorQue(3, 7)) == 0.0)
9       }
10
11      test("Pertenece 5 en muchoMayorQue(1, 5)") {
12          assert(c.Pertenece(5, c.muchoMayorQue(1, 5)) == 1.0)
13      }
14
15      test("Pertenece 0 en muchoMayorQue(1, 5)") {
16          assert(c.Pertenece(0, c.muchoMayorQue(1, 5)) == 0.0)
17      }
18
19      test("Pertenece 7 en muchoMayorQue(3, 7)") {
20          assert(c.Pertenece(7, c.muchoMayorQue(3, 7)) == 1.0)
21      }
22  }
```

PASSED:

```
TestConjuntoDifusos:
- Pertenece 3 en muchoMayorQue(3, 7)
- Pertenece 5 en muchoMayorQue(1, 5)
- Pertenece 0 en muchoMayorQue(1, 5)
- Pertenece 7 en muchoMayorQue(3, 7)
Execution took 45ms
4 tests, 4 passed
All tests in taller.TestConjuntoDifusos passed

===============================================
Total duration: 45ms
All 1 test suites passed.
```

INFORME DE CORRECCIÓN

Def grande:

```
def grande(d: Int, e: Int): ConjuntoDifuso = {
    def auxGrande(x: Int): Double = {
        math.pow(x.toDouble / (x + d).toDouble, e.toDouble)
    }
    auxGrande
}
```

La función **grande** determina el grado de pertenencia de un valor en un conjunto trapezoidal. Si el valor está dentro del intervalo la función aplica una fórmula matemática para calcular el grado de pertenencia. Si el valor está fuera del intervalo, devuelve 0. La implementación sigue bien la lógica de la función grande en conjuntos trapezoidales.

Def complemento:

```
def complemento(s: ConjuntoDifuso): ConjuntoDifuso = {
    def Auxcomp(x: Int): Double = {
        1.0 - s(x)
    }
    Auxcomp
}
```

La función **complemento** calcula el complemento del grado de pertenencia, lo que se hace restando el grado de pertenencia de 1 (es decir, 1 - grado_pertenencia). Esta función se implementa bien, ya que sigue la definición estándar de complemento en lógica difusa.

Def union:

```
def Union(cd1: ConjuntoDifuso, cd2: ConjuntoDifuso): ConjuntoDifuso = {
    def auxUnion(x: Int): Double = {
        math.max(cd1(x), cd2(x))
    }
    auxUnion
}
```

La función **unión** calcula el grado de pertenencia de la unión entre dos conjuntos difusos. La unión se determina tomando el máximo entre los grados de pertenencia de cada conjunto para un mismo valor. Esto está bien porque según la teoría de conjuntos difusos, la unión de dos conjuntos se define tomando el máximo entre los grados de pertenencia correspondientes.

Def interseccion:

```
def Interseccion(cd1: ConjuntoDifuso, cd2: ConjuntoDifuso): ConjuntoDifuso = {
    def auxInterseccion(x: Int): Double = {
        math.min(cd1(x), cd2(x))
    }
    auxInterseccion
}
```

La función **intersección** devuelve el grado de pertenencia de la intersección entre dos conjuntos difusos. La intersección se calcula tomando el mínimo entre los grados de pertenencia de los dos conjuntos. Está bien implementado porque la intersección de conjuntos difusos se define como el mínimo de los grados de pertenencia correspondientes.

Def inclusion:

```
def Inclusion (cd1: ConjuntoDifuso, cd2: ConjuntoDifuso): Boolean = {
    def AuxInclusion(x: Int): Boolean = {
        if (x>1000) true
        else if (cd1(x) > cd2(x)) false//0,2 esta incluido 0,199999
        else AuxInclusion(x+1)

    }
    AuxInclusion(0)//hasta que mi iterador llegue a 1001
}
```

La función **inclusión** verifica si un conjunto está incluido dentro de otro. Para eso se compara los grados de pertenencia de cada valor en ambos conjuntos. Si todos los grados del primer conjunto son menores o iguales que los del segundo, se considera que el primer conjunto está incluido en el segundo. La implementación es correcta

porque compara correctamente los grados de pertenencia de cada valor en ambos conjuntos.

Def igualdad:

```
def Igualdad(cd1: ConjuntoDifuso, cd2: ConjuntoDifuso): Boolean = {
    def AuxIgualdad(x: Int): Boolean = {
        if (x>1000) true
        else if (cd1(x) != cd2(x)) false
        else AuxIgualdad(x+1)
    }
    AuxIgualdad(0)
}
```

La función **igualdad** verifica si dos conjuntos difusos son iguales, comparando sus grados de pertenencia. Si todos los grados de pertenencia de un conjunto son iguales a los grados del otro conjunto, la función devuelve true indicando que los conjuntos son iguales. La implementación es correcta porque sigue la definición estándar de igualdad en conjuntos difusos.

Conclusión

El desarrollo de esta implementación para conjuntos difusos en Scala nos permitió explorar conceptos clave como la pertenencia parcial, la combinación de conjuntos mediante operaciones como unión e intersección, y la evaluación de igualdad e inclusión. Las funciones como grande, complemento, demuestran que se pueden modelar distintas características difusas utilizando funciones matemáticas. También usando los test para verificar la validez o facilitando las correcciones, lo que al final nos deja herramientas para manejar los conjuntos difusos y se resalta lo ideal del lenguaje para implementar soluciones de este tipo.