Controlador de aparcamiento usando lógica difusa

César Antonio Enrique Ramírez

Universidad de Huelva

Resumen En el presente documento se expone el desarrollo de varios módulos escritos en Haskell con el objetivo de utilizarlos para definir sistemas basados en lógica difusa y describir un controlador difuso de aparcamiento en batería con ellos. El desarrollo de este trabajo está basado en [1]

1. Introducción

La lógica difusa es un tipo de lógica en el que las proposiciones no son verdaderas o falsas, sino que tienen un *grado de verdad* que se representa mediante un número real comprendido entre 0 y 1. El valor 0 indica que la proposición es totalmente falsa. El valor 1 indica que la proposición es totalmente verdadera. Los valores intermedios indican que la proposición debe considerarse verdadera en cierto grado.

Los diferentes conceptos usados en lógica difusa se representan mediante fuciones de pertenencia. Estas fuciones pueden tener muchas formas, pero generalmete para describir conceptos relacionados con variables continuas se utilizan fuciones de pertenencia con forma de triángulo, trapezoide, rampa, campana, sigmoide, etc.

Una proposición difusa básica expresa la relación entre el valor de una cierta variable y un concepto. Por ejemplo, *la temperatura es alta* es una proposición difusa que relaciona la variable *temperatura* con el concepto *alta*. Al evaluar la proposición aplicando el valor de la variable a la función de pertenencia, obtenemos un valor entre 0 y 1 que representa el *grado de activación* de la proposición.

Las proposiciones difusas se componen por medio de operaciones AND y OR. Estas operaciones representan los operadores lógicos AND y OR, pero deben ser funciones definidas sobre valores reales entre 0 y 1 para poder aplicarlas a valores difusos. Las funciones que cumplen este requisito se llaman T-normas y S-normas. Las T-normas más utilizadas para representar la operación AND and son el mínimo o el producto (en esta implementación usaremos el producto). Para representar la operación OR se utilizan el máximo o la suma acotada entre otras (en nuestro caso usaremos la suma algebraica).

Una regla difusa es una contrucción del tipo *if ... then ...* donde el antecedente de la regla es una proposición difusa.

Una base de reglas es un conjunto de reglas en las que las variables de entrada del sistema se usan en el antecedente y las variables de salida se usan en los consecuentes. Mediante un método de "defuzzificación" se obtienen las variables de salida a partir del grado de activación de las reglas y su consecuente.

2. Implementación

Para la implementación de este sistema difuso se ha dividido la funcionalidad en tres módulos:

- El módulo *Fuzzy.hs* contiene la definición de los tipos de datos necesarios para la implementación del resto de módulos.

- El módulo *FuzzyLib.hs* contiene la definición de funciones de pertenencia, así como de T-normas, S-normas y funciones de defuzzificacion.
- El módulo *FuzzyParking.hs* contiene la definición concreta del sistema difuso para el problema del aparcamiento en batería.

En esta memoria no entraremos en detalle con cada uno de ellos por ser relativamente sencillos además de estar debidamente documentados, por lo que en vez de eso, nos detendremos a analizar detalladamente algunas partes clave del funcionamiento del sistema.

El primer trozo de código que analizaremos corresponde a la definición de una fución de pertenencia. Para ilustrar la técnica que hemos usado vamos a emplear la fución de pertenencia con forma de triángulo, pero el concepto se aplica igual con el resto.

```
triangle :: (Float, Float, Float) -> MemFunc
triangle (a,b,c) f
  | f > a && f <= b = (f-a) / (b-a)
  | f > b && f < c  = 1 - (f-b) / (c-b)
  | otherwise       = 0
```

Figura 1: Función de pertenencia en forma de triángulo

Como podemos ver en la figura 1, la función *triangle* es de tipo *(Float, Float, Float) -¿MemFunc*, lo que quiere decir que toma como parámetro una tupla con tres valores reales y devuelve un valor de tipo *MemFunc*, que está definido en el módulo *Fuzzy.hs*.

```
type memFunc = (Float -> Fuzzy)
```

Por lo tanto, la función *triangle* recibe como parámetro una tupla y devuelve una fución, la cual toma como parámetro un valor real y de vuelve un *Fuzzy* (que no es más que un valor real entre 0 y 1).

El tipo de la función *triangle* se podría reescribir como y sería totalmente válido, sin embargo

```
triangle :: (Float, Float, Float) -> Float -> Fuzzy
```

utilizamos el tipo *MemFunc* para hacer énfasis en que lo que devuelve no es sólo el *Fuzzy* que aparece al final de la definición del tipo, sino una función de tipo *Float ->Fuzzy*. Esto es así gracias a la aplicación parcial de funciones. De hecho, la definición de la función tiene cuatro parámetros de entrada, pero cuando la llamamos en el resto del código sólo le pasamos tres. De esta forma, podemos definir funciones de pertenencia parametrizadas sin necesidad de progrmarlas a mano todas.

Los tres parámetros en la tupla que recibe la fución *triangle* corresponden a los tres vértices del triángulo descritos en una dimensión, teniendo en cuenta que la altura del triángulo es siempre 1.

El siguiente trozo de código corresponde a una función de defuzzificaicón. En concreto a la función *wightedFuzzyMean*.

En este caso, vemos una declaración de tipo *Defuzmethod*, que no es más que *[(Float, Float, Float)] ->Float*. Esta función calcula la media de los centros de los consecuentes de cada regla, ponderados por el producto entre el grado de activación de la regla y la base o el área del consecuente. La expresión 1 corresponde a su funcionamiento.

```
weightedFuzzyMean :: Defuzzmethod
weightedFuzzy x = foldr (\(c,d,a) acc -> acc + (a*d*c)) 0 x /
  (foldr (\(_,d,a) acc -> acc + (a*d)) 0 x)
```

Figura 2: Función de defuzzificación usando la media difusa ponderada por la base

$$y = \frac{\sum_r \alpha_r \cdot d_r \cdot c_r}{\sum_r \alpha_r \cdot d_r} \tag{1}$$

Los tres parámetros que recibe la función *weightedFuzzyMean* en cada tupla de la lista corresponden a el centro de la función de pertenencia del consecuente, la base de la misma, y el grado de activación de la regla en la que aparece dicho conseguente.

La siguiente función que vamos a analizar es la función que realiza la inferencia, *fuzzyParking*.

```
fuzzyParking :: Inference -- [Float] -> Float
fuzzyParking sys (x:a:_) = (defuzzmethod sys) $ zip3 centros base alphaList
  where alphaList = map (evalTerm (tnorma sys) (snorma sys) . tmap f . getTerm) $ reglas sys
        params = unzip . map (getParams . getConseq) $ (reglas sys)
        centros = fst params
        base = snd params
        f var = case var of
          "X"      -> x
          "Angulo" -> a
```

Figura 3: Función que realiza la inferencia del sistema de aparcamiento en batería

Esta función toma como parámetros un sistema difuso y una lista de parámetros de entrada para el sistema. En este caso concreto nuestro sistema sólo utiliza dos entradas. Para calcular el resultado a devolver, la función tendrá que aplicar las entradas a las reglas del sistema y calcular el valor de salida usando el método de defuzzificación del sistema, como podemos ver en la línea 2 de la figura 3.

Para calcular el grado de activación de cada regla se mapea la función *evalTerm* sobre todas las reglas. Podemos ver esta función en la figura 4.

```
evalTerm :: Tnorm -> Snorm -> Term -> Fuzzy
evalTerm and or (t1 :&& t2)          = (evalTerm and or t1) `and` (evalTerm and or t2)
evalTerm and or (t1 :|| t2)          = (evalTerm and or t1) `or`  (evalTerm and or t2)
evalTerm and or ((Val x) := memFunc) = memFunc x
evalTerm _   _   _                   = 0
```

Figura 4: Función que evalúa un término difuso, calculando así el grado de activación

Pero primero se necesita sustituir cada variable en las reglas por su correspondiente valor de entrada. Para eso usamos la función *tmap*, que aunque no es exactamente igual, recuerda a *fmap* en su funcionamiento, pero debido a sus restricciones no podemos declarar una instancia de *Functor*. Podemos ver esta función en la figura 5.

Lo siguiente será echar un vistazo a cómo definimos el sistema difuso, y para eso vamos a ver cómo definimos una regla.

Abusando un poco del sistema de tipos de haskell, hemos ideado una forma de construir los tipos que nos permite describir las reglas con una sintaxis no demasiado incómoda, siendo bastante fáciles de leer en código directamente. Primero veamos el tipo Term, tal y como aparece en la figura 6. Está formado por tres constructores, pero no son contructores normales, son *infijos*. Podemos observar

```
tmap :: (String -> Float) -> Term -> Term
tmap f (l :&& r) = (tmap f l) :&& (tmap f r)
tmap f (l :|| r) = (tmap f l) :|| (tmap f r)
tmap f ((Var x) := mem) = (Val $ f x) := mem
```

Figura 5: Función que aplica una función de String a Float a la estuctura de un Term

también que la definición es recursiva, y que el caso base es en el que *Term* está definido como una asignación de una función de pertenencia a una variable.

```
data Term = (Term) :&& (Term) | (Term) :|| (Term) | (Var) := (MemFunc)
```

Figura 6: Tipo de dato que representa una proposición difusa.

Examinemos a continuación el tipo Rule. Podemos verlo en la figura 7. Su constructor *IF* representa convenientemente el comienzo de una expresión lógica, a continuación recibe un *Term* y por último un *Conseq*. La estructura es muy clara: *IF regla-difusa consecuente*. También podemos ver en la figura 8 que el constructor del tipo *Conseq* también está convenientemente nombrado como *THEN*, por lo que nos queda una estructura del tipo:

$$IF((variable := metodo) : \&\&(variable2 := metodo2) : \&\&...)\$THENconsecuente \qquad (2)$$

Es importante fijarnos en el símbolo $ que hay entre la parte del antecedente y *THEN*. En vez de eso podemos usar paréntesis rodeando a todo el consecuente.

En la figura 9 podemos ver código en el que se definen algunas reglas.

```
data Rule = IF Term Conseq
```

Figura 7: Tipo de dato que representa una regla del sistema difuso.

Por último, tenemos la función *main*, la cual podemos ver en la figura 10, que calcula la salida del sistema difuso para todos los posibles valores del ángulo y la distancia. Con estos valores podremos representar gráficamente al sistema.

3. Grafica

Si representamos los datos obtenidos de la ejecución del programa vemos como la gráfica representa una superficie con varios niveles. Dichos niveles corresponden con las reglas que hemos definido para el sistema, basándonos en las definidas en [1].

4. Pruebas

Las puebas se han diseñado usando el framework *HUnit*. Podemos ver los diferentes casos de prueba que se han diseñado en la figura 12.

```
data Conseq = THEN Var (Float, Float)
```

Figura 8: Tipo de dato que representa el consecuente de una regla.

```
reglas = [
IF ((Var "X" := lE) :&& (Var "Angulo" := rB)) $ THEN (Var "volante") (sera3 ps),
IF ((Var "X" := lE) :&& (Var "Angulo" := rU)) $ THEN (Var "volante") (sera3 ns),
IF ((Var "X" := lE) :&& (Var "Angulo" := rV)) $ THEN (Var "volante") (sera3 nm),
IF ((Var "X" := lE) :&& (Var "Angulo" := vE)) $ THEN (Var "volante") (sera3 nm),
...
]
```

Figura 9: Reglas del sistema difuso escritas con una sintaxis facil de leer.

```
main :: IO ()
main = putStrLn $ (unlines . map (unwords . map (show))) vs
  where vs = [[a,x,fuzzyParking system [x,a]] | x <- [-50,-45..50], a <- [-180,-170..180]]
```

Figura 10: Función main, que calcula la salida del sistema difuso para todos los posibles valores de entrada.

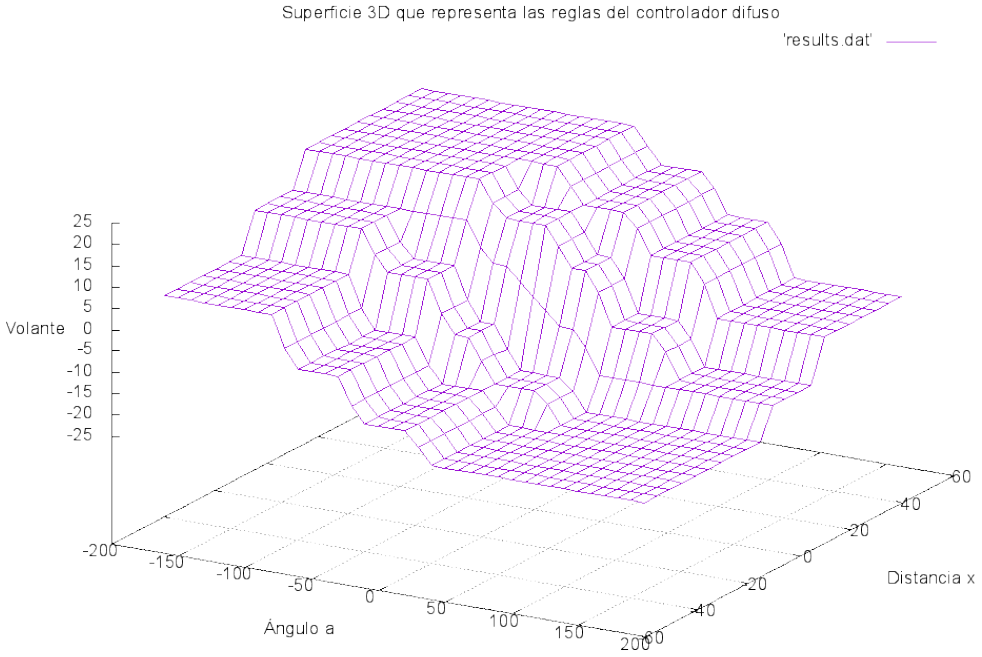

Figura 11: Representación gráfica de la salida del sistema para todos los valores posibles.

```
tests = test [
"test-triangle-move1" ~: (triangle (5,6,7)) 5.3 ~=? ((triangle (-5,-4,-3)) . flip (-) 10) 5.3,
"test-triangle-move2" ~: (triangle (5,6,7)) 6.3 ~=? ((triangle (-5,-4,-3)) . flip (-) 10) 6.3,
"test-triangle-move3" ~: (triangle (5,8,9.5)) 6 ~=? ((triangle (-5,-2,-0.5)) . flip (-) 10) 6,

"test-trapezoid-move1" ~: (trapezoid (5,8,9.5, 10)) 6 ~=? ((trapezoid (-5,-2,-0.5, 0)) . flip (-) 10) 6,
"test-trapezoid-move2" ~: (trapezoid (5,8,9.5, 10)) 9 ~=? ((trapezoid (-5,-2,-0.5, 0)) . flip (-) 10) 9,
"test-trapezoid-move3" ~:
  (trapezoid (5,8,9.5, 10)) 9.7 ~=? ((trapezoid (-5,-2,-0.5, 0)) . flip (-) 10) 9.7,

"test-sramp-move1" ~: (sramp (5,8)) 9 ~=? ((sramp (-5,-2)) . flip (-) 10) 9,
"test-sramp-move2" ~: (sramp (5,8)) 7 ~=? ((sramp (-5,-2)) . flip (-) 10) 7,

"test-zramp-move1" ~: (zramp (5,8)) 4 ~=? ((zramp (-5,-2)) . flip (-) 10) 4,
"test-zramp-move2" ~: (zramp (5,8)) 7 ~=? ((zramp (-5,-2)) . flip (-) 10) 7,

"test-trapezoid1" ~: (trapezoid (5,8,9.5, 10)) 4 ~=? 0,
"test-trapezoid21" ~: (trapezoid (5,8,9.5, 10)) 6 > 0 ~? "",
"test-trapezoid22" ~: (trapezoid (5,8,9.5, 10)) 6 < 1 ~? "",
"test-trapezoid3" ~: (trapezoid (5,8,9.5, 10)) 9 ~=? 1,

"test-sramp-zramp" ~: (sramp (5,8)) 7 ~=? (1 - ((zramp (5,8))  7)),

"test-tnormMin" ~: tnormMin 0.5 0.8 ~=? 0.5,
"test-tnormProd" ~: tnormProd 0.5 0.8 ~=? 0.4,
"test-snormMax" ~: snormMax 0.5 0.8 ~=? 0.8,
"test-snormSum" ~: snormSum 0.5 0.8 ~=? 0.9,

"test-2fuzzyMean" ~:
   fuzzyMean [(10, 20, 0.5), (30, 20, 0.1)] ~=? weightedFuzzyMean [(10, 20, 0.5), (30, 20, 0.1)]
             ]
```

Figura 12: casos de prueba

5. Conclusión

Para concluir, podemos decir que el desarrollo de un sistema difuso se adecua mucho a las caracteríasticas y funcionalidades que proporciona Haskell tanto por su sintaxis como por usar el paradigma de programación funcional. El sistema no ha sido complicado de desarrollar pero si muy reconfortante cuando ves los resultados. Como cosas a mejorar, se podría hacer una parser para obtener el systema difuso desde una especificación en un fichecho externo, y desarrollar funcionalidad que permita fácilmente procesar sistemas con más de una variable de salida, ya que en lo que a esta implementación respecta, por parte de las librerías sí está soportado, pero lo que es el sistema difuso para el controlador de aparcamiento en batería, como solo tiene una variable de salida no se ha implentado el manejo de reglas con diferentes variables en el consecuente.

6. Código

El código de este trabajo se puede descargar desde el repositorio `git@github.com:caenrique/fuzzyParking.git`, o revisarlo online desde `https://github.com/caenrique/fuzzyParking`.

Referencias

1. Kong, S.-G., and Kosko, B. Adaptive fuzzy systems for backing up a truck-and-trailer. *IEEE TRANSACTIONS ON NEURAL NETWORKS 3* (1992), 211–223.