# Lógica de primer orden en Haskell

Eduardo Paluzo

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 16 de junio de 2016 (Versión de 11 de julio de 2016)

# Índice general

# Introducción

El objetivo del trabajo es la implementación de los algoritmos de la lógica de primer orden en Haskell. Consta de dos partes:

- La primera parte consiste en la adaptación de los programas del libro de J. van Eijck Computational semantics and type theory [1] ([4]) y su correspondiente teoría.
- En la segunda parte se programan en Haskell los algoritmos de la lógica de primer orden estudiados en la asignatura de Lógica matemática y fundamentos [2] ([2]).

[1] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.467.1441&rep=rep1&type=pdf
[2] https://www.cs.us.es/~jalonso/cursos/lmf-15

# Parte I

# Semántica computacional y teoría de tipos

# Capítulo 1

# Programación funcional con Haskell

En este capítulo se hace una breve introducción a la programación funcional en Haskell suficiente para entender su aplicación en los siguientes capítulos. Para una introducción más amplia se pueden consultar los apuntes de la asignatura de Informática de 1º del Grado en Matemáticas ([1]). También se puede emplear como lectura complementaria, y se ha empleado para algunas definiciones del trabajo ([5])

El contenido de este capítulo se encuentra en el módulo `PFH`

```
module PFH where
```

## 1.1. Introducción a Haskell

Para hacer una introducción intuitiva a Haskell, se proponen a una serie de funciones ejemplo. A continuación se muestra la definición de una función en Haskell. `(cuadrado x)` es el cuadrado de `x`. Por ejemplo,

```
ghci> cuadrado 3
9
ghci> cuadrado 4
16
```

La definición es

```
cuadrado :: Int -> Int
cuadrado x = x * x
```

Definimos otra función en Haskell. `(raizCuadrada x)` es la raiz cuadrada entera de `x`. Por ejemplo,

```
ghci> raizCuadrada 9
3
ghci> raizCuadrada 8
2
```

La definición es

```
raizCuadrada :: Int -> Int
raizCuadrada x = last [y | y <- [1..x], y*y <= x]
```

Posteriormente, definimos funciones que determinen si un elemento `x` cumple una cierta propiedad. Este es el caso de la propiedad 'ser divisible por $n$', donde $n$ será un número cualquiera.

```
ghci> 15 `divisiblePor` 5
True
```

La definición es

```
divisiblePor :: Int -> Int -> Bool
divisiblePor x n =  x `rem` n == 0
```

Hasta ahora hemos trabajado con los tipos de datos `Int` y `Bool`; es decir, números y booleanos respectivamente. Pero también se puede trabajar por ejemplo con cadenas de caracteres, que son tipo `[Char]` o `String`. Por ejemplo, `(contieneLaLetra xs l)` identifica si una palabra contiene una cierta letra `l` dada. Por ejemplo,

```
ghci>  "descartes" `contieneLaLetra` 'e'
True
ghci>  "topologia" `contieneLaLetra` 'm'
False
```

Y su definición es

```
contieneLaLetra :: String -> Char -> Bool
contieneLaLetra [] _     = False
contieneLaLetra (x:xs) l = x == l || contieneLaLetra xs l
```

### 1.1.1. Comprensión de listas

Las listas son una representación de un conjunto ordenado de elementos. Dichos elementos pueden ser de cualquier tipo, ya sean `Int`, `Bool`, `Char`, ... Siempre y cuando, todos los elementos de dicha lista compartan tipo. En Haskell, las listas se representan

```
ghci> [1,2,3,4]
[1,2,3,4]
ghci> [1..4]
[1,2,3,4]
```

Una lista por comprensión no es más que un conjunto representado por comprensión:

$$\{x|x \in A, P(x)\}$$

Se puede leer de manera intuitiva como: "tomar aquellos $x$ del conjunto $A$ tales que cumplen una cierta propiedad $P$". En Haskell se representa

$$[x|x \leftarrow \texttt{lista, condiciones que debe cumplir}]$$

Algunos ejemplos son:

```
ghci> [n | n <- [0 .. 10], even n]
[0,2,4,6,8,10]
ghci> [x | x <- ["descartes","pitagoras","gauss"], x `contieneLaLetra` 'e']
["descartes"]
```

**Nota:** En los distintos ejemplos hemos visto que se pueden componer las distintas funciones ya definidas.

### 1.1.2. Funciones map y filter

Introducimos un par de funciones de mucha relevancia en el uso de listas en Haskell.

La función `(map f xs)` aplica una función `f` a cada uno de los elementos de la lista `xs`. Por ejemplo,

```
ghci> map (`divisiblePor` 4) [8,12,3,9,16]
[True,True,False,False,True]
ghci> map (`div` 4) [8,12,3,9,16]
[2,3,0,2,4]
ghci> map (`div` 4) [x | x <- [8,12,3,9,16], x `divisiblePor` 4]
[2,3,4]
```

Dicha función está predefinida en el paquete `Data.List`, nosotros daremos una definición denotándola con el nombre `(aplicafun f xs)`, y su definición es

```
aplicafun :: (a -> b) -> [a] -> [b]
aplicafun f []     = []
aplicafun f (x:xs) = f x : aplicafun f xs
```

La función `(filter p xs)` es la lista de los elementos de `xs` que cumplen la propiedad `p`. Por ejemplo,

```
ghci> filter (<5) [1,5,7,2,3]
[1,2,3]
```

La función `filter` al igual que la función `map` está definida en el paquete `Data.List`, pero nosotros la denotaremos como `(aquellosQuecumplen p xs)`. Y su definición es

```
aquellosQuecumplen :: (a -> Bool) -> [a] -> [a]
aquellosQuecumplen p [] = []
aquellosQuecumplen p (x:xs) | p x       = x: aquellosQuecumplen p xs
                            | otherwise = aquellosQuecumplen p xs
```

En esta última definición hemos introducido las ecuaciones por guardas, representadas por `|`. Otro ejemplo más simple del uso de guardas es el siguiente

$$g(x) = \begin{cases} 5, & \text{si } x \neq 0 \\ 0, & \text{en caso contrario} \end{cases}$$

Que en Haskell sería

```
g :: Int -> Int
g x | x /= 0    = 5
    | otherwise = 0
```

### 1.1.3. n-uplas

Una n-upla es un elemento del tipo $(a_1, \ldots, a_n)$ y existen una serie de funciones para el empleo de las dos-uplas $(a_1, a_2)$. Dichas funciones están predefinidas bajo los nombres `fst` y `snd`, y las redefinimos como `primerElemento` y `segundoElemento` respectivamente.

```
primerElemento :: (a,b) -> a
primerElemento (x,_) = x
segundoElemento :: (a,b) -> b
segundoElemento (_,y) = y
```

### 1.1.4. Conjunción, disyunción y cuantificación

En Haskell, la conjunción está definida mediante el operador `&&`, y se puede generalizar a listas mediante la función predefinida `(and xs)` que nosotros redefinimos denotándola `(conjuncion xs)`. Su definición es

```
conjuncion :: [Bool] -> Bool
conjuncion []     = True
conjuncion (x:xs) = x && conjuncion xs
```

Dicha función es aplicada a una lista de booleanos y ejecuta una conjunción generalizada.

La disyunción en Haskell se representa mediante el operador `||`, y se generaliza a listas mediante una función predefinida `(or xs)` que nosotros redefinimos con el nombre `(disyuncion xs)`. Su definición es

```
disyuncion :: [Bool] -> Bool
disyuncion []     = False
disyuncion (x:xs) = x || disyuncion xs
```

Posteriormente, basándonos en estas generalizaciones de operadores lógicos se definen los siguientes cuantificadores, que están predefinidos como `(any p xs)` y `(all p xs)` en Haskell, y que nosotros redefinimos bajo los nombres `(algun p xs)` y `(todos p xs)`. Se definen

```
algun, todos :: (a -> Bool) -> [a] -> Bool
algun p = disyuncion . aplicafun p
todos p = conjuncion . aplicafun p
```

**Nota:** Hemos empleando composición de funciones para la definición de `(algun)` y `(todos)`. Se representa mediante `.` y además, se omite el argumento de entrada común a todas las funciones.

En matemáticas, estas funciones representan los cuantificadores lógicos $\exists$ y $\forall$, y determinan si alguno de los elementos de una lista cumple una cierta propiedad, y si todos los elementos cumplen una determinada propiedad respectivamente. Por ejemplo.

$$\forall x \in \{0,...,10\} \texttt{ se cumple que } x < 7. \texttt{ Es Falso}$$

En Haskell se aplicaría la función `(todos p xs)` de la siguiente forma

```
ghci> todos (<7) [0..10]
False
```

Finalmente, definimos las funciones `(pertenece x xs)` y `(noPertenece x xs)`

```
pertenece, noPertenece :: Eq a => a -> [a] -> Bool
pertenece   = algun . (==)
noPertenece = todos . (/=)
```

Estas funciones determinan si un elemento `x` pertenece a una lista `xs` o no.

### 1.1.5. Plegados especiales foldr y foldl

No nos hemos centrado en una explicación de la recursión pero la hemos empleado de forma intuitiva. En el caso de la recursión sobre listas, hay que distinguir un caso base; es decir, asegurarnos de que tiene fin. Un ejemplo de recursión es la función `(factorial x)`, que definimos

```
factorial :: Int -> Int
factorial 0 = 1
factorial x = x*(x-1)
```

Añadimos una función recursiva sobre listas, como puede ser `(sumaLaLista xs)`

```
sumaLaLista :: Num a => [a] -> a
sumaLaLista []     = 0
sumaLaLista (x:xs) = x + sumaLaLista xs
```

Tras este preámbulo sobre recursión, introducimos la función `(foldr f z xs)` que no es más que una recursión sobre listas o plegado por la derecha, la definimos bajo el nombre `(plegadoPorlaDerecha f z xs)`

```
plegadoPorlaDerecha :: (a -> b -> b) -> b -> [a] -> b
plegadoPorlaDerecha f z []     = z
plegadoPorlaDerecha f z (x:xs) = f x (plegadoPorlaDerecha f z xs)
```

Un ejemplo de aplicación es el producto de los elementos o la suma de los elementos de una lista

```
ghci> plegadoPorlaDerecha (*) 1 [1,2,3]
6
ghci> plegadoPorlaDerecha (+) 0 [1,2,3]
6
```

Un esquema informal del funcionamiento de `plegadoPorlaDerecha` es

$$\texttt{plegadoPorlaDerecha}\ (\otimes)\ z\ [x_1, x_2, \ldots, x_n] := x_1 \otimes (x_2 \otimes (\cdots (x_n \otimes z) \cdots))$$

**Nota:** $\otimes$ representa una operación cualquiera.

Por lo tanto, podemos dar otras definiciones para las funciones `(conjuncion xs)` y `(disyuncion xs)`

```
conjuncion1, disyuncion1 :: [Bool] -> Bool
conjuncion1 = plegadoPorlaDerecha (&&) True
disyuncion1 = plegadoPorlaDerecha (||) False
```

Hemos definido `plegadoPorlaDerecha`, ahora el lector ya intuirá que `(foldl f z xs)` no es más que una función que pliega por la izquiera. Definimos `(plegadoPorlaIzquiera f z xs)`

```
plegadoPorlaIzquierda :: (a -> b -> a) -> a -> [b] -> a
plegadoPorlaIzquierda f z []     = z
plegadoPorlaIzquierda f z (x:xs) = plegadoPorlaIzquierda f (f z x) xs
```

De manera análoga a `foldr` mostramos un esquema informal para facilitar la comprensión

$$\texttt{plegadoPorlaIzquierda}\ (\otimes)\ z\ [x_1, x_2, \ldots, x_n] := (\cdots ((z \otimes x_1) \otimes x_2) \otimes \cdots) \otimes x_n$$

Definamos una función ejemplo como es la inversa de una lista. Está predefinida bajo el nombre `(reverse xs)` y nosotros la redefinimos como `(listaInversa xs)`

```
listaInversa :: [a] -> [a]
listaInversa = plegadoPorlaIzquierda (\xs x -> x:xs) []
```

Por ejemplo

```
ghci> listaInversa [1,2,3,4,5]
[5,4,3,2,1]
```

Podríamos comprobar por ejemplo si la frase `'Yo dono rosas, oro no doy'` es un palíndromo

```
ghci> listaInversa "yodonorosasoronodoy"
"yodonorosasoronodoy"
ghci> listaInversa "yodonorosasoronodoy" == "yodonorosasoronodoy"
True
```

### 1.1.6. Teoría de tipos

**Notación $\lambda$**

Cuando hablamos de notación lambda simplemente nos referimos por ejemplo a expresiones del tipo `\x ->x+2`. La notación viene del $\lambda$ *Calculus* y se escribiría $\lambda$`x. x+2`. Los diseñadores de Haskell tomaron el símbolo \ debido a su parecido con $\lambda$ y por ser fácil y rápido de teclear. Una función ejemplo es `(divideEntre2 xs)`

```
divideEntre2 :: Fractional b => [b] -> [b]
divideEntre2 xs = map (\x -> x/2) xs
```

Para una información más amplia recomiendo consultar ([3])

**Representación de un dominio de entidades**

Construimos un ejemplo de un dominio de entidades compuesto por las letras del abecedario, declarando el tipo de dato `Entidades` contenido en el módulo `Dominio`

```
module Dominio where

data Entidades = A | B | C | D | E | F | G
               | H | I | J | K | L | M | N
               | O | P | Q | R | S | T | U
               | V | W | X | Y | Z | Inespecifico
     deriving (Eq, Bounded, Enum)
```

Se añade `deriving (Eq, Bounded, Enum)` para establecer relaciones de igualdad entre las entidades `(Eq)`, una acotación `(Bounded)` y enumeración de los elementos `(Enum)`.

Para mostrarlas por pantalla, definimos las entidades en la clase `(Show)` de la siguiente forma

```
instance Show Entidades where
    show A = "A"; show B = "B"; show C = "C";
    show D = "D"; show E = "E"; show F = "F";
    show G = "G"; show H = "H"; show I = "I";
    show J = "J"; show K = "K"; show L = "L";
    show M = "M"; show N = "N"; show O = "O";
    show P = "P"; show Q = "Q"; show R = "R";
    show S = "S"; show T = "T"; show U = "U";
    show V = "V"; show W = "W"; show X = "X";
    show Y = "Y"; show Z = "Z"; show Inespecifico = "*"
```

Colocamos todas las entidades en una lista

```
entidades :: [Entidades]
entidades = [minBound..maxBound]
```

De manera que si lo ejecutamos

```
ghci> entidades
[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,*]
```

.

# Capítulo 2

# Lógica de predicados en Haskell

En este capítulo se estudiará la lógica de predicados en Haskell, para ello necesitamos de un preámbulo de definiciones necesarias, propias de la lógica.

## 2.1. Conceptos previos

El alfabeto de la lógica proposicional está compuesto por

1. Variables proposicionales
2. Conectivas lógicas:

| | |
|---|---|
| $\neg$ | Negación |
| $\vee$ | Disyunción |
| $\wedge$ | Conjunción |
| $\rightarrow$ | Condicional |
| $\leftrightarrow$ | Bicondicional |

**Definición 2.1.1.** *Se dice que* `F` *es una fórmula si satisface la siguiente definición inductiva*

1. *Las variables proposicionales son fórmulas atómicas.*
2. *Si $F$ y $G$ son fórmulas, entonces $F \wedge G)$, $F \vee G)$, $F \rightarrow G)$, $\neg F$ y $(F \leftrightarrow G)$ son fórmulas.*

**Definición 2.1.2.** *Un predicado es una oración narrativa que puede ser verdadera o falsa.*

Nosotros trabajaremos con la lógica de primer orden. Para ello, debemos añadir a los conceptos ya introducidos de la lógica proposicional los siguientes elementos

1. Cuantificadores: $\forall$ (Universal) y $\exists$ (Existencial)
2. Símbolo de igualdad: $=$

3. Constantes: $a, b, \ldots, a_1, a_2, \ldots$
4. Predicados
5. Funciones

## 2.2. Representación de modelos

Trabajaremos con modelos a través de un dominio de entidades; en concreto, aquellas del módulo `Dominio`. Cada entidad de dicho módulo representa un sujeto. Cada sujeto cumplirá distintos predicados.

**Definición 2.2.1.** *Una interpretación es una aplicación $I : VP \to Bool$, donde $VP$ representa el conjunto de las variables proposicionales.*

Una interpretación toma valores para las variables proposicionales, y se evalúan en una fórmula. Determinando si la fórmula es verdadera o falsa. Se definirá más adelante mediante las funciones `valor` y `val`.

**Definición 2.2.2.** *Un modelo de una fórmula F es una interpretación en la que el valor de F es verdadero.*

```
module Modelo where
import Dominio
import PFH
```

A continuación damos un ejemplo de predicados lógicos para la clasificación botánica. La cual no es completa, pero da una idea de la potencia de Haskell para este tipo de uso.

Primero definimos los elementos que pretendemos clasificar, y que cumplirán los predicados. Para ello, definimos como función cada elemento a clasificar y le asociamos una entidad.

```
adelfas, aloeVera, boletus, cedro, chlorella, girasol, guisante, helecho,
  hepatica, jaramago, jazmin, lenteja, loto, magnolia, maiz, margarita,
  musgo, olivo, pino, pita, posidonia, rosa, sargazo, scenedesmus,
  tomate, trigo
  :: Entidades
adelfas     = U
aloeVera    = L
boletus     = W
cedro       = A
```

```
chlorella   = Z
girasol     = Y
guisante    = S
helecho     = E
hepatica    = V
jaramago    = X
jazmin      = Q
lenteja     = R
loto        = T
magnolia    = O
maiz        = F
margarita   = K
musgo       = D
olivo       = C
pino        = J
pita        = M
posidonia   = H
rosa        = P
sargazo     = I
scenedesmus = B
tomate      = N
trigo       = G
```

Una vez que ya tenemos todos los elementos a clasificar definidos, se procede a la interpretación de los predicados.

```
acuatica, algasVerdes, angiosperma, asterida, briofita, cromista,
  crucifera, dicotiledonea, gimnosperma, hongo, leguminosa,
  monoaperturada, monocotiledonea, rosida, terrestre,
  triaperturada, unicelular
  :: Entidades -> Bool
acuatica        = (‘pertenece‘ [B,H,I,T,Z])
algasVerdes     = (‘pertenece‘ [B,Z])
angiosperma     = (‘pertenece‘ [C,F,G,H,K,L,M,N,O,P,Q,R,S,T,U,X,Y])
asterida        = (‘pertenece‘ [C,K,N,Q,U,Y])
briofita        = (‘pertenece‘ [D,V])
cromista        = (‘pertenece‘ [I])
crucifera       = (‘pertenece‘ [X])
dicotiledonea   = (‘pertenece‘ [C,K,N,O,P,Q,R,S,T,U,X,Y])
gimnosperma     = (‘pertenece‘ [A,J])
hongo           = (‘pertenece‘ [W])
```

```
leguminosa      = (`pertenece` [R,S])
monoaperturada  = (`pertenece` [F,G,H,L,M,O])
monocotiledonea = (`pertenece` [F,G,H,L,M])
rosida          = (`pertenece` [P])
terrestre       =
  (`pertenece` [A,C,D,E,F,G,J,K,L,M,N,O,P,Q,R,S,U,V,W,X,Y])
triaperturada   = (`pertenece` [C,K,N,P,Q,R,S,T,U,X,Y])
unicelular      = (`pertenece` [B,Z])
```

Por ejemplo, podríamos comprobar si el `scenedesmus` es gimnosperma

```
ghci> gimnosperma scenedesmus
False
```

Esto nos puede facilitar establecer una jerarquía en la clasificación, por ejemplo `(espermatófitas)`; es decir, plantas con semillas.

```
espermatofitas :: Entidades -> Bool
espermatofitas x = angiosperma x || gimnosperma x
```

## 2.3. Lógica de predicados en Haskell

El contenido de esta sección se encuentra en el módulo `LPH`, en él se pretende dar representación a variables y fórmulas lógicas para la posterior evaluación de las mismas.

```
module LPH where
import Dominio
import Modelo
import Data.List
import Test.QuickCheck
```

Se define un tipo de dato para las variables.

```
type Nombre   = String

type Indice   = [Int]

data Variable = Variable Nombre Indice
  deriving (Eq,Ord)
```

Y para su representación en pantalla

```
instance Show Variable where
  show (Variable nombre [])  = nombre
  show (Variable nombre [i]) = nombre ++ show i
  show (Variable nombre is)  = nombre ++ showInts is
    where showInts []     = ""
          showInts [i]     = show i
          showInts (i:is') = show i ++ "_" ++ showInts is'
```

Ejemplos de definición de variables

```
x,y,z :: Variable
x = Variable "x" []
y = Variable "y" []
z = Variable "z" []
```

Empleando índices

```
a1,a2,a3 :: Variable
a1 = Variable "a" [1]
a2 = Variable "a" [2]
a3 = Variable "a" [3]
```

De manera que el resultado queda

```
ghci> a1
a1
```

A continuación se define un tipo de dato para las fórmulas

```
data Formula = Atomo Nombre [Variable]
             | Igual Variable Variable
             | Negacion Formula
             | Implica Formula Formula
             | Equivalente Formula Formula
             | Conjuncion [Formula]
             | Disyuncion [Formula]
             | ParaTodo Variable Formula
             | Existe Variable Formula
     deriving (Eq,Ord)
```

Y se emplea `show` para la visualización por pantalla.

```
instance Show Formula where
    show (Atomo str [])       = str
    show (Atomo str vs)       = str ++ show vs
    show (Igual t1 t2)        = show t1 ++ "≡" ++ show t2
    show (Negacion formula)   = '¬' : show formula
    show (Implica f1 f2)      = "(" ++ show f1 ++ "⟹" ++ show f2 ++ ")"
    show (Equivalente f1 f2)  = "(" ++ show f1 ++ "⟺" ++ show f2 ++ ")"
    show (Conjuncion [])      = "true"
    show (Conjuncion (f:fs))  = "(" ++ show f ++ "⋀" ++ show fs ++ ")"
    show (Disyuncion [])      = "false"
    show (Disyuncion (f:fs))  = "(" ++ show f ++ "⋁" ++ show fs ++ ")"
    show (ParaTodo v f)       = "∀" ++ show v ++ (' ': show f)
    show (Existe v f)         = "∃" ++ show v ++ (' ': show f)
```

Por ejemplo expresemos la propiedad `reflexiva` y la `simétrica`

```
reflexiva, simetrica :: Formula
reflexiva = ParaTodo x (Atomo "R" [x,x])
simetrica = ParaTodo x (ParaTodo y ( Atomo "R" [x,y] `Implica`
                                     Atomo "R" [y,x]))
```

```
ghci> reflexiva
∀x R[x,x]
ghci> simetrica
∀x ∀y (R[x,y]==>R[y,x])
```

## 2.4. Evaluación de fórmulas

Implementamos $s(x|d)$,mediante la función `(sustituye s x d v)`. $s(x|d)$ viene dado por la fórmula

$$\text{sustituye (s(t),x,d,v)} = \begin{cases} \text{d}, & \text{si x} = \text{v} \\ \text{s(v)}, & \text{en caso contrario} \end{cases}$$

donde `s` es una aplicación que asigna un valor a una variable. En Haskell se expresa mediante guardas

```
sustituye :: (Variable -> a) -> Variable -> a -> Variable -> a
sustituye s x d v | x == v     = d
                  | otherwise = s v
```

**Definición 2.4.1.** *Una asignación es una función $A$ : Variable $\rightarrow$ Universo que hace corresponder a cada variable un elemento del universo.*

Definimos una asignación arbitraria para los ejemplos

```
asignacion :: a -> Entidades
asignacion v = A
```

Ejemplos de la función `(sustituye s x d v)`

```
ghci> sustituye asignacion y B z
A
ghci> sustituye asignacion y B y
B
```

```
type Universo a = [a]

type Interpretacion a = String -> [a] -> Bool

type Asignacion a = Variable -> a
```

Definimos la función `(valor u i s form)` que calcula el valor de una fórmula en un universo `u`, con una interpretación `i` y la asignación `s`. Para ello vamos a definir previamente el valor de las interpretaciones para las distintas conectivas lógicas

| P | Q | $(P \wedge Q)$ | $(P \vee Q)$ | $(P \rightarrow Q)$ | $(P \leftrightarrow Q)$ |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 1 |

```
valor :: Eq a =>
         Universo a -> Interpretacion a -> Asignacion a
                    -> Formula -> Bool
```

```
valor _ i s (Atomo str vs)      = i str (map s vs)
valor _ _ s (Igual v1 v2)       = s v1 == s v2
valor u i s (Negacion f)        = not (valor u i s f)
valor u i s (Implica f1 f2)     = valor u i s f1 <= valor u i s f2
valor u i s (Equivalente f1 f2) = valor u i s f1 == valor u i s f2
valor u i s (Conjuncion fs)     = all (valor u i s) fs
valor u i s (Disyuncion fs)     = any (valor u i s) fs
valor u i s (ParaTodo v f)      = and [valor u i (sustituye s v d) f
                                       | d <- u]
valor u i s (Existe v f)        = or  [valor u i (sustituye s v d) f
                                       | d <- u]
```

Empleando las entidades y los predicados definidos en los módulos `Dominio` y `Modelo`, establecemos un ejemplo del valor de una interpretación en una fórmula.

Primero definimos la fórmula a interpretar

```
formula_1 :: Formula
formula_1 = ParaTodo x (Disyuncion [Atomo "P" [x],Atomo "Q" [x]])
```

```
ghci> formula_1
∀x (P[x]⋁[Q[x]])
```

Una interpretación para las propiedades `P` y `Q`, es comprobar si las plantas deben tener o no tener frutos.

```
interpretacion1 :: String -> [Entidades] -> Bool
interpretacion1 "P" [x] = angiosperma x
interpretacion1 "Q" [x] = gimnosperma x
interpretacion1 _ _     = False
```

Una segunda interpretación es si las plantas deben ser o no acuáticas o terrestres.

```
interpretacion2 :: String -> [Entidades] -> Bool
interpretacion2 "P" [x] = acuatica  x
interpretacion2 "Q" [x] = terrestre x
interpretacion2 _ _     = False
```

Tomamos como Universo todas las entidades menos la que denotamos `Inespecífico`

```
ghci> valor (take 26 entidades) interpretacion1 asignacion formula_1
False
ghci> valor (take 26 entidades) interpretacion2 asignacion formula_1
True
```

Por ahora siempre hemos establecido propiedades, pero podríamos haber definido relaciones binarias, ternarias, . . . , n–arias.

## 2.5. Términos funcionales

En la sección anterior todos los términos han sido variables. Ahora consideraremos funciones, entre ellas las constantes.

**Definición 2.5.1.** *Son términos en un lenguaje de primer orden:*

1. *Variables*
2. *Constantes*
3. $f(t_1, \ldots, t_n)$ *si* $t_i$ *son términos* $\forall i = 1, \ldots, n$

```
data Termino = Var Variable | Ter Nombre [Termino]
     deriving (Eq,Ord)
```

Algunos ejemplos de variables como términos

```
tx, ty, tz :: Termino
tx = Var x
ty = Var y
tz = Var z
```

Como hemos introducido, también tratamos con constantes, por ejemplo:

```
a, b, c, cero :: Termino
a    = Ter "a" []
b    = Ter "b" []
c    = Ter "c" []
cero = Ter "cero" []
```

Para mostrarlo por pantalla de manera comprensiva

```
instance Show Termino where
    show (Var v)    = show v
    show (Ter str []) = str
    show (Ter str ts) = str ++ show ts
```

Una función que puede resultar útil es `(esVariable x)`, que determina si un término es una variable

```
esVariable :: Termino -> Bool
esVariable (Var _) = True
esVariable _       = False
```

Ahora, creamos el tipo de dato `Form` de manera análoga a como lo hicimos en la sección anterior considerando simplemente variables, pero en este caso considerándo términos.

```
data Form = Atom String [Termino]
          | Ig Termino Termino
          | Neg Form
          | Impl Form Form
          | Equiv Form Form
          | Conj [Form]
          | Disy [Form]
          | PTodo Variable Form
          | Ex Variable Form
     deriving (Eq,Ord)
```

Y seguimos con la analogía y empleamos la función `show`

```
instance Show Form where
    show (Atom a [])   = a
    show (Atom f ts)   = f ++ show ts
    show (Ig t1 t2)    = show t1 ++ "≡" ++ show t2
    show (Neg form)    = '¬': show form
    show (Impl f1 f2)  = "(" ++ show f1 ++ "⟹" ++ show f2 ++ ")"
    show (Equiv f1 f2) = "(" ++ show f1 ++ "⟺" ++ show f2 ++ ")"
    show (Conj [])     = "true"
    show (Conj (f:fs)) = "(" ++ show f ++ "⋀" ++ show fs ++ ")"
    show (Disy [])     = "false"
```

```
    show (Disy (f:fs)) = "(" ++ show f ++ "⋁" ++ show fs ++ ")"
    show (PTodo v f)   = "∀" ++ show v ++ (' ': show f)
    show (Ex v f)      = "∃" ++ show v ++ (' ': show f)
```

Ejemplo de fórmulas

```
formula_2, formula_3 :: Form
formula_2 = PTodo x (PTodo y (Impl (Atom "R" [tx,ty])
                                   (Ex z (Conj [Atom "R" [tx,tz],
                                                Atom "R" [tz,ty]])))))
formula_3 = Impl (Atom "R" [tx,ty])
            (Ex z (Conj [Atom "R" [tx,tz],Atom "R" [tz,ty]]))
```

Quedando

```
ghci> formula_2
∀x ∀y (R[x,y]⟹∃z (R[x,z]⋀[R[z,y]]))
ghci> formula_3
(R[x,y]⟹∃z (R[x,z]⋀[R[z,y]]))
```

Para poder hacer las interpretaciones necesitamos primero una función auxiliar que extienda los valores de las variables a términos. Esta función es `(conversion s f str)`

```
conversion :: (Variable -> a) -> (String -> [a] -> a) -> (Termino -> a)
conversion s f (Var v)    = s v
conversion s f (Ter str ts) = f str (map (conversion s f) ts)
```

Siguiendo la linea de la sección anterior, definimos una función que determine el valor de una interpretación aplicada a una fórmula dada. Dicha función la denotamos por `(val u i f s form)`, en la que `u` denota el universo, `i` es la interpretación de las propiedades o relaciones, `f` es la interpretación del término funcional, `s` la asignación, y `form` una fórmula.

```
val :: Eq a =>
    [a] -> (String -> [a] -> Bool) -> (String -> [a] -> a)
        -> (Variable -> a)  -> Form -> Bool
val u i f s (Atom str ts) = i str (map (conversion s f) ts)
val u i f s (Ig t1 t2)    =
    conversion s f t1 == conversion s f t2
```

```
val u i f s (Neg g)       = not (val u i f s g)
val u i f s (Impl f1 f2)  =
    not (val u i f s f1 && not (val u i f s f2))
val u i f s (Equiv f1 f2) = val u i f s f1 == val u i f s f2
val u i f s (Conj fs)     = all (val u i f s) fs
val u i f s (Disy fs)     = any (val u i f s) fs
val u i f s (PTodo v g)   =
    and [ val u i f (sustituye s v d) g | d <- u]
val u i f s (Ex v g)      =
    or  [ val u i f (sustituye s v d) g | d <- u]
```

Veamos un ejemplo. Para ello tenemos que interpretar los elementos de una fórmula, por ejemplo la `formula 4`.

```
formula_4 :: Form
formula_4 = Ex x (Atom "R" [cero,tx])
```

```
ghci> formula_4
∃x R[cero,x]
```

En este caso tomamos como universo `u` los números naturales. Interpretamos `R` como la desigualdad $<$. Es decir, vamos a comprobar si es cierto que existe un número natural mayor que el 0.

```
interpretacion3:: String -> [Int] -> Bool
interpretacion3 "R" [x,y] = x < y
interpretacion3 _ _       = False
```

Interpretamos los símbolos; es decir, las interpretaciones de la `f`

```
interpretacionDef :: String -> [Int] -> Int
interpretacionDef "cero" []    = 0
interpretacionDef "s"    [i]   = succ i
interpretacionDef "mas"  [i,j] = i + j
interpretacionDef "por"  [i,j] = i * j
interpretacionDef _ _          = 0
```

Empleamos la asignación

```
asignacion1 :: Variable -> Int
asignacion1 x = 0
```

Quedando el ejemplo

```
ghci> val [0..] interpretacion3 interpretacionDef asignacion1 formula_4
True
```

**Nota :** Haskell es perezoso, así que podemos utilizar un universo infinito. Haskell no hace cálculos innecesarios, es decir, para cuando encuentra un elemento que cumple la propiedad.

### 2.5.1. Generadores

Para poder emplear el sistema de comprobación `QuickCheck`, necesitamos poder generar elementos aleatorios de los tipos de datos creados hasta ahora.

```
module Generadores where
import PFH
import Modelo
import LPH
import Dominio
import PTLP
import Test.QuickCheck
import Control.Monad
```

#### Generador de Nombres

```
abecedario :: Nombre
abecedario = "abcdefghijklmnopqrstuvwxyz"

genletra :: Gen Char
genletra = elements abecedario

genNombre :: Gen Nombre
genNombre = liftM (take 1) (listOf genletra)
```

### Generador de Índices

```
genNumero :: Gen Int
genNumero = choose (0,100)

genIndice :: Gen Indice
genIndice =  liftM (take 1) (listOf genNumero)
```

### Generador de Variables

```
generaVariable :: Gen Variable
generaVariable = liftM2 Variable (genNombre) (genIndice)
instance Arbitrary (Variable) where
    arbitrary = generaVariable
```

### Generador de Fórmulas

```
generaFormula :: Gen Formula
generaFormula = oneof [liftM2 Atomo genNombre (listOf generaVariable),
                       liftM  Negacion generaFormula,
                       liftM2 Implica generaFormula generaFormula,
                       liftM2 Equivalente generaFormula generaFormula,
                       liftM Conjuncion (listOf generaFormula),
                       liftM Disyuncion (listOf generaFormula),
                       liftM2 ParaTodo generaVariable generaFormula,
                       liftM2 Existe   generaVariable generaFormula]
instance Arbitrary (Formula) where
    arbitrary = generaFormula
```

### Generador de Términos

```
generaTermino :: Gen Termino
generaTermino = oneof [liftM  Var generaVariable,
                       liftM2 Ter genNombre (listOf generaTermino)]
instance Arbitrary (Termino) where
    arbitrary = generaTermino
```

### 2.5.2. Funciones útiles en el manejo de fórmulas

La función `varEnTerm` y `varEnTerms` devuelve las variables que aparecen en un término o en una lista de ellos.

```
varEnTerm :: Termino -> [Variable]
varEnTerm (Var v)    = [v]
varEnTerm (Ter str ts) = varEnTerms ts

varEnTerms :: [Termino] -> [Variable]
varEnTerms = nub . concatMap varEnTerm
```

**Nota 1:** La función `nub xs` elimina elementos repetidos en una lista `xs`, se encuentra en el paquete `Data.List`.

**Nota 2:** Se emplea un tipo de recursión cruzada entre funciones. Las funciones se llaman la una a la otra.

Por ejemplo

```
ghci> varEnTerm tx
[x]
ghci> varEnTerms [tx,ty,tz]
[x,y,z]
```

La función `varEnForm` devuelve una lista de las variables que aparecen en una fórmula.

```
varEnForm :: Form -> [Variable]
varEnForm (Atom str terms)    = varEnTerms terms
varEnForm (Ig term1 term2)    = nub (varEnTerm term1 ++ varEnTerm term2)
varEnForm (Neg form)          = varEnForm form
varEnForm (Impl form1 form2)  = nub (varEnForm form1 ++ varEnForm form2)
varEnForm (Equiv form1 form2) = nub (varEnForm form1 ++ varEnForm form2)
varEnForm (Conj forms)        = nub (concatMap varEnForm forms)
varEnForm (Disy forms)        = nub (concatMap varEnForm forms)
varEnForm (PTodo x form)      = nub (x : varEnForm form)
varEnForm (Ex x form)         = nub (x : varEnForm form)
```

Por ejemplo

```
ghci> varEnForm formula_2
[x,y,z]
ghci> varEnForm formula_3
[x,y,z]
ghci> varEnForm formula_4
[x]
```

**Definición 2.5.2.** *Una variable es libre en una fórmula si tiene una aparición ligada a un cuantificador existencial o universal. ($\forall x, \exists x$)*

La función `variablesLibres` devuelve las variables libres de una formula dada.

```
variablesLibres :: Form -> [Variable]
variablesLibres (Atom str terms)    = varEnTerms terms
variablesLibres (Ig term1 term2)    =
    nub (varEnTerm term1 ++ varEnTerm term2)
variablesLibres (Neg form)          = variablesLibres form
variablesLibres (Impl form1 form2)  =
    nub (variablesLibres form1 ++ variablesLibres form2)
variablesLibres (Equiv form1 form2) =
    nub (variablesLibres form1 ++ variablesLibres form2)
variablesLibres (Conj forms)        =
    nub (concatMap variablesLibres forms)
variablesLibres (Disy forms)        =
    nub (concatMap variablesLibres forms)
variablesLibres (PTodo x form)     = delete x (variablesLibres form)
variablesLibres (Ex x form)        = delete x (variablesLibres form)
```

Se proponen varios ejemplos

```
ghci> variablesLibres formula_2
[]
ghci> variablesLibres formula_3
[x,y]
ghci> variablesLibres formula_4
[]
```

**Definición 2.5.3.** *Una fórmula abierta es una fórmula con variables libres.*

La función `formulaAbierta` determina si una fórmula dada es abierta.

```
formulaAbierta :: Form -> Bool
formulaAbierta = not. null . variablesLibres
```

Como acostumbramos, ponemos algunos ejemplos

```
ghci> formulaAbierta formula_2
False
ghci> formulaAbierta formula_3
True
ghci> formulaAbierta formula_4
False
```

.

# Capítulo 3

# Prueba de teoremas en lógica de predicados

Este capítulo pretende aplicar métodos de tableros para la demostración de teoremas en lógica de predicados. El contenido de este capítulo se encuentra en el módulo `PTLP`.

## 3.1. Sustitución

**Definición 3.1.1.** *Una sustitución es una aplicación* $S$ : *Variable* $\rightarrow$ *Termino.*

```
module PTLP where
import Data.List
import LPH
import PFH
import Test.QuickCheck
```

Hemos importado la librería `Debug.Trace` porque emplearemos la función `trace`. Esta función tiene como argumentos una cadena de caracteres, una función, y un valor sobre el que se aplica la función. Por ejemplo

```
ghci> trace ("aplicando even a x = " ++ show 3) (even 3)
aplicando even a x = 3
False
```

**Definición 3.1.2.** *Una variable* $x$ *es ligada en una fórmula cuando tiene una aparición de la forma* $\forall x$ *o* $\exists x$.

En la lógica de primer orden, aquellas variables que están ligadas, a la hora de emplear el método de tableros, es necesario sustituirlas por términos. Para ello definimos un nuevo tipo de dato

```
type Sust = [(Variable, Termino)]
```

Este nuevo tipo de dato es una asociación de la variable con el término mediante 2-uplas. Denotamos el elemento identidad de la sustitución como `identidad`

```
identidad :: Sust
identidad = []
```

Para que la sustitución sea correcta, debe ser lo que denominaremos como apropiada. Para ello eliminamos aquellas sustituciones que dejan la variable igual.

```
hacerApropiada :: Sust -> Sust
hacerApropiada xs = [x | x <- xs, Var (fst x) /= snd x]
```

Como la sustitución es una aplicación, podemos distinguir `dominio` y `recorrido`.

```
dominio :: Sust -> [Variable]
dominio = map fst

recorrido :: Sust -> [Termino]
recorrido = map snd
```

Posteriormente, se define una función que haga una sustitución de una variable concreta. La denotamos `(sustituyeVar sust var)`

```
sustituyeVar :: Sust -> Variable -> Termino
sustituyeVar [] y                    = Var y
sustituyeVar ((x,x'):xs) y | x==y    = x'
                           | otherwise = sustituyeVar xs y
```

Ahora aplicando una recursión entre funciones, podemos hacer sustituciones basándonos en los términos, mediante las funciones `(susTerm xs t)` y `(susTerms sust ts)`

```
susTerm :: Sust -> Termino -> Termino
susTerm xs (Var y) = sustituyeVar xs y
susTerm xs (Ter n ts) = Ter n (susTerms xs ts)

susTerms :: Sust -> [Termino] -> [Termino]
susTerms = map . susTerm
```

Finalmente, esta construcción nos sirve para generalizar a cualquier fórmula. Con este fin definimos `(sustitucionForm xs f)`, donde `xs` representa la lista de pares que definen las sustituciones, y `f` la fórmula.

```
sustitucionForm :: Sust -> Form -> Form
sustitucionForm xs (Atom a ts)  = Atom a (susTerms xs ts)
sustitucionForm xs (Neg f) = Neg (sustitucionForm xs f)
sustitucionForm xs (Impl f1 f2)  =
    Impl (sustitucionForm xs f1) (sustitucionForm xs f2)
sustitucionForm xs (Equiv f1 f2) =
    Equiv (sustitucionForm xs f1) (sustitucionForm xs f2)
sustitucionForm xs (Conj fs)     = Conj (sustitucionForms xs fs)
sustitucionForm xs (Disy fs)     = Disy (sustitucionForms xs fs)
sustitucionForm xs (PTodo v f)   = PTodo v (sustitucionForm xs' f)
    where
      xs' = [ x | x <- xs, fst x /= v]
sustitucionForm xs (Ex v f)      = Ex v (sustitucionForm xs' f)
    where
      xs' = [ x | x <- xs, fst x /= v]
```

Se puede generalizar a una lista de fórmulas mediante la funcion `sustitucionForms xs fs`. La hemos necesitado en la definición de la función anterior, pues las conjunciones y disyunciones trabajan con listas de fórmulas.

```
sustitucionForms :: Sust -> [Form] -> [Form]
sustitucionForms xs fs = map (sustitucionForm xs) fs
```

Nos podemos preguntar si la sustitución conmuta con la composición. Para ello definimos la función `(composicion xs ys)`

```
composicion :: Sust -> Sust -> Sust
composicion xs ys =
```

```
    (hacerApropiada [ (y,(susTerm xs y')) | (y,y') <- ys ])
    ++
    [ x | x <- xs, fst x `notElem` (dominio ys)]
```

```
composicionConmutativa :: Sust -> Sust -> Bool
composicionConmutativa xs ys =
    composicion xs ys == composicion ys xs
```

## 3.2. Unificación

Y comprobando con QuickCheck, no lo es

```
ghci> quickCheck composicionConmutativa
*** Failed! Falsifiable (after 4 tests and 3 shrinks)
```

**Definición 3.2.1.** *Una unificación de las variables $x_1$ y $x_2$ es una sustitución $S$ tal que $S(x_1) = S(x_2) = t$.*

```
unificacionTerminos :: Termino -> Termino -> [Sust]
unificacionTerminos (Var x) (Var y)   | x==y      = [identidad]
                                      | otherwise = [[(x,Var y)]]
unificacionTerminos (Var x) t         = [ [(x,t)] | x `notElem` varEnTerm t]
unificacionTerminos t (Var y)         = [ [(y,t)] | y `notElem` varEnTerm t]
unificacionTerminos (Ter a ts) (Ter b rs) =
    [ u | a==b, u <- unificaTermLista ts rs ]
```

**Definición 3.2.2.** *`unificaTermLista` es una función que se aplica a dos listas dadas, tal que*

1. *Si las dos listas son vacías, devuelve la `identidad` de la sustitución.*
2. *La unificación de las listas $t_1, \ldots, t_n$ y $r_1, \ldots, r_n$ es el resultado de ejecutar la composición de unificaciones.*

```
unificaTermLista :: [Termino] -> [Termino] -> [Sust]
unificaTermLista [] []         = [identidad]
unificaTermLista [] (r:rs)     = []
unificaTermLista (t:ts) []     = []
unificaTermLista (t:ts) (r:rs) =
    [ composicion u1 u2 | u1 <- unificacionTerminos t r,
     u2 <- unificaTermLista (susTerms u1 ts) (susTerms u1 rs) ]
```

# Bibliografía


[1] J. Alonso. Temas de programación funcional. Technical report, Univ. de Sevilla, 2015.

[2] J. Alonso. Temas de Lógica matemática y fundamentos. Technical report, Univ. de Sevilla, 2015.

[3] A. S. Mena. Beginning in Haskell. Technical report, Utrecht University, 2014.

[4] J. van Eijck. Computational semantics and type theory. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, 2003.

[5] Y. A. P. Yu. L. Yershov. Lógica matemática. Technical report, URSS, 1990.

# Índice alfabético