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 19 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 λ

Cuando hablamos de notación lambda simplemente nos referimos por ejemplo a expresiones del tipo `\x ->x+2`. La notación viene del λ *Calculus* y se escribiría λ`x. x+2`. Los diseñadores de Haskell tomaron el símbolo \ debido a su parecido con λ 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,*]
```

1.1.7. Generador de tipos en Haskell: Descripción de funciones

En esta sección se introducirán y describirán funciones útiles en la generación de ejemplos en tipos de datos abstractos. Estos generadores son útiles para las comprobación de propiedades con QuickCheck .

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

Primero, debemos conocer la lógica proposicional. 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 $\neg F$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$ 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

El contenido de esta sección se encuentra en el módulo `Modelo`.

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

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.

Posteriormente, se definirá un modelo lógico. Aquí empleamos el término modelo como una representación de la realidad. En secciones posteriores estos modelos serán posibles interpretaciones para fórmulas.

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
```

Definición 2.4.2. *Una estructura del lenguaje es un par $\mathcal{I} = (\mathcal{U}, I)$ tal que*
1. *$\mathcal{I}$ es un conjunto no vacío, denominado universo.*
2. *I es una función `Símbolos` $\rightarrow$ `Símbolos`*

Definición 2.4.3. *Una interpretación de una estructura del lenguaje es un par $(\mathcal{I}, A)$ formado por una estructura del lenguaje y una asignación A.*

Definimos los tipos de datos relativos a los elementos de la estructura del lenguaje.

```
type Universo a = [a]

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

type Asignacion a = Variable -> a
```

Definición 2.4.4. *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 2.4.5. *Un modelo de una fórmula `F` es una interpretación en la que el valor de `F` es verdadero.*

Definición 2.4.6. *Una fórmula es válida si toda estructura es modelo de la fórmula.*

Definición 2.4.7. *Una fórmula es satisfacible si existe alguna interpretación para la que sea verdadera, es decir, algún modelo.*

Definición 2.4.8. *Una fórmula es insatisfacible si no tiene ningún modelo.*

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	$P \wedge Q$	$P \vee Q$	$P \to Q$	$P \leftrightarrow Q$
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 -> InterpretacionR 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 considerando términos.

```
data Form = Atom Nombre [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]]))
```

La interpretación de los símbolos de funciones es

```
type InterpretacionF a = String -> [a] -> a
```

Para poder hacer las interpretaciones necesitamos primero una función auxiliar para calcular el valor de los términos. Esta función es `(valorT s f str)`

```
valorT :: InterpretacionF a -> Asignacion a -> Termino -> a
valorT i a (Var v)    = a v
valorT i a (Ter f ts) = i f (map (valorT i a) ts)
```

Una interpretación es un par formado por las interpretaciones de los símbolos de relación y la de los símbolos de función.

```
type Interpretacion a = (InterpretacionR a, InterpretacionF a)
```

Siguiendo la línea de la sección anterior, definimos una función que determine el valor de una fórmula. 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.

```
valorF :: Eq a => Universo a -> Interpretacion a -> Asignacion a
                              -> Form -> Bool
valorF u (iR,iF) a (Atom r ts) =
  iR r (map (valorT iF a) ts)
valorF u (_,iF) a (Ig t1 t2) =
  valorT iF a t1 == valorT iF a t2
valorF u i a (Neg g) =
  not (valorF u i a g)
valorF u i a (Impl f1 f2) =
  valorF u i a f1 <= valorF u i a f2
valorF u i a (Equiv f1 f2) =
  valorF u i a f1 == valorF u i a f2
valorF u i a (Conj fs) =
  all (valorF u i a) fs
valorF u i a (Disy fs) =
  any (valorF u i a) fs
valorF u i a (PTodo v g) =
  and [valorF u i (sustituye a v d) g | d <- u]
valorF u i a (Ex v g)  =
  or  [valorF u i (sustituye a 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. Por tanto, la interpretación de los símbolos de relación es

```
interpretacionR1 :: String -> [Int] -> Bool
interpretacionR1 "R" [x,y] = x < y
interpretacionR1 _ _       = False
```

La interpretación de los símbolos de función es

```
interpretacionF1 :: String -> [Int] -> Int
interpretacionF1 "cero" []    = 0
interpretacionF1 "s"    [i]   = succ i
interpretacionF1 "mas"  [i,j] = i + j
interpretacionF1 "por"  [i,j] = i * j
interpretacionF1 _ _          = 0
```

Empleamos la asignación

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

Quedando el ejemplo

```
ghci> valorF [0..] (interpretacionR1,interpretacionF1) 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 Test.QuickCheck
import Control.Monad
```

Generador de Nombres

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

genLetra :: Gen Char
genLetra = elements abecedario
```

Ejemplo de generación de letras

```
ghci> sample genLetra
'w'
'r'
'l'
'o'
'u'
'z'
'f'
'x'
'k'
'q'
'b'
```

```
genNombre :: Gen Nombre
genNombre = liftM (take 1) (listOf1 genLetra)
```

Se puede definir `genNombre` como sigue

```
genNombre2 :: Gen Nombre
genNombre2 = do
  c <- elements ['a'..'z']
  return [c]
```

Ejemplo de generación de nombres

```
ghci> sample genNombre2
"z"
"u"
"j"
"h"
"v"
"w"
"v"
"b"
"e"
"d"
"s"
```

Generador de Índices

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

genIndice :: Gen Indice
genIndice =  liftM (take 1) (listOf1 genNumero)
```

Ejemplo

```
ghci> sample genIndice
[98]
[62]
[50]
[89]
[97]
[6]
[14]
[87]
[14]
[92]
[1]
```

Generador de variables

```
generaVariable :: Gen Variable
generaVariable = liftM2 Variable (genNombre) (genIndice)

instance Arbitrary (Variable) where
    arbitrary = generaVariable
```

Ejemplo

```
ghci> sample generaVariable
q10
e5
m97
n92
h15
a52
c58
s74
t30
g78
i75
```

Generador de Fórmulas

```

instance Arbitrary (Formula) where
    arbitrary = sized formula
        where
          formula 0 = liftM2 Atomo genNombre (listOf generaVariable)
          formula n = oneof [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]
              where
                generaFormula = formula (n-1)
```

Generador de Términos

```
instance Arbitrary (Termino) where
    arbitrary = sized termino
        where
          termino 0 = liftM Var generaVariable
          termino n = liftM2 Ter genNombre (listOf generaTermino)
              where
              generaTermino = termino (n-1)
```

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 _ 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 _ ts)   = varEnTerms ts
varEnForm (Ig t1 t2)    = nub (varEnTerm t1 ++ varEnTerm t2)
varEnForm (Neg f)       = varEnForm f
varEnForm (Impl f1 f2)  = varEnForm f1 `union` varEnForm f2
varEnForm (Equiv f1 f2) = varEnForm f1 `union` varEnForm f2
varEnForm (Conj fs)     = nub (concatMap varEnForm fs)
varEnForm (Disy fs)     = nub (concatMap varEnForm fs)
varEnForm (PTodo x f)   = nub (x : varEnForm f)
varEnForm (Ex x f)      = nub (x : varEnForm f)
```

Por ejemplo

```
varEnForm formula_2  ==  [x,y,z]
varEnForm formula_3  ==  [x,y,z]
varEnForm formula_4  ==  [x]
```

Definición 2.5.2. *Una variable es libre en una fórmula si no tiene ninguna aparición ligada a un cuantificador existencial o universal.* $(\forall x, \exists x)$

La función `(variablesLibres f` devuelve las variables libres de la fórmula `f`.

```
variablesLibres :: Form -> [Variable]
variablesLibres (Atom _ ts) =
  varEnTerms ts
variablesLibres (Ig t1 t2) =
  varEnTerm t1 `union` varEnTerm t2
variablesLibres (Neg f) =
  variablesLibres f
variablesLibres (Impl f1 f2)  =
  variablesLibres f1 `union` variablesLibres f2
variablesLibres (Equiv f1 f2) =
  variablesLibres f1 `union` variablesLibres f2
variablesLibres (Conj fs) =
  nub (concatMap variablesLibres fs)
variablesLibres (Disy fs) =
  nub (concatMap variablesLibres fs)
variablesLibres (PTodo x f) =
  delete x (variablesLibres f)
variablesLibres (Ex x f) =
  delete x (variablesLibres f)
```

Se proponen varios ejemplos

```
variablesLibres formula_2  ==  []
variablesLibres formula_3  ==  [x,y]
variablesLibres formula_4  ==  []
```

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

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

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

Como acostumbramos, ponemos algunos ejemplos

```
formulaAbierta formula_2  ==  False
formulaAbierta formula_3  ==  True
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

```
module PTLP where
import LPH
import Data.List
import Test.QuickCheck -- Para ejemplos
import Generadores     -- Para ejemplos

```

Definición 3.1.1. *Una variable x está ligada en una fórmula cuando tiene una aparición de la forma $\forall x$ o $\exists x$.*

Definición 3.1.2. *Una sustitución es una aplicación S : $Variable \to Termino$.*

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 pares. 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 hace la 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 s (Var y)    = sustituyeVar s y
susTerm s (Ter f ts) = Ter f (susTerms s ts)

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

Finalmente, esta construcción nos sirve para generalizar a cualquier fórmula. Con este fin definimos `(sustitucionForm s f)`, donde `s` representa la susticución y `f` la fórmula.

```
sustitucionForm :: Sust -> Form -> Form
sustitucionForm s (Atom r ts) =
  Atom r (susTerms s ts)
sustitucionForm s (Ig t1 t2) =
```

```
  Ig (susTerm s t1) (susTerm s t2)
sustitucionForm s (Neg f) =
  Neg (sustitucionForm s f)
sustitucionForm s (Impl f1 f2) =
  Impl (sustitucionForm s f1) (sustitucionForm s f2)
sustitucionForm s (Equiv f1 f2) =
  Equiv (sustitucionForm s f1) (sustitucionForm s f2)
sustitucionForm s (Conj fs) =
  Conj (sustitucionForms s fs)
sustitucionForm s (Disy fs) =
  Disy (sustitucionForms s fs)
sustitucionForm s (PTodo v f) =
  PTodo v (sustitucionForm s' f)
  where s' = [x | x <- s, fst x /= v]
sustitucionForm s (Ex v f) =
  Ex v (sustitucionForm s' f)
  where s' = [x | x <- s, fst x /= v]
```

Se puede generalizar a una lista de fórmulas mediante la funcion `(sustitucionForms s 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 s = map (sustitucionForm s)
```

Nos podemos preguntar si la sustitución conmuta con la composición. Para ello definimos la función `(composicion s1 s2)`

```
composicion :: Sust -> Sust -> Sust
composicion s1 s2 =
    hacerApropiada [(y,susTerm s1 y') | (y,y') <- s2 ] ++
    [x | x <- s1, fst x `notElem` dominio s2]
```

Por ejemplo,

```
ghci> composicion [(x,tx)] [(y,ty)]
[(x,x)]
ghci> composicion [(x,tx)] [(x,ty)]
[(x,y)]
```

```
composicionConmutativa :: Sust -> Sust -> Bool
composicionConmutativa s1 s2 =
  composicion s1 s2 == composicion s2 s1
```

Y comprobando con QuickCheck, no lo es

```
ghci> quickCheck composicionConmutativa
*** Failed! Falsifiable (after 3 tests and 1 shrink):
[(i3,n)]
[(c19,i)]
```

Un contraejemplo más claro es

```
ghci> composicion [(x,tx)] [(y,ty)]
[(x,x)]
ghci> composicion  [(y,ty)] [(x,tx)]
[(y,y)]
ghci> composicion  [(y,ty)] [(x,tx)] == composicion [(x,tx)] [(y,ty)]
False
```

Nota: Las comprobaciones con QuickCheck emplean código del módulo `Generadores`.

3.2. Unificación

Definición 3.2.1. *Un unificador de dos términos t_1 y t_2 es una sustitución S tal que $S(t_1) = S(t_2)$.*

```
unificadoresTerminos :: Termino -> Termino -> [Sust]
unificadoresTerminos (Var x) (Var y)
  | x == y    = [identidad]
  | otherwise = [[(x,Var y)]]
unificadoresTerminos (Var x) t =
  [[(x,t)] | x `notElem` varEnTerm t]
unificadoresTerminos t (Var y) =
  [[(y,t)] | y `notElem` varEnTerm t]
unificadoresTerminos (Ter f ts) (Ter g rs) =
  [u | f == g, u <- unificadoresListas ts rs]
```

El valor de `(unificadoresListas ts rs)` es un unificador de las listas de términos `ts` y `rs`; es decir, una sustitución `s` tal que si `ts = [t1,...,tn]` y `rs = [r1,...,rn]` entonces `s(t1) = s(r1),...,s(tn) = s(rn)`.

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

Por ejemplo

```
ghci> unificadoresListas [tx] [ty]
[[(x,y)]]
ghci> unificadoresListas [tx] [tx]
[[]]
```

3.3. Skolem

Definición 3.3.1. *La fórmula* `F` *está en forma de Skolem si es de la forma* $\forall x_1 \ldots \forall x_n G$*, donde* $n \geq 0$ *y* `G` *no tiene cuantificadores.*

Para alcanzar una fórmula en forma de Skolem emplearemos sustituciones y unificaciones. Además, necesitamos eliminar las equivalencias e implicaciones. Para ello definimos la equivalencia y equisatisfacibilidad entre fórmulas.

Definición 3.3.2. *Las fórmulas* `F` *y* `G` *son equivalentes si para toda interpretación valen lo mismo.*

Definición 3.3.3. *Las fórmulas* `F` *y* `G` *son equisatisfacibles si se cumple* (`F` *satisfacible* $\Leftrightarrow$ `G` *satisfacible*)

Definimos la función `(elimImpEquiv f)`, para obtener fórmulas equivalentes sin equivalencias ni implicaciones.

```
elimImpEquiv :: Form -> Form
elimImpEquiv (Atom f xs) =
  Atom f xs
elimImpEquiv (Ig t1 t2) =
  Ig t1 t2
elimImpEquiv (Equiv f1 f2) =
  Conj [ elimImpEquiv (Impl f1 f2),
         elimImpEquiv (Impl f2 f1)]
elimImpEquiv (Impl f1 f2) =
  Disy [ Neg f1, f2]
elimImpEquiv (Neg f) =
  Neg (elimImpEquiv f)
elimImpEquiv (Disy fs) =
  Disy (map elimImpEquiv fs)
elimImpEquiv (Conj fs) =
  Conj (map elimImpEquiv fs)
elimImpEquiv (PTodo x f) =
  PTodo x (elimImpEquiv f)
elimImpEquiv (Ex x f) =
  Ex x (elimImpEquiv f)
```

Empleamos las fórmulas 2,3 y 4 ya definidas anteriormenta como ejemplo:

```
ghci> formula_2
∀x ∀y (R[x,y]⟹∃z (R[x,z]⋀[R[z,y]]))
ghci> elimImpEquiv 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]]))
ghci> elimImpEquiv formula_3
(¬R[x,y]⋁[∃z (R[x,z]⋀[R[z,y]])])
ghci> formula_4
∃x R[cero,x]
ghci> elimImpEquiv formula_4
∃x R[cero,x]
```

Finalmente, definamos una cadena de funciones, para finalizar con `(skolem f)` que transforma `f` a su forma de Skolem.

```
skol :: Int -> [Variable] -> Termino
skol k vs = Ter ("sk" ++ show k) [Var x | x <- vs]
```

Definimos la función `(skf f vs pol k)`, donde

1. `f` es la fórmula que queremos convertir.
2. `vs` es la lista de los cuantificadores (son necesarios en la recursión).
3. `pol` es la polaridad, es de tipo `Bool`.
4. `k` es de tipo `Int` y sirve como idetificador de la forma de Skolem.

```
skf :: Form -> [Variable] -> Bool -> Int -> (Form,Int)
skf (Atom n ts) _ _ k =
  (Atom n ts,k)
skf (Conj fs) vs pol k =
  (Conj fs',j)
  where (fs',j) = skfs fs vs pol k
skf (Disy fs) vs pol k =
  (Disy fs', j)
  where (fs',j) = skfs fs vs pol k
skf (PTodo x f) vs True k =
  (PTodo x f',j)
  where vs'    = insert x vs
        (f',j) = skf f vs' True k
skf (PTodo x f) vs False k =
  skf (sustitucionForm b f) vs False (k+1)
  where b = [(x,skol k vs)]
skf (Ex x f) vs True k =
  skf (sustitucionForm b f) vs True (k+1)
  where b = [(x,skol k vs)]
skf (Ex x f) vs False k =
```

```
  (Ex x f',j)
  where vs' = insert x vs
        (f',j) = skf f vs' False k
skf (Neg f) vs pol k =
  (Neg f',j)
  where (f',j) = skf f vs (not pol) k
```

donde la skolemización de una lista está definida por

```
skfs :: [Form] -> [Variable] -> Bool -> Int -> ([Form],Int)
skfs [] _ _ k        = ([],k)
skfs (f:fs) vs pol k = (f':fs',j)
  where (f',j1) = skf  f  vs pol k
        (fs',j) = skfs fs vs pol j1
```

La skolemizacoón de una fórmula sin equivalencias ni implicaciones se define por

```
sk :: Form -> Form
sk f = fst (skf f [] True 0)
```

La función `(skolem f)` devuelve la forma de Skolem de la fórmula `f`.

```
skolem :: Form -> Form
skolem  = sk . elimImpEquiv
```

Por ejemplo,

```
ghci> sk formula_2
∀x ∀y (¬R[x,y]⋁[(R[x,sk0[x,y]]⋀[R[sk0[x,y],y]])])
ghci> skolem formula_3
(¬R[x,y]⋁[(R[x,sk0]⋀[R[sk0,y]])])
ghci> skolem formula_4
R[cero,sk0]
```

3.4. Tableros semánticos

Definición 3.4.1. *Una fórmula o conjunto de fórmulas es consistente si tiene algún modelo. En caso contrario, se denomina inconsistente.*

La idea de obtener fórmulas equivalentes, nos hace introducir los tipos de fórmulas alfa, beta, gamma y delta. No son más que equivalencias ordenadas, por orden teórico en el que se pueden acometer, para una simplicación eficiente de una fórmula a otra cuyas únicas conectivas lógicas sean disyunciones y conjunciones.

- Fórmulas alfa

$\neg(F_1 \to F_2)$	$F_1 \wedge F_2$
$\neg(F_1 \vee F_2)$	$F_1 \wedge \neg F_2$
$F_1 \leftrightarrow F_2$	$(F_1 \to F_2) \wedge (F_2 \to F_1)$

Las definimos en Haskell

```
alfa :: Form -> Bool
alfa (Conj _) = True
alfa (Neg (Disy _)) = True
alfa _ = False
```

- Fórmulas beta

$F_1 \to F_2$	$\neg F_1 \vee F_2$
$\neg(F_1 \wedge F_2)$	$\neg F_1 \vee \neg F_2$
$\neg(F_1 \leftrightarrow F_2)$	$\neg(F_1 \to F_2) \vee (\neg F_2 \to F_1)$

Las definimos en Haskell

```
beta :: Form -> Bool
beta (Disy _)       = True
beta (Neg (Conj _)) = True
beta _              = False
```

- Fórmulas gamma

$\forall x F$	$F[x/t]$
$\neg\exists x F$	$\neg F[x/t]$

Notar que t es un término básico.

Las definimos en Haskell

```
gamma :: Form -> Bool
gamma (PTodo _ _)    = True
gamma (Neg (Ex _ _)) = True
gamma _              = False
```

- Fórmulas delta

$\exists x F$	$F[x/a]$
$\neg\forall F$	$\neg F[x/a]$

Notar que a es una constante nueva.

Nota 3.4.1. En las tablas, cada elemento de una columna es equivalente a su análogo en la otra columna.

Mediante estas equivalencias se procede a lo que se denomina método de los tableros semánticos. Uno de los objetivos del método de los tableros es determinar si una fórmula es inconsistente, así como la búsqueda de modelos.

Definición 3.4.2. *Un literal es un átomo o la negación de un átomo.*

Lo definimos en haskell

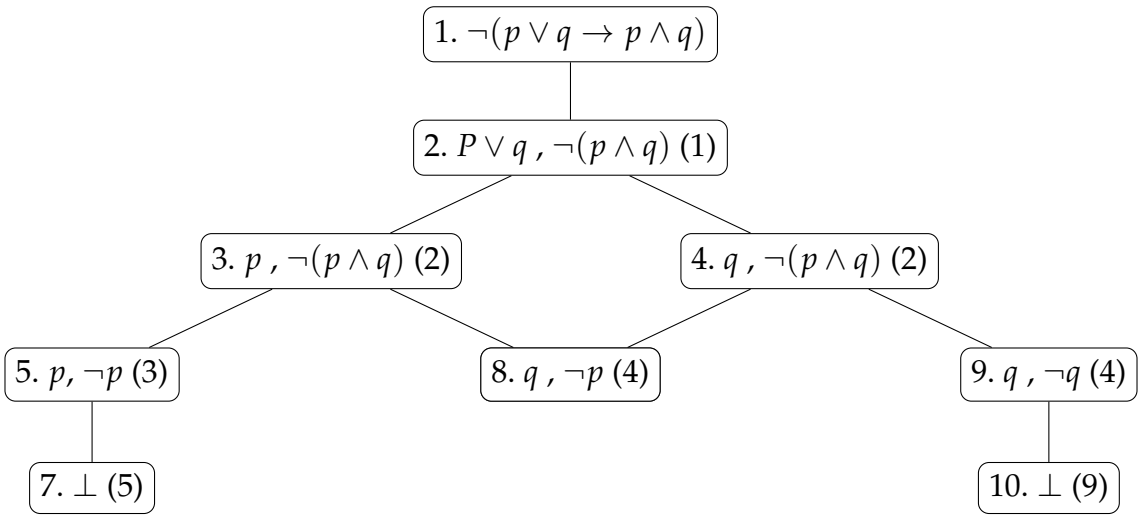
```
atomo, negAtomo, literal :: Form -> Bool
atomo (Atom n ts)          = True
atomo _                    = False
negAtomo (Neg (Atom n ts)) = True
negAtomo  _                = False
literal f = atomo f || negAtomo f
```

Definición 3.4.3. *Se dice que una hoja es cerrada si contiene una fórmula y su negación. Se representa* $\perp$

Definición 3.4.4. *Se dice que una hoja es abierta si es un conjunto de literales y no contiene un literal y su negación.*

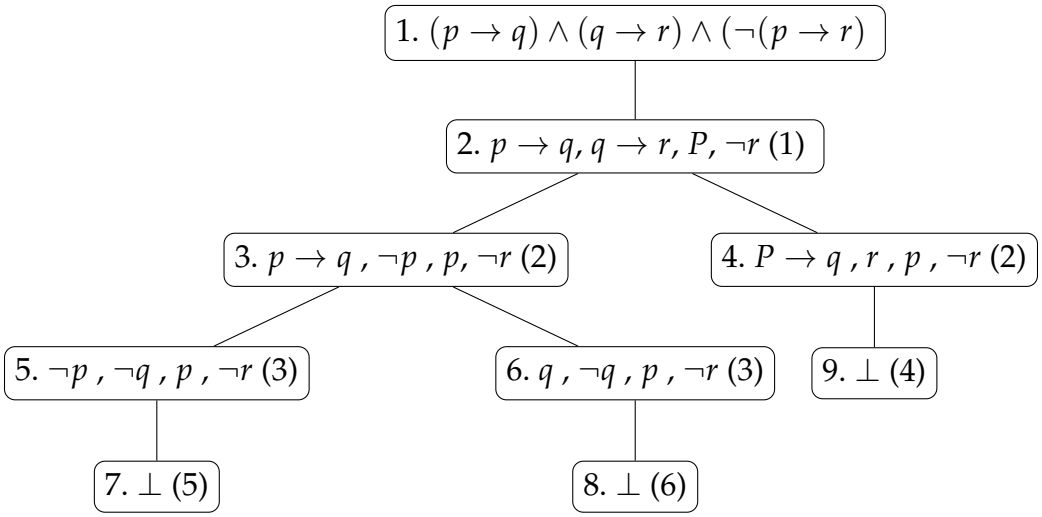
Definición 3.4.5. *Un tablero completo es un tablero tal que todas sus hojas son abiertas o cerradas.*

Ejemplo de tablero completo



Definición 3.4.6. *Un tablero es cerrado si todas sus hojas son cerradas.*

Un ejemplo de tablero cerrado es



.

Teorema 3.4.1. *Si una fórmula F es consitente, entonces cualquier tablero de F tendrá ramas abiertas.*

Nuestro objetivo es definir en Haskell un método para el cálculo de tableros semánticos. El contenido relativo a tableros semánticos se encuentra en el módulo `Tableros`.

```
module Tableros where
import PTLP
import LPH
import Debug.Trace
```

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
```

A lo largo de esta sección trabajaremos con fórmulas en su forma de Skolem. Definimos el tipo de dato `Nodo`

```
data Nodo = Nd Indice [Termino] [Termino] [Form]
           deriving Show
```

Para finalmente, definir los tableros como una lista de nodos.

```
type Tablero = [Nodo]
```

Necesitamos poder reconocer las dobles negaciones

```
dobleNeg (Neg (Neg f)) = True
dobleNeg _             = False
```

Una función auxiliar de conversión de literales a términos.

```
literalATer :: Form -> Termino
literalATer (Atom n ts)  = Ter n ts
literalATer (Neg (Atom n ts)) = Ter n ts
```

Definimos la función `(componentes f)` que determina los componentes de una fórmula `f`.

```
componentes :: Form -> [Form]
componentes (Conj fs) = fs
componentes (Disy fs) = fs
componentes (Neg (Conj fs)) = [Neg f | f <- fs]
componentes (Neg (Disy fs)) = [Neg f | f <- fs]
componentes (Neg (Neg f)) = [f]
componentes (PTodo x f) = [f]
componentes (Neg (Ex x f)) = [Neg f]
```

Definimos la función `(varLigada f)` que devuelve la variable ligada de la fórmula `f`

```
varLigada :: Form -> Variable
varLigada (PTodo x f)    = x
varLigada (Neg (Ex x f)) = x
```

Definimos la función `(descomponer f)` que determina los cuantificadores universales de `f`.

```
descomponer :: Form -> ([Variable],Form)
descomponer f = descomp [] f where
    descomp xs f = if gamma f then descomp (xs ++ [x]) f' else (xs,f)
        where x = varLigada f
              [f'] = componentes f
```

Por ejemplo

```
ghci> descomponer formula_2
([x,y],(R[x,y]⟹∃z (R[x,z]⋀[R[z,y]])))
ghci> descomponer formula_3
([],(R[x,y]⟹∃z (R[x,z]⋀[R[z,y]])))
ghci> descomponer formula_4
([],∃x R[cero,x])
```

Definimos (`ramificacion nodo`) que ramifica un nodo.

`ramificacion`

```
ramificacion :: Nodo -> Tablero
ramificacion (Nd i pos neg []) = [Nd i pos neg []]
ramificacion (Nd i pos neg (f:fs))
    | atomo f = if elem (literalATer f) neg then []
                else [Nd i ((literalATer f):pos) neg fs]
    | negAtomo f = if elem (literalATer f) pos then []
                   else [Nd i pos ((literalATer f):neg) fs]
    | dobleNeg f = [Nd i pos neg ((componentes f) ++ fs)]
    | alfa f  = [Nd i pos neg ((componentes f) ++ fs)]
    | beta f  = [(Nd (i++[n]) pos neg (f':fs)) |
                (f',n) <- zip (componentes f) [0..]]
    | gamma f = [Nd i pos neg (f':(fs++[f]))]
    where
      (xs,g) = descomponer f
      b      = [((Variable nombre j),
                 Var (Variable nombre i)) |
                (Variable nombre j) <- xs]
      f'     = sustitucionForm b g
```

Debido a que pueden darse la infinitud de un árbol por las fórmulas gamma, definimos otra función (`ramificacionP k nodo`) que ramifica un nodo teniendo en cuenta la profundidad.

```
ramificacionP :: Int -> Nodo -> (Int,Tablero)
ramificacionP k nodo@(Nd i pos neg []) = (k,[nodo])
ramificacionP k (Nd i pos neg (f:fs))
    | atomo    f = if elem (literalATer f) neg then (k,[])
                   else (k,[Nd i ((literalATer f):pos) neg fs])
    | negAtomo f = if elem (literalATer f) neg then (k,[])
                   else (k,[Nd i pos ((literalATer f):neg) fs])
    | dobleNeg f = (k,[Nd i pos neg ((componentes f) ++ fs)])
    | alfa     f = (k,[Nd i pos neg ((componentes f) ++ fs)])
    | beta     f = (k,[(Nd (i++[n]) pos neg (f':fs)) |
                       (f',n) <- zip (componentes f) [0..] ])
    | gamma    f = (k-1, [Nd i pos neg (f':(fs++[f]))])
    where
      (xs,g) = descomponer f
      b      = [((Variable nombre j), Var (Variable nombre i)) |
                (Variable nombre j) <- xs]
      f'     = sustitucionForm b g
```

Definición 3.4.7. *Un nodo está completamente expandido si no se puede seguir ramificando*

Se define en Haskell

```
nodoExpandido :: Nodo -> Bool
nodoExpandido (Nd i pos neg []) = True
nodoExpandido _                 = False
```

Definimos la función `(expandeTablero n tab)` que desarrolla un tablero a una profundidad `n`.

```
expandeTablero :: Int -> Tablero -> Tablero
expandeTablero 0 tab = tab
expandeTablero _ []  = []
expandeTablero n (nodo:nodos)
    | nodoExpandido nodo = nodo:(expandeTablero n nodos)
    | otherwise =  if k == n then expandeTablero n (nuevoNodo ++ nodos)
                   else expandeTablero (n-1) (nodos ++ nuevoNodo)
    where (k,nuevoNodo) = ramificacionP n nodo
```

Para una visualización más gráfica, definimos `(expandeTableroG)` empleando la función `(trace)`.

```
expandeTableroG :: Int -> Tablero -> Tablero
expandeTableroG 0 tab = tab
expandeTableroG _ [] = []
expandeTableroG n (nodo:nodos)
    | nodoExpandido nodo = trace (show nodo ++ "\n\n")
                           (nodo:(expandeTableroG n nodos))
    | otherwise = if k == n then trace (show nodo ++ "\n\n")
                  (expandeTableroG k (nuevoNodo ++ nodos))
                  else trace (show nodo ++"\n\n")
                  (expandeTableroG (n-1) (nodos ++ nuevoNodo))
    where (k, nuevoNodo) = ramificacionP n nodo
```

Definimos la función `(compruebaNodo)` para comprobar si hay hoja cerrada.

```
compruebaNodo :: Nodo -> [Sust]
compruebaNodo (Nd _ pos neg _) =
    concat [ unificadoresTerminos p n | p <- pos,
                                        n <- neg ]
```

Capítulo 4

Apéndice: GitHub

En este apéndice se pretende introducir al lector en el empleo de GitHub, sistema remoto de versiones.

4.1. Crear una cuenta

El primer paso es crear una cuenta en la página web de GitHub, para ello `sign up` y se rellena el formulario.

4.2. Crear un repositorio

Mediante `New repository` se crea un repositorio nuevo. Un repositorio es una carpeta de trabajo. En ella se guardarán todas las versiones y modificaciones de nuestros archivos.

Necesitamos darle un nombre adecuado y seleccionar

1. En `Add .gitignore` se selecciona `Haskell`.
2. En `add a license` se selecciona `GNU General Public License v3.0`.

Finalmente `Create repository`

4.3. Conexión

Nuestro interés está en poder trabajar de manera local y poder tanto actualizar GitHub como nuestra carpeta local. Los pasos a seguir son

1. Generar una clave SSH mediante el comando

   ```
   ssh-keygen -t rsa -b 4096 -C "tuCorreo"
   ```

   Indicando una contraseña. Si queremos podemos dar una localización de guardado de la clave pública.

2. Añadir la clave a `ssh-agent`, mediante

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_rsa
```

3. Añadir la clave a nuestra cuenta. Para ello: `Setting` → `SSH and GPG keys` → `New SSH key`. En esta última sección se copia el contenido de

```
~/.ssh/id_rsa.pub
```

   por defecto. Podríamos haber puesto otra localización en el primer paso.
4. Se puede comprobar la conexión mediante el comando

```
ssh -T git@github.com
```

5. Se introducen tu nombre y correo

```
git config --global user.name "Nombre"
git config --global user.email "<tuCorreo>"
```

4.4. Pull y push

Una vez hecho los pasos anteriores, ya estamos conectados con GitHub y podemos actualizar nuestros ficheros. Nos resta aprender a actualizarlos.

1. Clonar el repositorio a una carpeta local:
   Para ello se ejecuta en una terminal

```
git clone <enlace que obtienes en el repositorio>
```

   El enlace que sale en el repositorio pinchando en `(clone or download)` y, eligiendo `(use SSH)`.
2. Actualizar tus ficheros con la versión de GitHub:
   En emacs se ejecuta `(Esc-x)+(magit-status)`. Para ver una lista de los cambios que están `(unpulled)`, se ejecuta en magit `remote update`. Se emplea `pull`,y se actualiza. `(Pull: origin/master)`
3. Actualizar GitHub con tus archivos locales:
   En emacs se ejecuta `(Esc-x)+(magit-status)`. Sale la lista de los cambios `(UnStages)`. Con la `(s)` se guarda, la `(c)+(c)` hace `(commit)`. Le ponemos un título y, finalmente `(Tab+P)+(P)` para hacer `(push)` y subirlos a GitHub.

4.5. Ramificaciones (“branches”)

Uno de los puntos fuertes de Git es el uso de ramas. Para ello, creamos una nueva rama de trabajo. En `(magit-status)`, se pulsa `b`, y luego `(c) (create a new branch`

`and checkout)`. Checkout cambia de rama a la nueva, a la que habrá que dar un nombre.

Se trabaja con normalidad y se guardan las modificaciones con `magit-status`. Una vez acabado el trabajo, se hace `(merge)` con la rama principal y la nueva.

Se cambia de rama `(branch...)` y se hace `(pull)` como acostumbramos.

Finalmente, eliminamos la rama mediante `(magit-status)` → `(b)` → `(k)` → `(Nombre de la nueva rama)`

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