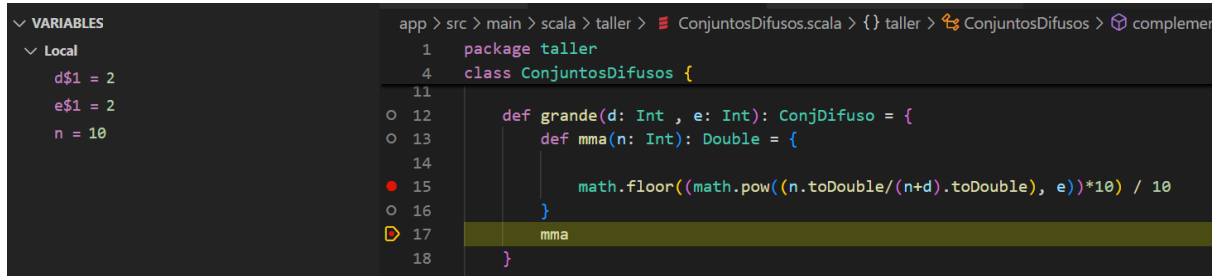


4. cuarto paso

Ya que se tiene en parámetro $n = 10$, vuelve a invocar la función que evaluará si este es un número grande, con la operación:

$\left(\frac{n}{n+d}\right)^e \Rightarrow$ Dicho valor es redondeado con `math.floor` para que quede con su primera cifra decimal.

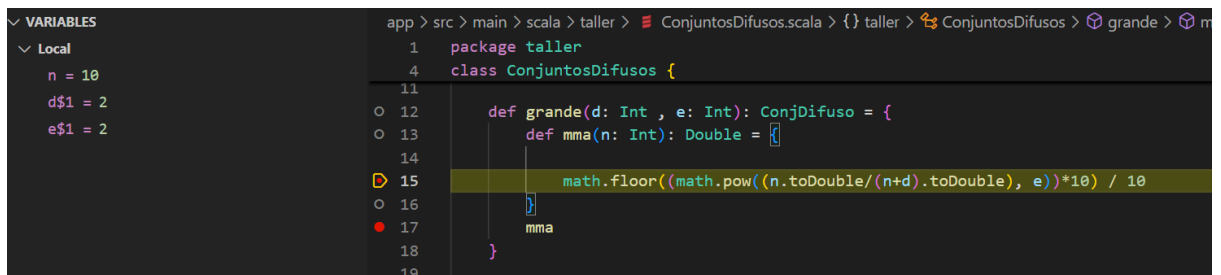


```
app > src > main > scala > taller > ConjuntosDifusos.scala > {} taller > ConjuntosDifusos > complemen

1 package taller
4 class ConjuntosDifusos {
11
12     def grande(d: Int , e: Int): ConjDifuso = {
13         def mma(n: Int): Double = {
14
15             math.floor((math.pow((n.toDouble/(n+d).toDouble), e))*10) / 10
16         }
17         mma
18     }
19 }
```

5. Quinto paso

Ahora evalúa los parámetros en la operación y retorna el resultado obtenido. Para este caso será 0.6.



```
app > src > main > scala > taller > ConjuntosDifusos.scala > {} taller > ConjuntosDifusos > grande > m

1 package taller
4 class ConjuntosDifusos {
11
12     def grande(d: Int , e: Int): ConjDifuso = {
13         def mma(n: Int): Double = {
14
15             math.floor((math.pow((n.toDouble/(n+d).toDouble), e))*10) / 10
16         }
17         mma
18     }
19 }
```

• Función *grande*

Esta función devuelve un conjunto difuso, es decir, una función que define el grado de pertenencia, así que, no retornará un dato explícito como tal, sino, que por medio de la currificación, tomará el dato n en dado caso y hará la operación correspondiente, como en el ejemplo anterior.

```
def grande(d: Int , e: Int): ConjDifuso = {
    def mma(n: Int): Double = {

math.floor((math.pow((n.toDouble/(n+d).toDouble), e))*10) /
10
    }
    mma
}
```

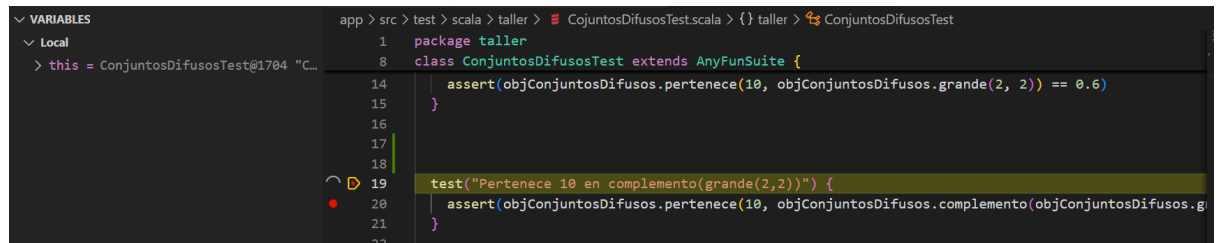
• Función *complemento*

Como su nombre lo indica, esta función calcula el complemento de un conjunto difuso dado, que es lo mismo que el grado de 'no pertenencia', o mejor dicho, $1 - f_s$, siendo f_s la función característica del conjunto.

Si un elemento tiene un grado de pertenencia 0.8 su complemento sería 0.2.

1. Primer paso

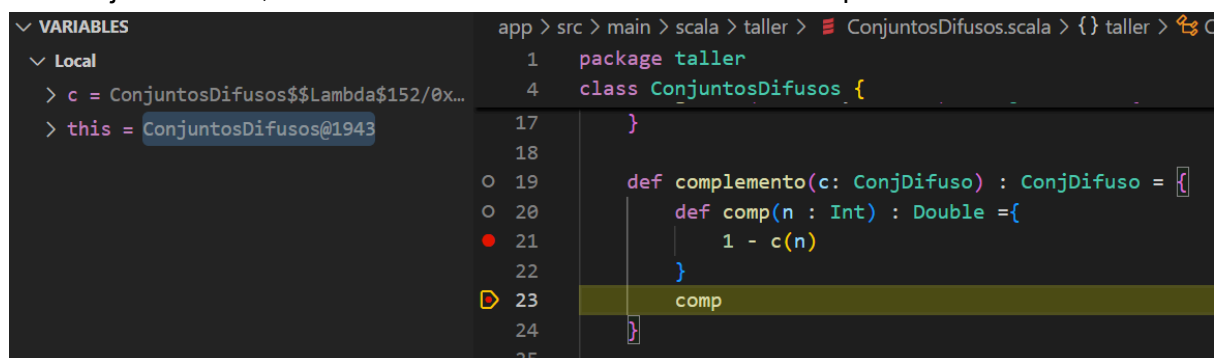
En este caso, se hallará el complemento del conjunto difuso 'grande(2,2)'.



```
app > src > test > scala > taller > ConjuntosDifusosTest.scala > {} taller > ConjuntosDifusosTest
1 package taller
8 class ConjuntosDifusosTest extends AnyFunSuite {
14   assert(objConjuntosDifusos.pertenece(10, objConjuntosDifusos.grande(2, 2)) == 0.6)
15 }
16
17
18
19 test("Pertenece 10 en complemento(grande(2,2))") {
20   assert(objConjuntosDifusos.pertenece(10, objConjuntosDifusos.complemento(objConjuntosDifusos.grande(2, 2))) == 0.6)
21 }
22
```

2. Segundo paso

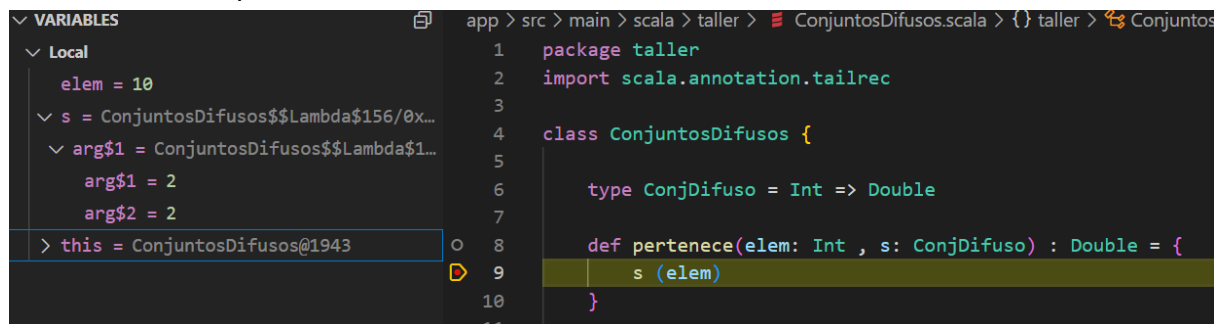
Entra en la función complemento, con c como conjunto difuso, está nos retornará otro conjunto difuso, e invocará la función auxiliar dentro de complemento.



```
app > src > main > scala > taller > ConjuntosDifusos.scala > {} taller > ConjuntosDifusos
1 package taller
4 class ConjuntosDifusos {
17 }
18
19 def complemento(c: ConjDifuso) : ConjDifuso = {
20   def comp(n : Int) : Double = {
21     1 - c(n)
22   }
23   comp
24 }
25
```

3. Tercer paso

Para este paso, el elemento será 10, e invocará la función del conjunto difuso con este valor, dando paso a la curificación de la misma.



```
app > src > main > scala > taller > ConjuntosDifusos.scala > {} taller > ConjuntosDifusos
1 package taller
2 import scala.annotation.tailrec
3
4 class ConjuntosDifusos {
5
6   type ConjDifuso = Int => Double
7
8   def pertenece(elem: Int , s: ConjDifuso) : Double = {
9     s(elem)
10   }
11
```

4. Cuarto paso

Para este paso ya tendrá todos los valores necesarios en la operación de la función comp.



```
app > src > main > scala > taller > ConjuntosDifusos.scala > {} taller > ConjuntosDifusos
1 package taller
4 class ConjuntosDifusos {
17 }
18
19 def complemento(c: ConjDifuso) : ConjDifuso = {
20   def comp(n : Int) : Double = {
21     1 - c(n)
22   }
23   comp
24 }
25
```

5. Quinto paso

Evalúa ahora los parámetros en la función, dando como resultado el complemento del conjunto.

```
app > src > main > scala > taller > ConjuntosDifusos.scala > {} taller > Co
1  package taller
4  class ConjuntosDifusos {
17  }
18
19  def complemento(c: ConjDifuso) : ConjDifuso = {
20      def comp(n : Int) : Double = {
21          1 - c(n)
22      }
23      comp
24  }
25
```

• Función *unión*

La función unión toma como entrada dos conjuntos difusos, cd_1 y cd_2 , y devuelve un nuevo conjunto difuso que representa la unión de ambos. La unión de dos conjuntos difusos se define como el máximo de los grados de pertenencia de cada conjunto para cada elemento del universo.

$def union(cd1: ConjDifuso, cd2: ConjDifuso): ConjDifuso$

cd_1 : Un conjunto difuso de tipo $ConjDifuso$.

cd_2 : Otro conjunto difuso de tipo $ConjDifuso$.

Valor de retorno:

Devuelve un nuevo conjunto difuso que representa la unión de cd_1 y cd_2 . Para cada elemento n , el grado de pertenencia en la unión se calcula como:

$$fs_1 \cup fs_2(n) = \max(fs_1(n), fs_2(n))$$

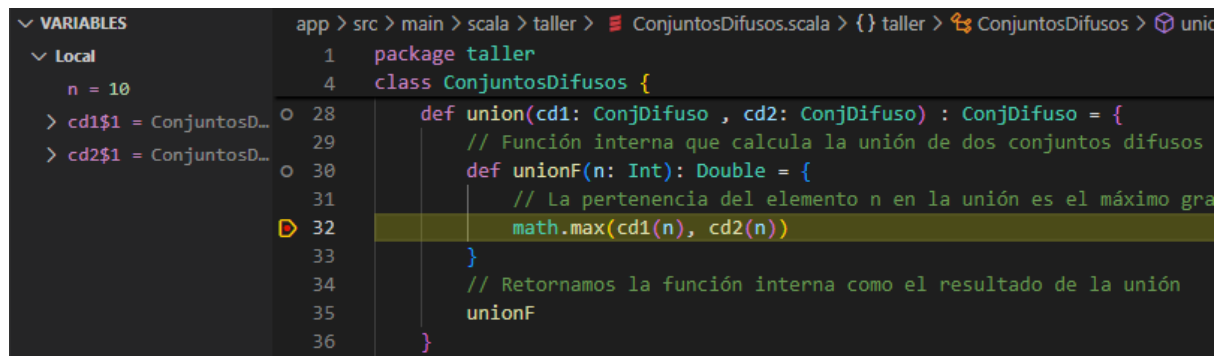
1. Primer paso

```
app > src > test > scala > taller > ConjuntosDifusosTest.scala > {} taller > ConjuntosDifusosTest
1  package taller
8  class ConjuntosDifusosTest extends AnyFunSuite {
38  test("Union grande(1,3) y grande(3,6)") {
39      val union = objConjuntosDifusos.union(objConjuntosDifusos.grande(1, 3), objConjuntosDifusos.grande(3, 6))
40      assert(union(10) == 0.7)
41  }
42  }

app > src > test > scala > taller > ConjuntosDifusosTest.scala > {} taller > ConjuntosDifusosTest
1  package taller
8  class ConjuntosDifusosTest extends AnyFunSuite {
38  test("Union grande(1,3) y grande(3,6)") {
39      val union = objConjuntosDifusos.union(objConjuntosDifusos.grande(1, 3), objConjuntosDifusos.grande(3, 6))
40      assert(union(10) == 0.7)
41  }
42  }
```

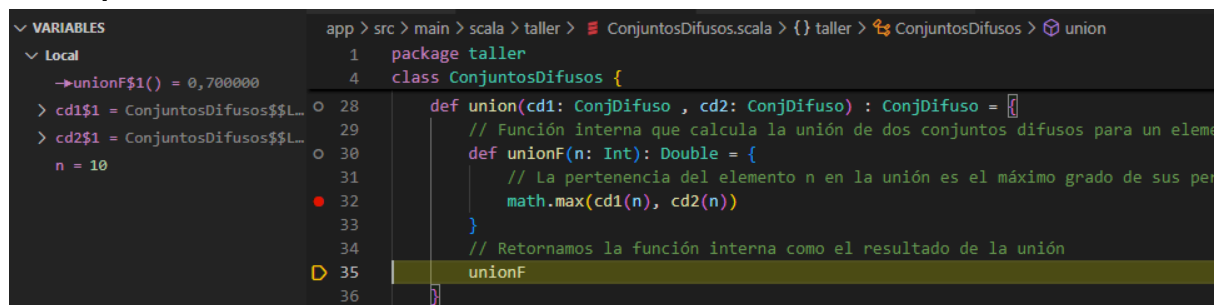
Se ejecuta el test. En este caso los conjuntos difusos son $grande(1,3)$ y $grande(3,6)$ y el caso a evaluar es $n=10$ y debe retornar 0,7.

2. Segundo paso



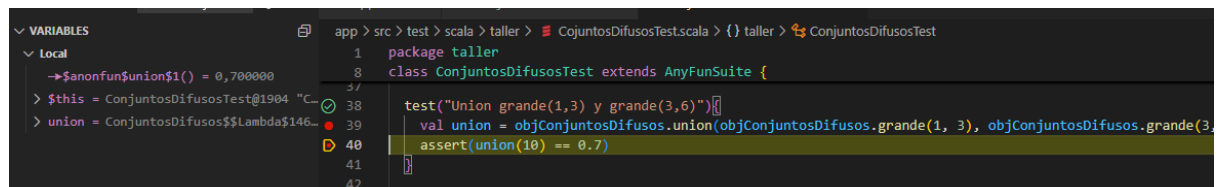
Se entra directamente a la función unionF

3. Tercer paso



Se retorna la función interna unionF que sería el resultado de la unión de los conjuntos difusos. En este caso retornó 0,700000.

4. Cuarto paso



Se pasa a verificar si el resultado dado por la función unión coincide con el test, y como es correcto finaliza la ejecución.

• Función *intersección*

La función intersección toma como entrada dos conjuntos difusos, cd1 y cd2, y devuelve un nuevo conjunto difuso que representa la intersección de ambos. La intersección de dos conjuntos difusos se define como el mínimo de los grados de pertenencia de cada conjunto para cada elemento del universo.

def interseccion(cd1: ConjDifuso, cd2: ConjDifuso): ConjDifuso

cd1: Un conjunto difuso de tipo ConjDifuso.

cd2: Otro conjunto difuso de tipo ConjDifuso.

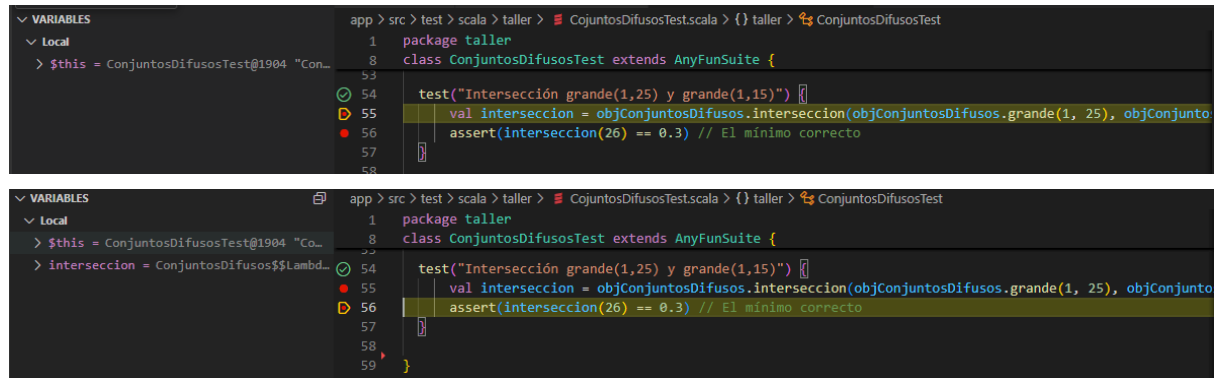
Valor de retorno:

Devuelve un nuevo conjunto difuso que representa la intersección de cd_1 y cd_2 .

Para cada elemento n , el grado de pertenencia en la intersección se calcula como:

$$fs_1 \cap fs_2(n) = \min(fs_1(n), fs_2(n))$$

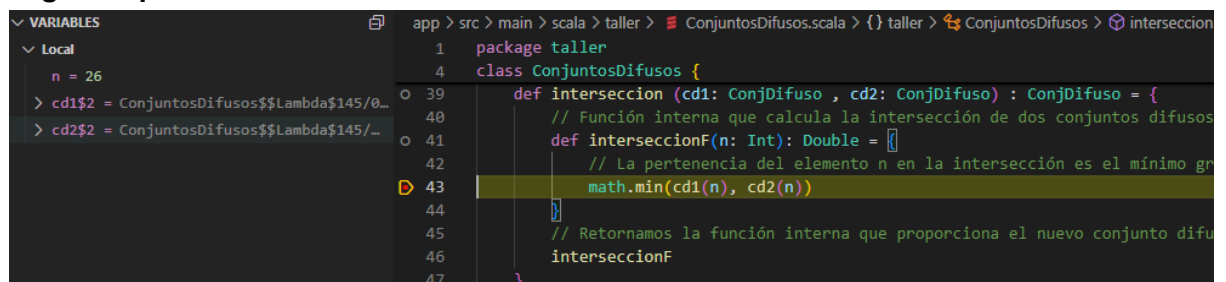
1. Primer paso



```
app > src > test > scala > taller > CojuntosDifusosTest.scala > {} taller > CojuntosDifusosTest
1 package taller
8 class CojuntosDifusosTest extends AnyFunSuite {
53
54 test("Intersección grande(1,25) y grande(1,15)") {
55   val interseccion = objCojuntosDifusos.interseccion(objCojuntosDifusos.grande(1, 25), objCojuntosDifusos.grande(1, 15))
56   assert(interseccion(26) == 0.3) // El mínimo correcto
57 }
58
59 }
```

En este caso los conjuntos difusos son $grande(1,25)$ y $grande(1,15)$ y el caso a evaluar es $n=26$ y debe retornar 0,3.

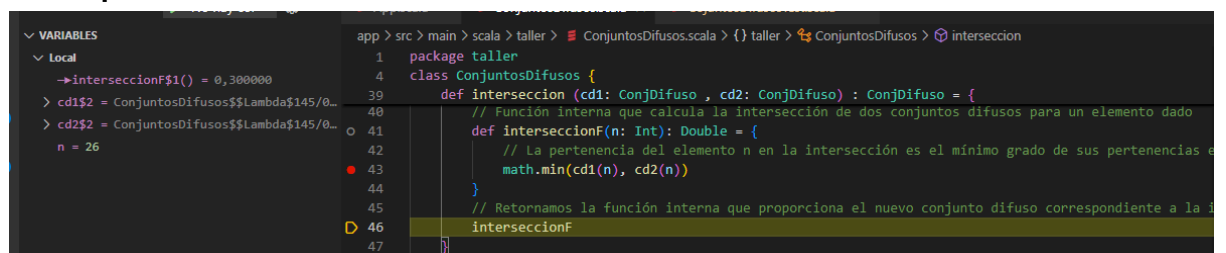
2. Segundo paso



```
app > src > main > scala > taller > CojuntosDifusos.scala > {} taller > CojuntosDifusos > interseccion
1 package taller
4 class CojuntosDifusos {
39   def interseccion (cd1: ConjDifuso , cd2: ConjDifuso) : ConjDifuso = {
40     // Función interna que calcula la intersección de dos conjuntos difusos
41     def interseccionF(n: Int): Double = {
42       // La pertenencia del elemento n en la intersección es el mínimo gr
43       math.min(cd1(n), cd2(n))
44     }
45     // Retornamos la función interna que proporciona el nuevo conjunto difu
46     interseccionF
47 }
```

Se entra directamente a la función `interseccionF`

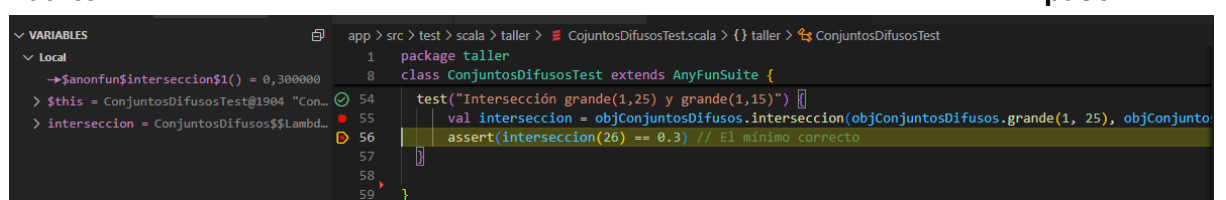
3. Tercer paso



```
app > src > main > scala > taller > CojuntosDifusos.scala > {} taller > CojuntosDifusos > interseccion
1 package taller
4 class CojuntosDifusos {
39   def interseccion (cd1: ConjDifuso , cd2: ConjDifuso) : ConjDifuso = {
40     // Función interna que calcula la intersección de dos conjuntos difusos para un elemento dado
41     def interseccionF(n: Int): Double = {
42       // La pertenencia del elemento n en la intersección es el mínimo grado de sus pertenencias e
43       math.min(cd1(n), cd2(n))
44     }
45     // Retornamos la función interna que proporciona el nuevo conjunto difuso correspondiente a la i
46     interseccionF
47 }
```

Se retorna la función interna `interseccionF` que sería el resultado de la intersección de los conjuntos difusos. En este caso retornó 0,300000.

4. Cuarto paso



```
app > src > test > scala > taller > CojuntosDifusosTest.scala > {} taller > CojuntosDifusosTest
1 package taller
8 class CojuntosDifusosTest extends AnyFunSuite {
53
54 test("Intersección grande(1,25) y grande(1,15)") {
55   val interseccion = objCojuntosDifusos.interseccion(objCojuntosDifusos.grande(1, 25), objCojuntosDifusos.grande(1, 15))
56   assert(interseccion(26) == 0.3) // El mínimo correcto
57 }
58
59 }
```

Se pasa a verificar si el resultado dado por la función unión coincide con el test, y como es correcto finaliza la ejecución.

- **Función *inclusión***

La función inclusión toma dos conjuntos difusos, cd1 y cd2, y determina si cd1 está incluido en cd2. En conjuntos difusos, un conjunto A está incluido en otro conjunto B si el grado de pertenencia de cada elemento de A es menor o igual que el grado de pertenencia del mismo elemento en B.

def inclusión(cd1: ConjDifuso, cd2: ConjDifuso): Boolean

cd1: Un conjunto difuso de tipo ConjDifuso.

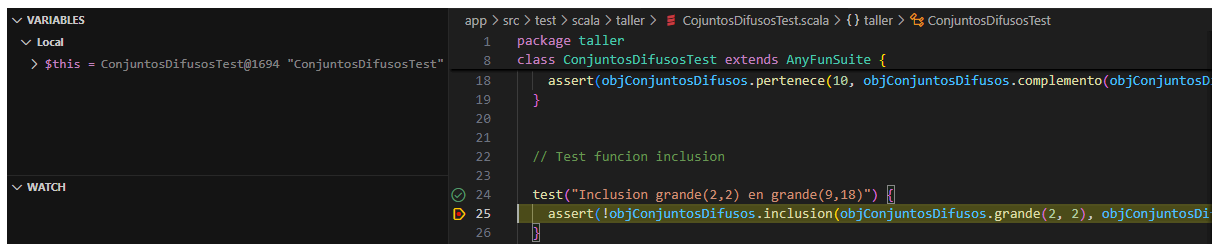
cd2: Otro conjunto difuso de tipo ConjDifuso.

Valor de retorno:

La función devuelve true si el cd1 está incluido en cd2, ósea si cada elemento de cd1 tiene un grado de pertenencia menor o igual al grado de pertenencia en cd2. Si algún elemento no lo cumple la condición, la función retorna false de inmediato.

Para cada elemento n, la inclusión se verifica como: $fS1(n) \leq fS2(n)$

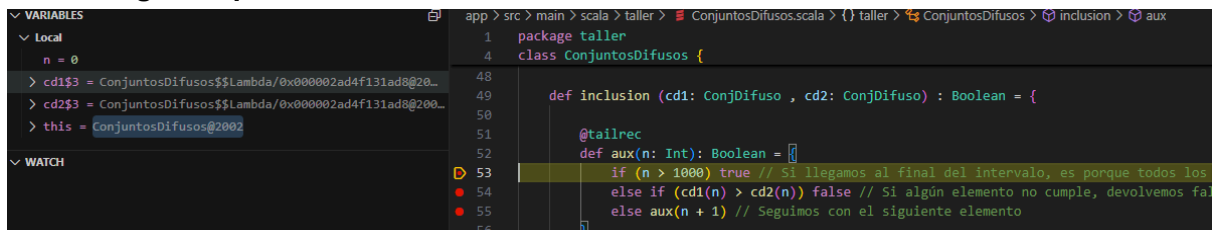
1. Primer paso



```
app > src > test > scala > taller > ConjuntosDifusosTest.scala > {} taller > ConjuntosDifusosTest
1 package taller
8 class ConjuntosDifusosTest extends AnyFunSuite {
18   assert(objConjuntosDifusos.pertenece(10, objConjuntosDifusos.complemento(objConjuntosD
19 }
20
21
22 // Test funcion inclusion
23
24 test("Inclusion grande(2,2) en grande(9,18)") {
25   assert(!objConjuntosDifusos.inclusion(objConjuntosDifusos.grande(2, 2), objConjuntosDi
26 }
27
```

Se ejecuta el test. Donde se evalúa si el conjunto difuso grande (2,2) está incluido en grande (9,1), debiendo retornar false

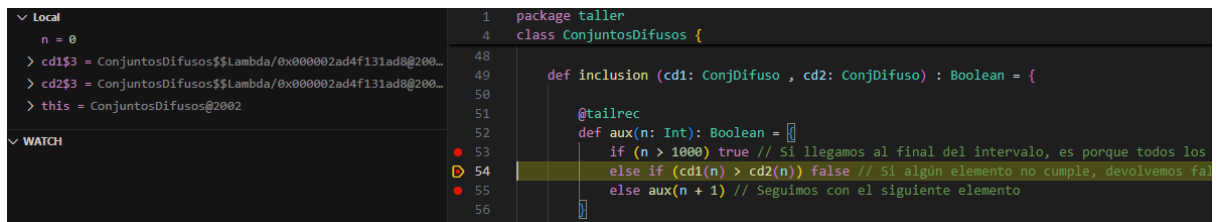
2. Segundo paso



```
app > src > main > scala > taller > ConjuntosDifusos.scala > {} taller > ConjuntosDifusos > inclusion > aux
1 package taller
4 class ConjuntosDifusos {
48
49
50
51 def inclusion (cd1: ConjDifuso , cd2: ConjDifuso) : Boolean = {
52   @tailrec
53   def aux(n: Int): Boolean = {
54     if (n > 1000) true // Si llegamos al final del intervalo, es porque todos los
55     else if (cd1(n) > cd2(n)) false // Si algún elemento no cumple, devolvemos fa
56     else aux(n + 1) // Seguimos con el siguiente elemento
57   }
58 }
```

Entra en el primero if donde al no ser cero mayor que mil pasa al siguiente condicional

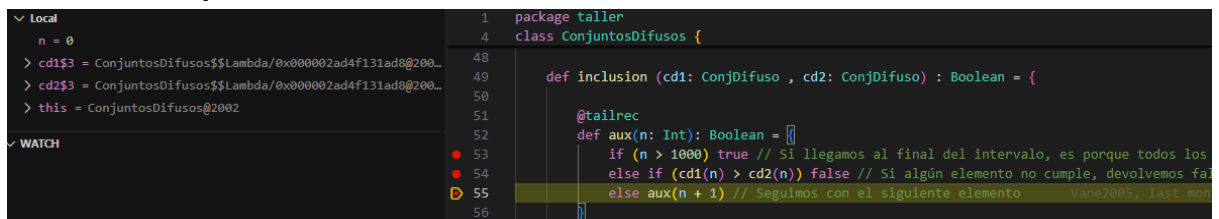
3. Tercer paso



```
1 package taller
4 class ConjuntosDifusos {
48
49   def inclusion (cd1: ConjDifuso , cd2: ConjDifuso) : Boolean = {
50
51     @tailrec
52     def aux(n: Int): Boolean = {
53       if (n > 1000) true // Si llegamos al final del intervalo, es porque todos los
54       else if (cd1(n) > cd2(n)) false // Si algún elemento no cumple, devolvemos fa
55       else aux(n + 1) // Seguimos con el siguiente elemento
56     }
57   }
58 }
```

En el else if entra y al no cumplir con el condicional, avanza al siguiente

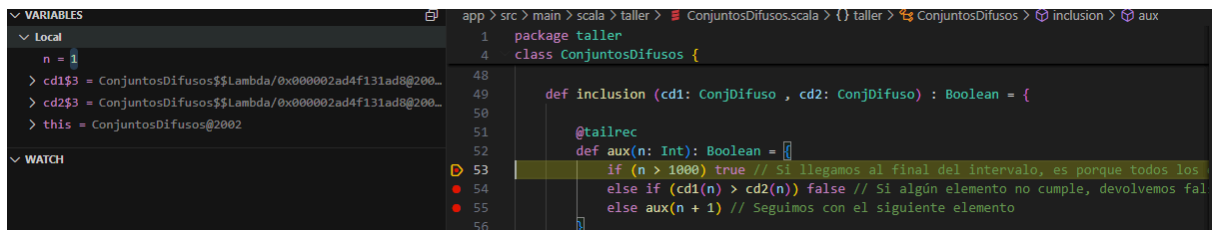
4. Cuarto paso



```
1 package taller
4 class ConjuntosDifusos {
48
49   def inclusion (cd1: ConjDifuso , cd2: ConjDifuso) : Boolean = {
50
51     @tailrec
52     def aux(n: Int): Boolean = {
53       if (n > 1000) true // Si llegamos al final del intervalo, es porque todos los
54       else if (cd1(n) > cd2(n)) false // Si algún elemento no cumple, devolvemos fa
55       else aux(n + 1) // Seguimos con el siguiente elemento
56     }
57   }
58 }
```

En el else al no cumplir ninguno de los anteriores suma a n el valor de uno

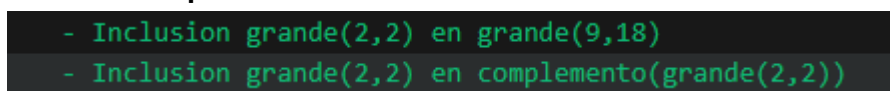
5. Quinto paso



```
1 package taller
4 class ConjuntosDifusos {
48
49   def inclusion (cd1: ConjDifuso , cd2: ConjDifuso) : Boolean = {
50
51     @tailrec
52     def aux(n: Int): Boolean = {
53       if (n > 1000) true // Si llegamos al final del intervalo, es porque todos los
54       else if (cd1(n) > cd2(n)) false // Si algún elemento no cumple, devolvemos fal
55       else aux(n + 1) // Seguimos con el siguiente elemento
56     }
57   }
58 }
```

A partir de aquí entra en un bucle donde evalúa cada uno de los datos, hasta que este sea mayor que mil, es decir, todos los datos sean evaluados, o se encuentre uno que no cumpla la condición del else if

6. Sexto paso



```
- Inclusion grande(2,2) en grande(9,18)
- Inclusion grande(2,2) en complemento(grande(2,2))
```

Dado a la gran cantidad de procesos paso a paso que habría que capturar, se realizó una captura del test pasado, donde se puede ver que lo pasa y, por tanto, la función retorna una false como el resultado correcto.

- **Función *igualdad***

La función toma como entrada dos conjuntos difusos cd1 y cd2, y determina si son iguales, en los conjuntos difusos, la igualdad se define de manera que el grado de pertenencia de cada elemento en cd1 es igual al grado de pertenencia de ese mismo elemento en cd2

```
def igualdad(cd1: ConjDifuso, cd2: ConjDifuso): Boolean
```

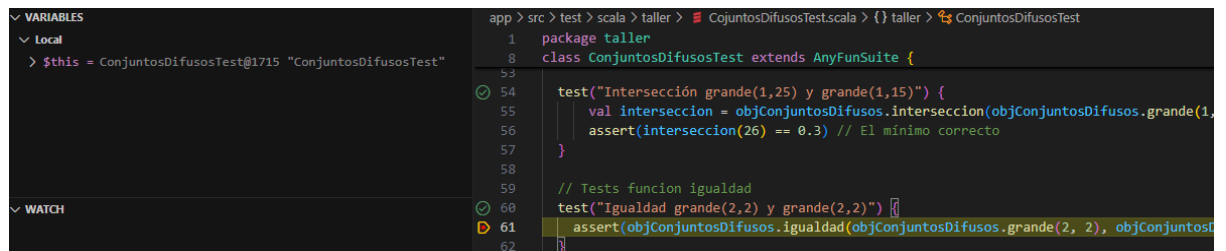
cd1: Un conjunto difuso de tipo ConjDifuso.

cd2: Otro conjunto difuso de tipo ConjDifuso

Valor de retorno:

La función devuelve true si cd1 y cd2, si el grado de pertenencia de cada elemento de ambos conjuntos son iguales, de lo contrario devuelve false. Verificandose de la siguiente manera: $f_{S1}(n)=f_{S2}(n)$

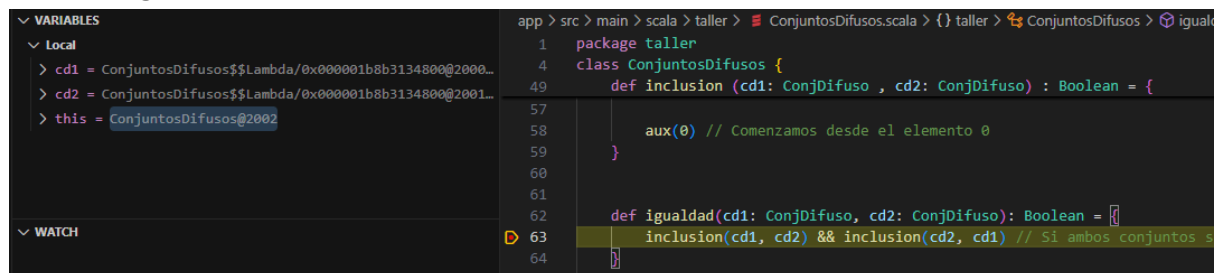
1. Primer paso



```
app > src > test > scala > taller > ConjuntosDifusosTest.scala > {} taller > ConjuntosDifusosTest
1 package taller
8 class ConjuntosDifusosTest extends AnyFunSuite {
53
54 test("Intersección grande(1,25) y grande(1,15)") {
55     val interseccion = objConjuntosDifusos.interseccion(objConjuntosDifusos.grande(1,
56     assert(interseccion(26) == 0.3) // El mínimo correcto
57 }
58
59 // Tests función igualdad
60 test("Igualdad grande(2,2) y grande(2,2)") {
61     assert(objConjuntosDifusos.igualdad(objConjuntosDifusos.grande(2, 2), objConjuntosDifusos.grande(2, 2)))
62 }
```

Se ejecuta el test. En este caso los conjuntos difusos son grande(2,2) y grande(2,2) y se evalúa si son iguales.

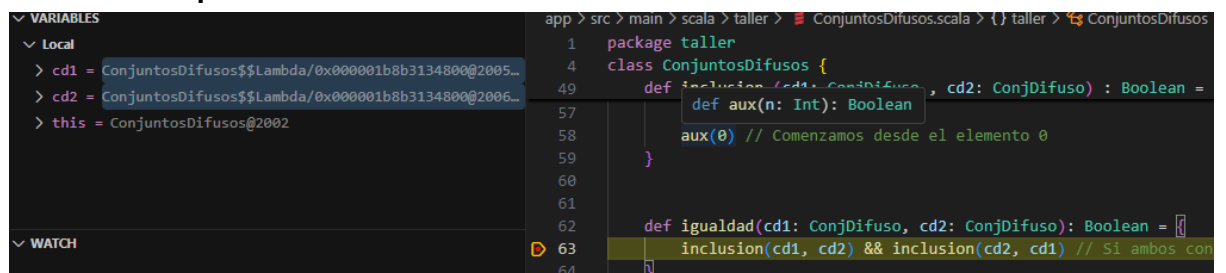
2. Segundo paso



```
app > src > main > scala > taller > ConjuntosDifusos.scala > {} taller > ConjuntosDifusos > igualdad
1 package taller
4 class ConjuntosDifusos {
49     def inclusion (cd1: ConjDifuso , cd2: ConjDifuso) : Boolean = {
57
58         aux(0) // Comenzamos desde el elemento 0
59     }
60
61
62     def igualdad(cd1: ConjDifuso, cd2: ConjDifuso): Boolean = {
63         inclusion(cd1, cd2) && inclusion(cd2, cd1) // Si ambos conjuntos s
64     }
65 }
```

Entra en la función igualdad y llama a la función inclusión para que verifique que ambas se contienen mutuamente, si esto es verdad retorna true de caso contrario false, en este paso primero verifica que cd1 esté incluido en cd2

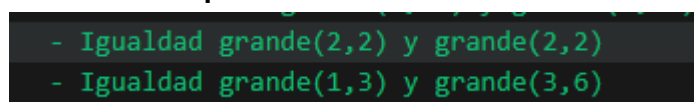
3. Tercer paso



```
app > src > main > scala > taller > ConjuntosDifusos.scala > {} taller > ConjuntosDifusos
1 package taller
4 class ConjuntosDifusos {
49     def inclusion (cd1: ConjDifuso , cd2: ConjDifuso) : Boolean = {
57         def aux(n: Int): Boolean
58         aux(0) // Comenzamos desde el elemento 0
59     }
60
61
62     def igualdad(cd1: ConjDifuso, cd2: ConjDifuso): Boolean = {
63         inclusion(cd1, cd2) && inclusion(cd2, cd1) // Si ambos con
64     }
65 }
```

Ahora revisa que cd2 esté incluido en cd1.

4. Cuarto paso



```
- Igualdad grande(2,2) y grande(2,2)
- Igualdad grande(1,3) y grande(3,6)
```

Al ser ambos llamados de la inclusión verdaderos, retorna verdadero, lo cual concuerda con lo esperado por el test.

ARGUMENTACIÓN SOBRE LA CORRECCIÓN

- **Demostración por inducción estructural de la función pertenece:**

Un **conjunto difuso** es una función $S: \text{Int} \Rightarrow \text{Double}$ que devuelve un valor entre 0 y 1, indicando el grado de pertenencia de un número entero al conjunto. Cada función puede tener diferentes comportamientos según su definición, pero siempre cumple con esta estructura.

Hipótesis de inducción:

Sea S un conjunto difuso definido recursivamente, que asocia a cada entero n un valor $S(n)$ tal que $0 \leq S(n) \leq 1$. Queremos probar por inducción estructural que, para cualquier conjunto difuso S , la función $\text{pertenece}(\text{elem}, S)$ devuelve el grado correcto de pertenencia del elemento elem a S .

Paso base:

Consideremos el caso más simple donde el conjunto difuso S es una función constante. Es decir, para cualquier entero n , $s(n)$ es siempre un valor constante C tal que $0 \leq C \leq 1$. Por ejemplo:

$\text{val } S: \text{ConjDifuso} = (n: \text{Int}) \Rightarrow 0.5$

Para cualquier elem , la función $\text{pertenece}(\text{elem}, S)$ simplemente aplicará S al valor elem , lo que resulta en 0.5.

$\text{pertenece}(\text{elem}, S) = S(\text{elem}) = 0.5$

Esto es correcto porque, por definición, el valor de pertenencia de cualquier elemento a S es constante y cumple $0 \leq 0.5 \leq 1$. Esto prueba el caso base.

Paso inductivo:

Ahora supongamos que pertenece funciona correctamente para un conjunto difuso S definido como una función más compleja, por ejemplo, una que depende del valor de n . Consideremos el siguiente conjunto difuso:

$\text{val } S: \text{ConjDifuso} = (n: \text{Int}) \Rightarrow \text{if } (n > 10) \text{ } 0.8 \text{ else } 0.2$

Este conjunto difuso devuelve 0.8 si el número es mayor que 10, y 0.2 en caso contrario. Queremos probar que pertenece funciona correctamente para cualquier valor de elem .

Si $\text{elem} > 10$, entonces:

$\text{pertenece}(\text{elem}, S) = S(\text{elem}) = 0.8$

Este valor es correcto ya que el conjunto difuso S está definido para devolver 0.8 cuando el número es mayor que 10.

Si $\text{elem} \leq 10$, entonces:

$\text{pertenece}(\text{elem}, S) = S(\text{elem}) = 0.2$

Esto también es correcto, ya que el conjunto difuso S está definido para devolver 0.2 cuando el número es menor o igual a 10.

- **Demostración por inducción matemática de la función grande:**

La función grande genera un conjunto difuso (una función ConjDifuso) que calcula el grado de pertenencia de un número n de acuerdo con la fórmula:

$$\text{mma}(n) = \left(\frac{n}{n+d}\right)^e$$

Se quiere probar por **inducción matemática** que para cualquier número entero $n \geq 0$, el valor de $\text{mma}(n)$ cumple $0 \leq \text{mma}(n) \leq 1$. Esto asegura que la función grande genera un conjunto difuso válido.

Paso base (para $n=0$):

Cuando $n=0$, la expresión para $\text{mma}(n)$ es:

$$\text{mma}(0) = \left(\frac{0}{0+d}\right)^e = 0$$

Esto es correcto porque $0^e = 0$ para cualquier $e > 1$, y claramente $0 \leq 0 \leq 1$.

Paso inductivo:

Suponiendo que para un valor $n=k$, la fórmula para $\text{mma}(k)$ cumple $0 \leq \text{mma}(k) \leq 1$. Queremos demostrar que también se cumple para $n=k+1$.

La fórmula para $n=k+1$ es:

$$\text{mma}(k+1) = \left(\frac{k+1}{(k+1)+d}\right)^e$$

observando que:

$$\left(\frac{k+1}{(k+1)+d}\right) = \left(\frac{k}{k+d}\right) \text{ para cualquier } d > 0$$

Esto se debe a que, al aumentar k en 1, el denominador crece más rápido que el numerador. Como resultado, el valor de la fracción disminuye.

Dado que $0 \leq \left(\frac{k}{k+d}\right) \leq 1$, se cumple también que $0 \leq \left(\frac{k+1}{(k+1)+d}\right) \leq 1$

Siendo así, se ha demostrado que si $0 \leq \text{mma}(k) \leq 1$, entonces $0 \leq \text{mma}(k+1) \leq 1$.

- **Demostración por inducción estructural de la función complemento:**

Hipótesis de inducción:

Para cualquier conjunto difuso C , sabemos que $0 \leq C(n) \leq 1$ para cualquier $n \in \mathbb{Z}$. Queremos probar que $0 \leq \text{comp}(n) \leq 1$, donde $\text{comp}(n) = 1 - C(n)$.

Paso base :

Consideremos un conjunto difuso C que es constante, es decir, para cualquier n , el valor de pertenencia es siempre una constante $C(n) = C_0$, donde $0 \leq C_0 \leq 1$. Un ejemplo sería:

val C : ConjDifuso = (n : Int) \Rightarrow 0.7

val comp = complemento(C)

Ahora, para cualquier n :

$$\text{comp}(n) = 1 - C(n) = 1 - 0.7 = 0.3$$

Este valor es correcto, ya que cumple $0 \leq 0.3 \leq 1$.

Paso inductivo:

Supongamos que para un conjunto difuso más complejo C , que depende del valor de n , se cumple $0 \leq C(n) \leq 1$ para cualquier n . Queremos demostrar que también se cumple $0 \leq \text{comp}(n) \leq 1$.

Tomando como ejemplo un conjunto difuso C definido de la siguiente manera:

val C : ConjDifuso = (n : Int) \Rightarrow if ($n > 10$) 0.8 else 0.2

Caso 1: Si $n > 10$, entonces $C(n) = 0.8$. Aplicamos la función complemento:

$$\text{comp}(n) = 1 - C(n) = 1 - 0.8 = 0.2$$

Este valor es correcto, ya que $0 \leq 0.2 \leq 1$.

Caso 2: Si $n \leq 10$, entonces $C(n) = 0.2$. Aplicamos la función complemento:

$$\text{comp}(n) = 1 - C(n) = 1 - 0.2 = 0.8.$$

Este valor también es correcto, ya que $0 \leq 0.8 \leq 1$.

- **Demostración por inducción matemática de la función inclusión:**

En términos de conjuntos difusos, decimos que un conjunto difuso A está **incluido** en otro conjunto difuso B si para todo elemento x , el grado de pertenencia de x en A es menor o igual que su grado de pertenencia en B .

Hipótesis de inducción:

Queremos probar que para cualquier n , si la función $\text{aux}(n)$ devuelve true, entonces para todo $x \geq n$, $\text{cd1}(x) \leq \text{cd2}(x)$. Por otro lado, si la función $\text{aux}(n)$ devuelve false, esto implica que existe algún $x \geq n$ para el cual $\text{cd1}(x) > \text{cd2}(x)$.

Paso base ($n=0$):

Cuando $n=0$, la función aux evalúa:

```
if (cd1(0) > cd2(0)) false else aux(1)
```

Existen dos posibilidades:

1. Si $\text{cd1}(0) > \text{cd2}(0)$, la función devuelve inmediatamente false, lo cual es correcto porque ya encontramos un valor $n=0$ tal que $\text{cd1}(n)$ no está incluido en $\text{cd2}(n)$.
2. Si $\text{cd1}(0) \leq \text{cd2}(0)$, entonces la función continúa evaluando el siguiente valor $n=1$, lo que significa que hasta ahora la inclusión es válida.

Por lo tanto, para $n=0$, la función se comporta correctamente.

Paso inductivo:

Supongamos que para un valor $n=k$, la función $\text{aux}(k)$ devuelve true, lo que significa que para todo $x \geq k$, se cumple $\text{cd1}(x) \leq \text{cd2}(x)$. Queremos probar que la función también se comporta correctamente para $n = k+1$.

La función evalúa lo siguiente para $n = k+1$:

```
if (cd1(k + 1) > cd2(k + 1)) false else aux(k + 2)
```

Nuevamente, existen dos posibilidades:

1. Si $\text{cd1}(k+1) > \text{cd2}(k+1)$, la función devuelve inmediatamente false, lo que indica que la inclusión no se cumple para $n=k+1$, lo cual es correcto.
2. Si $\text{cd1}(k+1) \leq \text{cd2}(k+1)$, entonces la función llama recursivamente a $\text{aux}(k + 2)$. Dado que estamos suponiendo que la función se comporta correctamente para valores mayores de $n=k$, concluimos que también se comporta correctamente para $n=k+1$.

Paso final (cuando $n>1000$):

La condición de salida de la recursión es:

```
if (n > 1000) true
```

Esto implica que si hemos revisado todos los valores desde $n=0$ hasta $n=1000$ y ninguno de ellos no cumplió la condición $\text{cd1}(n) \leq \text{cd2}(n)$, la función retorna true, indicando que todos los elementos de cd1 están incluidos en cd2 .

- **Demostración de la función unión por inducción matemática**

Definición: Sea un conjunto difuso A definido como una función que asigna a cada elemento $x \in Z$ un grado de pertenencia $A(x) \in [0, 1]$. Es decir, $A: Z \rightarrow [0, 1]$. Para los conjuntos difusos A y B, se define la unión como: $(A \cup B)(x) = \max(A(x), B(x))$.

Caso base:

Aea $n=0$:

$$(A \cup B)(0) = \max(A(0), B(0)).$$

La implementación evalúa correctamente el máximo de las pertenencias de los elementos $A(0)$ y $B(0)$ lo que coincide con la definición matemática de la unión.

Paso inductivo:

Supongamos que para un $k \in Z$, la función **union** es correcta, es decir:

$$(A \cup B)(k) = \max(A(k), B(k)).$$

Ahora se considera el caso $k + 1$:

$$(A \cup B)(k + 1) = \max(A(k + 1), B(k + 1)).$$

La función `union(k + 1)` calcula `math.max(cd1(k + 1), cd2(k + 1))`, que corresponde exactamente al valor correcto según la definición de la unión de conjuntos difusos.

- **Demostración de la función Intersección**

Definición: Sea un conjunto difuso A definido como una función que asigna a cada elemento $x \in Z$ un grado de pertenencia $A(x) \in [0, 1]$. Es decir, $A: Z \rightarrow [0, 1]$. Para los conjuntos difusos A y B, se define la intersección como: $(A \cap B)(x) = \min(A(x), B(x))$.

Paso base:

Sea $n=0$:

$$(A \cap B)(0) = \min(A(0), B(0)).$$

La implementación evalúa correctamente el mínimo de las pertenencias de los elementos $A(0)$ y $B(0)$ lo que coincide con la definición matemática de la intersección.

Paso inductivo:

Supongamos que para un $k \in Z$, la función **interseccion** es correcta, es decir:

$$(A \cap B)(k) = \min(A(k), B(k)).$$

Ahora consideramos el caso para $k + 1$:

$$(A \cap B)(k + 1) = \min(A(k + 1), B(k + 1)).$$

la función `interseccion(k + 1)` calcula `math.min(cd1(k + 1), cd2(k + 1))`, que corresponde exactamente al valor correcto según la definición de la intersección de conjuntos difusos.

- **Demostración de la función igualdad:**

Definición: La función igualdad verifica si dos conjuntos difusos son iguales, es decir, si se tienen dos conjuntos A y B, estos son iguales si para todo elemento de $x \in Z$, $A(x) = B(x)$. Formalmente, $A=B$ si y solo si para todo x , $A(x) = B(x)$.
Teniendo que para este caso, compara la pertenencia de los elementos en A y B, devolviendo true si $A(x) = B(x)$ para todos los x , y false en caso contrario.

Paso base ($n=0$)

Para el caso base, se evalúa si los conjuntos A y B son iguales en $n = 0$:

- Si $A(0) = B(0)$, entonces la función verifica el siguiente elemento
- Si $A(0) \neq B(0)$, entonces la función devuelve false y, por tanto, A y B no son iguales

Si la función devuelve true, es correcto porque no se ha encontrado diferencia en los grados de pertenencia entre A y B.

Paso inductivo

Se supone que para un valor k, la función igualdad entre A y B devuelve true si $A(x) = B(x)$ para todos los $x \leq k$, y false si encuentra algún $x \leq k$ donde A y B no sean iguales

Entonces lo siguiente es demostrar que la función también retorna true para el caso $k + 1$

1. La función de igualdad entre A y B evalúa si $A(k+1) = B(k+1)$.
2. Si $A(k+1) \neq B(k+1)$, entonces la función devuelve false, lo cual es correcto porque se ha encontrado un elemento para el cual A y B no tienen el mismo grado de pertenencia
3. Si $A(k+1) = B(k+1)$, la función continua, evaluando el siguiente valor, llamando recursivamente a igualdad(A, B) para $k+2$.

Se asume que la función igualdad(A, B) se comporta correctamente para valores hasta k, entonces se pudo ver que también se comporta correctamente para su consecutivo ($k+1$).

CONCLUSIONES

- El concepto de grado de pertenencia en conjuntos difusos permite categorizar elementos de manera más flexible que en conjuntos clásicos. Esto se evidencia en la función **pertenece**, que devuelve valores entre 0 y 1, indicando la pertenencia de un elemento a un conjunto difuso. Además, la función complemento proporciona una forma de calcular el grado de “no pertenencia”, lo que permite una comprensión más completa de la relación entre los elementos y los conjuntos difusos.

Vanessa Alexandra Durán Mona - 2359394

Juan Damián Cuervo Buitrago - 2359413

Juan Sebastián Rodas Ramírez - 2359681

- Las funciones de unión e intersección demuestran cómo se pueden combinar conjuntos difusos para obtener nuevos grados de pertenencia. La unión toma el máximo grado de pertenencia entre dos conjuntos, mientras que la intersección utiliza el mínimo. Esto no solo refuerza la utilidad de los conjuntos difusos en aplicaciones prácticas, sino que también establece un marco matemático riguroso para trabajar con estas operaciones.