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 31 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]).

Adaptar a nueva estructura.

[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
import Data.List
```

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 con otro tipo de dato como son cadenas de caracteres, que son tipo `[Char]` o `String`. `(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 es parecido a su expresión como conjunto:

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

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 1.1.1. En los distintos ejemplos hemos visto que se pueden componer funciones ya definidas.

Otro ejemplo que nos será importante es poder construir subconjuntos de una lista.

```
subconjuntos :: [t] -> [[t]]
subconjuntos [] = [[]]
subconjuntos (x:xs) = zss++[x:ys | ys <- zss]
    where zss = subconjuntos xs

subconjuntosTam :: Int -> [a] -> [[a]]
subconjuntosTam n xs =
    concat [permutations x | x <- subconjuntos xs, length x == n]
```

1.1.2. Funciones map y filter

Introducimos un par de funciones de mucha relevancia en el uso de listas en Haskell. Son funciones que se denominan de orden superior.

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 1.1.2. Hemos empleando composición de funciones para la definición de `(algun)` y `(todos)`. Se representa mediante `.`,y 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 1.1.3. $\otimes$ 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 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

Definición 1.1.1. Un **dominio de entidades** es un conjunto de individuos cuyas propiedades son objeto de estudio para una clasificación

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

Sintaxis y semántica de la lógica de primer orden

El **lenguaje de la lógica de primer orden** está compuesto por

1. Variables proposicionales $p, q, r, \ldots$
2. Conectivas lógicas:

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

3. Símbolos auxiliares $"("$, $")"$
4. Cuantificadores: $\forall$ (Universal) y $\exists$ (Existencial)
5. Símbolo de igualdad: $=$
6. Constantes: $a, b, \ldots, a_1, a_2, \ldots$
7. Símbolos de relación: $P, Q, R, \ldots$
8. Símbolos de función: $f, g, h \ldots$

2.1. Representación de modelos

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

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

La lógica de primer orden permite dar una representación al conocimiento. Nosotros 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 tendrá distintas propiedades.

En secciones posteriores se definirá un modelo lógico. Aquí empleamos el término modelo como una modelización o representación de la realidad.

Damos un ejemplo de predicados lógicos para la clasificación botánica. La cual no es completa, pero nos da una idea de la manera de una representación lógica.

Primero definimos los elementos que pretendemos clasificar, y que cumplirán los predicados. Con este fin, definimos una función para cada elemento del dominio de entidades.

```
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. Es decir, una clasificación de aquellos elementos que cumplen un cierto predicado.

Definición 2.1.1. Un **predicado** es una oración narrativa que puede ser verdadera o falsa.

```
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 `(espermatofitas)`; es decir, plantas con semillas.

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

2.2. Lógica de primer orden en Haskell

El contenido de esta sección se encuentra en el módulo `LPH`. Se pretende asentar las bases de la lógica de primer orden y su implementación en Haskell, con el objetivo de construir los cimientos para las posteriores implementaciones de algoritmos en los siguientes capítulos.

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

Los elementos básicos de las fórmulas en la lógica de primer orden, así como en la lógica proposicional son las variables.

Definimos un tipo de dato para las variables. Una variable estará compuesta por:

Un `nombre`, que será una lista de caracteres.

```
type Nombre = String
```

Un `índice`, lista de enteros.

```
type Indice = [Int]
```

Quedando el tipo de dato `Variable`

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

Para una visualización agradable en pantalla se define su representación en la clase `Show`.

```
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'
```

Mostramos algunos ejemplos de definición de variables

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

Y definimos también variables empleando índices

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

De manera que su visualización será

```
ghci> x
x
ghci> y
y
ghci> a1
a1
ghci> a2
a2
```

Definición 2.2.1. 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 $\neg F$, $(F \wedge G)$, $(F \vee G)$, $(F \to G)$ y $(F \leftrightarrow G)$ son fórmulas.

Se define un tipo de dato para las fórmulas lógicas de primer orden.

```
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 define una visualización en la clase `Show`

```
instance Show Formula where
    show (Atomo r [])          = r
    show (Atomo r vs)          = r ++ 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)
```

Como ejemplo podemos representar las propiedades `reflexiva` y `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]))
```

Quedando su representación por pantalla

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

Definición 2.2.2. Una **estructura del lenguaje** L es un par $\mathcal{I} = (\mathcal{U}, I)$ tal que

1. $\mathcal{U}$ es un conjunto no vacío, denominado universo.
2. I es una función con dominio el conjunto de símbolos propios de L. L : `Símbolos` $\rightarrow$ `Símbolos` tal que
   - si c es una constante de L, entonces $I(c) \in \mathcal{U}$
   - si f es un símbolo de función n–aria de L, entonces $I(f) : \mathcal{U}^n \rightarrow \mathcal{U}$
   - si P es un símbolo de relación 0–aria de L, entonces $I(P) \in \{1, \}$
   - si R es un símbolo de relación n–aria de L, entonces $I(R) \subseteq \mathcal{U}^n$

Definimos el tipo de dato relativo al universo como una lista de elementos.

```
type Universo a = [a]
```

Definición 2.2.3. Una **asignación** es una función que hace corresponder a cada variable un elemento del universo.

Se define un tipo de dato para las asignaciones

```
type Asignacion a = Variable -> a
```

Necesitamos definir una asignación para los ejemplos. Tomamos una asignación constante muy sencilla.

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

2.3. Evaluación de fórmulas

En esta sección se pretende interpretar fórmulas. Una interpretación toma valores para las variables proposicionales, y se evalúan en una fórmula, determinando si la fórmula es verdadera o falsa, bajo esa interpretación.

Definición 2.3.1. Una **interpretación proposicional** es una aplicación $I : VP \to Bool$, donde VP representa el conjunto de las variables proposicionales.

A continuación, presentamos una tabla de valores de las distintas conectivas lógicas según las interpretaciones de `P` y `Q`. Falso lo representamos mediante el 0, y verdadero mediante el 1.

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

Se definirá mediante la función `valor`. Para ello se implementa $s(x|d)$, que es la aplicación de la asignación s pero con x interpretado como d, mediante la función `(sustituye s x d v)`. $s(x|d)$ viene dado por la fórmula

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

En Haskell se expresa mediante guardas

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

Esta función es auxiliar para la evaluación de fórmulas.

Un par de ejemplos de la función `(sustituye s x d v)` son

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

Definición 2.3.2. Una **interpretación de una estructura del lenguaje** es un par $(\mathcal{I}, A)$ formado por una estructura del lenguaje y una asignación A.

Definimos un tipo de dato para las interpretaciones de los símbolos de relación.

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

Definimos la función `(valor u i s form)` que calcula el valor de una fórmula en un universo `u`, con una interpretación `i`, respecto de la asignación `s`.

```
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

```
formula1 :: Formula
formula1 = ParaTodo x (Disyuncion [Atomo "P" [x],Atomo "Q" [x]])
```

```
ghci> formula1
∀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 formula1
False
ghci> valor (take 26 entidades) interpretacion2 asignacion formula1
True
```

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

2.4. Términos funcionales

En la sección anterior todos los términos han sido variables. Ahora consideraremos cualquier término.

Definición 2.4.1. Son **términos** en un lenguaje de primer orden:

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

Definimos un tipo de dato para los términos que serán la base para la definición de fórmulas en lógica de primer orden que no están compuestas sólo por variables.

```
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, definimos su representación.

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

Los términos funcionales son representados de la forma

```
ghci> Ter "f" [tx,ty]
f[x,y]
```

Caracterizamos las variables mediante la función `(esVariable x)`, que determina si un término es una variable

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

Por ejemplo,

```
ghci> esVariable tx
True
ghci> esVariable (Ter "f" [tx,ty])
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 cualquier término.

```
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 procedemos análogamente a la sección enterior, definiendo la representación de fórmulas por pantalla.

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

Algunos ejemplos de fórmulas son

```
formula2, formula3 :: Form
formula2 = PTodo x (PTodo y (Impl (Atom "R" [tx,ty])
                                  (Ex z (Conj [Atom "R" [tx,tz],
                                               Atom "R" [tz,ty]])))))
formula3 = Impl (Atom "R" [tx,ty])
                (Ex z (Conj [Atom "R" [tx,tz],Atom "R" [tz,ty]]))
```

Dichas funciones serán empleadas en futuros ejemplos. Su representación por pantalla queda:

```
ghci> formula2
∀x ∀y (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
ghci> formula3
(R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
```

Para la interpretación de los símbolos funcionales se define un nuevo tipo de dato

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

Para interpretar las fórmulas, se necesita primero una interpretación del valor en los términos.

Definición 2.4.2. Dada una estructura $\mathcal{I} = (U, I)$ de L y una asignación A en $\mathcal{I}$, se define la **función de evaluación de términos** $\mathcal{I}_A : Term(L) \to U$ por

$$\mathcal{I}_A(t) = \begin{cases} I(c), \text{ si } t \text{ es una constante } c \\ A(x), \text{ si } t \text{ es una variable } x \\ I(f)(\mathcal{I}_A(t_1), \ldots, \mathcal{I}_A(t_n)), \text{ si } t \text{ es } f(t_1, \ldots t_n) \end{cases}$$

Nota 2.4.1. $\mathcal{I}_A$ se lee "el valor de t en $\mathcal{I}$ respecto de A".

Definimos `(valorT i a t)` que es la función de evaluación de término

```
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)
```

Definimos el tipo de dato `Interpretación` como un par formado por las interpretaciones de los símbolos de relación y la de los símbolos funcionales.

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

Definición 2.4.3. Dada una estructura $\mathcal{I} = (U, I)$ de L y una asignación A sobre $\mathcal{I}$, se define la **función evaluación de fórmulas** $\mathcal{I}_A : Form(L) \to Bool$ por

- Si F es $t_1 = t_2$, $\mathcal{I}_A(F) = H_=(\mathcal{I}_A(t_1), \mathcal{I}_A(t_2))$
- Si F es $P(t_1, \ldots, t_n)$, $\mathcal{I}_A(F) = H_{I(P)}(\mathcal{I}_A(t_1), \ldots, \mathcal{I}_A(t_n))$
- Si F es $\neg G$, $\mathcal{I}_A(F) = H_\neg(\mathcal{I}_A(G))$
- Si F es $G * H$, $\mathcal{I}_A(F) = H_*(\mathcal{I}_A(G), \mathcal{I}_A(H))$
- Si F es $\forall x G$,

$$\mathcal{I}_A(F) = \begin{cases} 1, \text{ si para todo } u \in U \text{ se tiene } \mathcal{I}_{A[x/u]} = 1 \\ 0, \text{ en caso contario.} \end{cases}$$

- Si F es $\exists x G$,

$$\mathcal{I}_A(F) = \begin{cases} 1, \text{ si existe algún } u \in U \text{ tal que } \mathcal{I}_{A[x/u]} = 1 \\ 0, \text{ en caso contario.} \end{cases}$$

Definimos una función que determine el valor de una fórmula. Dicha función la denotamos por `(valorF u (iR,iF) a f)`, en la que `u` denota el universo, `iR` es la interpretación de los símbolos de relación, `iF` es la interpretación de los símbolos de función, `a` la asignación y `f` la 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]
```

Para construir un ejemplo tenemos que interpretar los elementos de una fórmula. Definimos las fórmulas 4 y 5, aunque emplearemos en el ejemplo sólo la `formula4`.

```
formula4, formula5 :: Form
formula4 = Ex x (Atom "R" [cero,tx])
formula5 = Impl (PTodo x (Atom "P" [tx])) (PTodo y (Atom "Q" [tx,ty]))
```

Sus representaciones quedan

```
ghci> formula4
∃x R[cero,x]
ghci> formula5
(∀x P[x]⟹∀y Q[x,y])
```

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 siguiente asignación

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

Quedando el ejemplo

```
ghci> valorF [0..] (interpretacionR1,interpretacionF1) asignacion1 formula4
True
```

Nota 2.4.2. 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.

Dada una fórmula F de L se tienen las siguientes definiciones:

Definición 2.4.4.

- Un **modelo** de una fórmula `F` es una interpretación para la que `F` es verdadera.
- Una fórmula F es **válida** si toda interpretación es modelo de la fórmula.
- Una fórmula F es **satisfacible** si existe alguna interpretación para la que sea verdadera.
- Una fórmula es **insatisfacible** si no tiene ningún modelo.

2.4.1. Generadores

Peniente de revisión.

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)
```

2.4.2. Otros conceptos de la lógica de primer orden

Las funciones `varEnTerm` y `varEnTerms` devuelven 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 2.4.3. La función `nub xs` elimina elementos repetidos en una lista `xs`. Se encuentra en el paquete `Data.List`.

Nota 2.4.4. 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 formula2  ==  [x,y,z]
varEnForm formula3  ==  [x,y,z]
varEnForm formula4  ==  [x]
```

Definición 2.4.5. 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)
```

Definición 2.4.6. Una variable x está **ligada** en una fórmula cuando tiene una aparición de la forma $\forall x$ o $\exists x$.

Se proponen varios ejemplos

```
variablesLibres formula2  ==  []
variablesLibres formula3  ==  [x,y]
variablesLibres formula4  ==  []
```

Definición 2.4.7. 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 formula2  ==  False
formulaAbierta formula3  ==  True
formulaAbierta formula4  ==  False
```

.

Capítulo 3

Deducción natural

En este capítulo se pretende implementar la deducción natural de la lógica de primer orden en Haskell. El contenido de este capítulo se encuentra en el módulo `DNH`.

```
module DNH where
import LPH
```

Capítulo 4

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`.

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

```

4.1. Sustitución

Definición 4.1.1. Una **sustitución** es una aplicación $S : \mathit{Variable} \to \mathit{Termino}$.

Nota 4.1.1. $[x_1/t_1, x_2/t_2, \ldots, x_n/t_n]$ representa la sustitución

$$S(x) = \begin{cases} t_i, \text{ si } x \text{ es } x_i \\ x, \text{ si } x \notin \{x_1, \ldots, x_n\} \end{cases}$$

En la lógica de primer orden, a la hora de emplear el método de tableros, es necesario sustituir las variables ligadas por términos. Por lo tanto, requerimos de la definición de un nuevo tipo de dato para las sustituciones.

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

Sería interesante comparar la representación de sustituciones mediante diccionarios con la librería Data.Map

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 [] = []
hacerApropiada (x:xs) | Var (fst x) /= snd x = x: hacerApropiada xs
                      | otherwise = hacerApropiada xs
```

Por ejemplo,

```
hacerApropiada [(x,tx)]         ==  []
hacerApropiada [(x,tx),(x,ty)]  ==  [(x,y)]
```

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

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

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

Por ejemplo,

```
dominio [(x,tx)]           ==  [x]
dominio [(x,tx),(x,ty)]    ==  [x]
recorrido [(x,tx)]         ==  [x]
recorrido [(x,tx),(x,ty)]  ==  [x,y]
```

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
```

Definición 4.1.2. $t[x_1/t_1,\ldots,x_n/t_n]$ es el término obtenido sustituyendo en t las apariciones de x_i por t_i.

Definición 4.1.3. La extensión de la sustitución a términos es la aplicación $S : Term(L) \to Term(L)$ definida por

$$S(t) = \begin{cases} c, & \text{si } t \text{ es una constante } c \\ S(x), & \text{si } t \text{ es una variable } x \\ f(S(t_1),\ldots,S(t_n)), & \text{si } T \text{ es } f(t_1,\ldots,t_n) \end{cases}$$

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
```

Por ejemplo,

```
susTerm  [(x,ty)] tx              ==  y
susTerms [(x,ty),(y,tx)] [tx,ty]  ==  [y,x]
```

Definición 4.1.4. $F[x_1/t_1,\ldots,x_n/t_n]$ es la fórmula obtenida sustituyendo en F las apariciones libres de x_i por t_i.

Definición 4.1.5. La extensión de S a fórmulas es la aplicación $S : Form(L) \to Form(L)$ definida por

$$S(F) = \begin{cases} P(S(t_1),\ldots,S(t_n)), & \text{si } F \text{ es la fórmula atómica } P(t_1,\ldots,t_n) \\ S(t_1) = S(t_2), & \text{si } F \text{ es la fórmula } t_1 = t_2 \\ \neg(S(G)), & \text{si } F \text{ es } \neg G \\ S(G) * S(H), & \text{si } F \text{ es } G * H \\ (Qx)(S_x(G)), & \text{si } F \text{ es } (Qx)G \end{cases}$$

donde S_x es la sustitución definida por

$$S_x(y) = \begin{cases} x, & \text{si } y \text{ es } x \\ S(y), & \text{si } y \text{ es distinta de } x \end{cases}$$

Definimos `(sustitucionForm s f)`, donde `s` representa la sustitución 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]
```

Por ejemplo,

```
ghci> formula3
(R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
ghci> sustitucionForm [(x,ty)] formula3
(R[y,y]⟹∃z (R[y,z]⋀R[z,y]))
```

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,

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

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

Y comprobando con QuickCheck que no lo es

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

Un contraejemplo más claro es

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

Nota 4.1.2. Las comprobaciones con QuickCheck emplean código del módulo `Generadores`.

Definición 4.1.6. Una sustitución se denomina **libre para una fórmula** cuando todas las apariciones de variables introducidas por la sustitución en esa fórmula resultan libres.

Un ejemplo de una sustitución que no es libre

```
ghci> Ex x (Atom "R" [tx,ty])
∃x R[x,y]
ghci> variablesLibres (Ex x (Atom "R" [tx,ty]))
[y]
ghci> sustitucionForm [(y,tx)] (Ex x (Atom "R" [tx,ty]))
∃x R[x,x]
ghci> variablesLibres (sustitucionForm [(y,tx)] (Ex x (Atom "R" [tx,ty])))
[]
```

Un ejemplo de una sustitución libre

```
ghci> formula5
(∀x P[x]⟹∀y Q[x,y])
ghci> variablesLibres formula5
[x]
ghci> sustitucionForm [(x,tz)] formula5
(∀x P[x]⟹∀y Q[z,y])
ghci> variablesLibres (sustitucionForm [(x,tz)] formula5)
[z]
```

¿Una sustitución libre se puede caracterizar por la longitud de la lista de variables libres antes y despues de la sustitución?

4.2. Unificación

Definición 4.2.1. 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,

```
unificadoresListas [tx] [ty]  ==  [[(x,y)]]
unificadoresListas [tx] [tx]  ==  [[]]
```

4.3. Skolem

Definición 4.3.1. Una fórmula está en **forma normal conjuntiva** si es una conjunción de disyunciones de literales.

$$(p_1 \vee \cdots \vee p_n) \wedge \cdots \wedge (q_1 \vee \cdots \vee q_m)$$

Definición 4.3.2. Una fórmula está en **forma normal disyuntiva** si es una disyunción de conjunciones de literales.

$$(p_1 \wedge \cdots \wedge p_n) \vee \cdots \vee (q_1 \wedge \cdots \wedge q_m)$$

4.3.1. Forma rectificada

Definición 4.3.3. Una fórmula F está en forma **rectificada** si ninguna variable aparece libre y ligada y cada cuantificador se refiere a una variable diferente.

4.3.2. Forma normal prenexa

Definición 4.3.4. Una fórmula F está en forma **normal prenexa** si es de la forma $Q_1x_1 \ldots Q_nx_nG$ donde $Q_i \in \{\forall, \exists\}$ y G no tiene cuantificadores.

4.3.3. Forma normal prenexa conjuntiva

Definición 4.3.5. Una fórmula F está en **forma normal prenexa conjuntiva** si está en forma normal prenexa con G en forma normal conjuntiva.

4.3.4. Forma de Skolem

Definición 4.3.6. La fórmula `F` está en **forma de Skolem** si es de la forma $\forall x_1 \ldots \forall x_nG$, donde $n \geq 0$ y G no tiene cuantificadores.

Para transformar 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 4.3.7. Las fórmulas `F` y `G` son **equivalentes** si para toda interpretación valen lo mismo.

Definición 4.3.8. Las fórmulas `F` y `G` son **equisatisfacibles** si se cumple que ambas son satisfacibles o ninguna lo es.

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 anteriormente como ejemplo:

```
ghci> formula2
∀x ∀y (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
ghci> elimImpEquiv formula2
∀x ∀y (¬R[x,y]⋁∃z (R[x,z]⋀R[z,y]))
ghci> formula3
(R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
ghci> elimImpEquiv formula3
(¬R[x,y]⋁∃z (R[x,z]⋀R[z,y]))
ghci> formula4
∃x R[cero,x]
ghci> elimImpEquiv formula4
∃x R[cero,x]
```

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

Se define la función `(skol k vs)` que convierte una lista de variables a un término de Skolem. Al calcular la forma de skolem de una fórmula, las variables cuantificadas son sustituidas por lo que denotamos **término de skolem** para obtener una fórmula libre. Los términos de skolem estan compuestos por las siglas "sk" y un entero que lo identifique.

Nota 4.3.1. El término de skolem está expresado con la misma estructura que los términos funcionales.

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

Por ejemplo,

```
skol 1 [x]  ==  sk1[x]
```

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 identificador de la forma de Skolem.

Definición 4.3.9. La **Polaridad** cuantifica las apariciones de las variables cuantificadas de la siguiente forma:

- Una cantidad de apariciones impar de x en la subfórmula F de $\exists xF$ indica que x tiene una polaridad negativa en la fórmula.
- Una cantidad de apariciones par de x en la subfórmula F de $\forall xF$ indica que x tiene una polaridad positiva en la fórmula.

```
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 skolemización 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> skolem formula2
∀x ∀y (¬R[x,y]⋁(R[x,sk0[x,y]]⋀R[sk0[x,y],y]))
ghci> skolem formula3
(¬R[x,y]⋁(R[x,sk0]⋀R[sk0,y]))
ghci> skolem formula4
R[cero,sk0]
ghci> skolem formula5
(¬P[sk0]⋁∀y Q[x,y])
```

4.4. Tableros semánticos

Definición 4.4.1. Un conjunto de fórmulas es **consistente** si tiene algún modelo. En caso contrario, se denomina **inconsistente**.

Distinguir el caso de fórmulas con variables libres.

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.

Las definimos en Haskell

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

Nota 4.4.1. En las tablas, cada elemento de una columna es equivalente a su análogo en la otra columna.

Precisar la equivalencia con las componentes.

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 consistente, así como la búsqueda de modelos.

Definición 4.4.2. Un **literal** es un átomo o la negación de un átomo.

Lo definimos en haskell

```
atomo, negAtomo, literal :: Form -> Bool

atomo (Atom _ _)          = True
atomo _                   = False

negAtomo (Neg (Atom _ _)) = True
negAtomo  _               = False

literal f = atomo f || negAtomo f
```

El método de tableros de un conjunto de fórmulas S sigue el siguiente algoritmo:

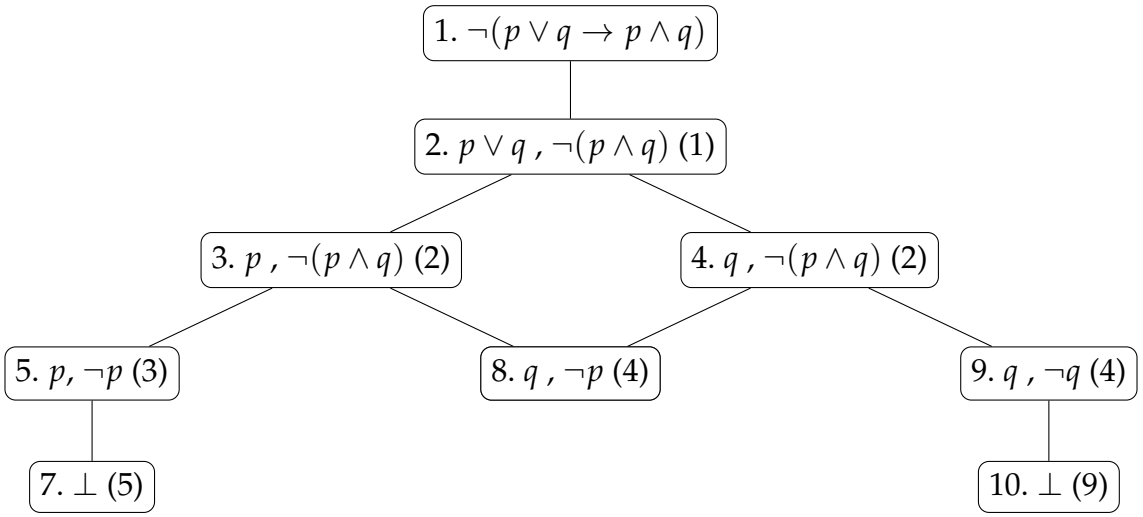
- El árbol cuyo único nodo tiene como etiqueta S es un tablero de S.
- Sea $\mathcal{T}$ un tablero de S y S_1 la etiqueta de una hoja de $\mathcal{T}$.
  1. Si S_1 contiene una fórmula y su negación, entonces el árbol obtenido añadiendo como hijo de S_1 el nodo etiquetado con $\{\bot\}$ es un tablero de S.
  2. Si S_1 contiene una doble negación $\neg\neg F$, entonces el árbol obtenido añadiendo como hijo de S_1 el nodo etiquetado con $(S_1 \setminus \{\neg\neg F\}) \cup \{F\}$ es un tablero de S.
  3. Si S_1 contiene una fórmula alfa F de componentes F_1 y F_2, entonces el árbol obtenido añadiendo como hijo de S_1 el nodo etiquetado con $(S_1 \setminus \{F\}) \cup \{F_1, F_2\}$ es un tablero de S.
  4. Si S_1 contiene una fórmula beta de F de componentes F_1 y F_2, entonces el árbol obtenido añadiendo como hijos de S_1 los nodos etiquetados con $(S_1 \setminus \{F\}) \cup \{F_1\}$ y $(S_1 \setminus \{F\}) \cup \{F_2\}$ es un tablero de S.

Definición 4.4.3. Se dice que una hoja es **cerrada** si contiene una fórmula y su negación. Se representa $\bot$

Definición 4.4.4. Se dice que una hoja es **abierta** si es un conjunto de literales y no contiene un literal y su negación.

Definición 4.4.5. Un **tablero completo** es un tablero tal que todas sus hojas son abiertas o cerradas.

Ejemplo de tablero completo



Se solapan las ramas del arbol

La fórmula del tablero se representa en Haskell Se definen los átomos.

```
p = Atom "p" []
q = Atom "q" []
r = Atom "r" []
```

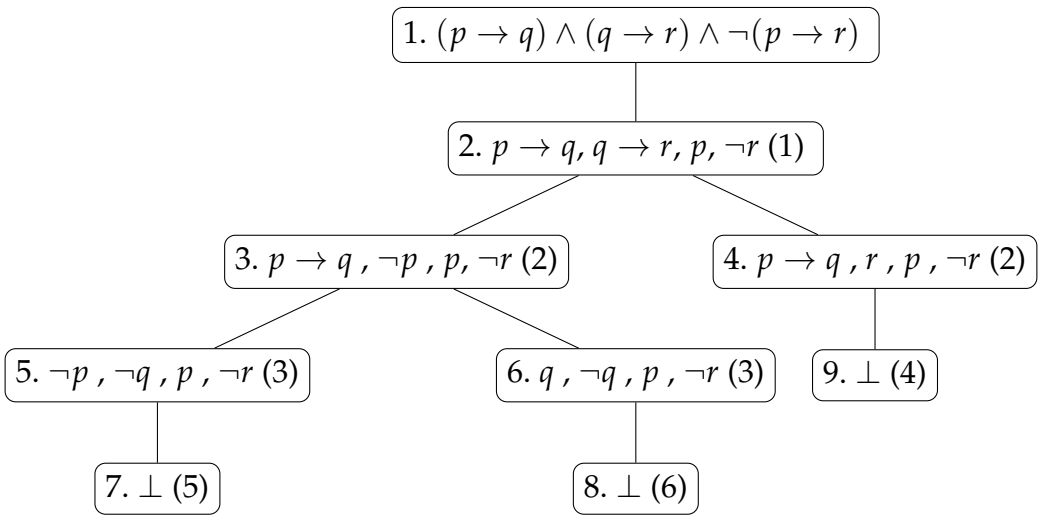
Para que la fórmula quede

```
tab1 = Neg (Impl (Disy [p,q]) (Conj [p,q]))
```

```
ghci> tab1
¬((p⋁q)⟹(p⋀q))
```

Definición 4.4.6. Un tablero es **cerrado** si todas sus hojas son cerradas.

Un ejemplo de tablero cerrado es



La fórmula del tablero se representa en Haskell

```
tab2 = Conj [Impl p q, Impl q r, Neg (Impl p r)]
```

```
ghci> tab2
((p⟹q)⋀((q⟹r)⋀¬(p⟹r)))
```

Teorema 4.4.1. *Si una fórmula F es consistente, 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`. CompruebaTabta 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.

El método de tableros se suele representar en forma de árbol, por ello definiremos el tipo de dato `Nodo`.

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

Donde la primera lista de términos representa los literales positivos, la segunda lista de términos negativos, y la lista de fórmulas son aquellas ligadas a los términos de las listas anteriores.

Definimos 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]
```

Por ejemplo,

```
ghci> componentes (skolem (tab1))
[¬¬(p⋀[q]),¬(p⋁[q])]
ghci> componentes (skolem (tab2))
[(¬p⋁[q]),(¬q⋁[r]),¬(¬p⋁[r])]
```

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 = descomp [] where
  descomp xs f | gamma f   = descomp (xs ++ [x]) g
               | otherwise = (xs,f)
    where x   = varLigada f
          [g] = componentes f
```

Por ejemplo,

```
ghci> formula2
∀x ∀y (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
ghci> descomponer formula2
([x,y],(R[x,y]⟹∃z (R[x,z]⋀[R[z,y]])))
ghci> formula3
(R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
ghci> descomponer formula3
([],(R[x,y]⟹∃z (R[x,z]⋀[R[z,y]])))
ghci> formula4
∃x R[cero,x]
ghci> descomponer formula4
([],∃x R[cero,x])
```

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

```
ramificacion :: Nodo -> Tablero
ramificacion (Nd i pos neg []) = [Nd i pos neg []]
ramificacion (Nd i pos neg (f:fs))
  | atomo f    = if literalATer f `elem` neg
                 then []
                 else [Nd i ( literalATer f : pos) neg fs]
  | negAtomo f = if literalATer f `elem` 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 literalATer f `elem` neg
                 then (k,[])
                 else (k,[Nd i (literalATer f : pos) neg fs])
```

```
  | negAtomo f = if literalATer f `elem` 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 4.4.7. 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
  | k == n             = expandeTablero n (nuevoNodo ++ nodos)
  | otherwise          = 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)
  | k == n =
      trace (show nodo ++ "\n\n") (expandeTableroG k (nuevoNodo ++ nodos))
```

```
  | otherwise =
      trace (show nodo ++"\n\n") (expandeTableroG (n-1) (nodos ++ nuevoNodo))
  where (k, nuevoNodo) = ramificacionP n nodo
```

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

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

Definimos las funciones auxiliares `(sustNodo nd)` y `(sustTab tb)` que aplican sustituciones a nodos y tableros.

```
sustNodo :: Sust -> Nodo -> Nodo
sustNodo b (Nd i pos neg f) =
  Nd i (susTerms b pos) (susTerms b neg) (sustitucionForms b f)

susTab :: Sust -> Tablero -> Tablero
susTab = map . sustNodo
```

Se define `esCerrado` para determinar si un tablero es cerrado.

```
esCerrado :: Tablero -> [Sust]
esCerrado []     = [identidad]
esCerrado [nodo] = esNodoCerrado nodo
esCerrado (nodo:nodos) =
  concat [esCerrado (susTab s nodos) | s <- esNodoCerrado nodo ]
```

Dada una fórmula es necesario crear un tablero inicial para posteriormente desarrollarlo. Lo hacemos mediante la función `(tableroInicial f)`.

```
tableroInicial :: Form -> Tablero
tableroInicial f = [Nd [] [] [] [f]]
```

Por ejemplo,

```
ghci> tab1
¬((p⋁q)⟹(p⋀q))
ghci> tableroInicial tab1
[Nd [] [] [] [¬((p⋁[q])⟹(p⋀[q]))]]
```

La función `(refuta k f)` intenta refutar la fórmula `f` con un tablero de profundidad `k`.

```
refuta :: Int -> Form -> Bool
refuta k f = esCerrado tab /= []
  where tab = expandeTablero k (tableroInicial f)
```

Nota 4.4.2. Se puede emplear tambien `expandeTableroG`, por ello se deja comentado para su posible uso.

Definición 4.4.8. Una fórmula F es un **teorema** mediante tableros semánticos si tiene una prueba mediante tableros; es decir, si $\neg F$ tiene un tablero completo cerrado

Finalmente, podemos determinar si una fórmula es un teorema y si es satisfacible mediante las funciones `(esTeorema n f)` y `(satisfacible n f)`.

```
esTeorema, satisfacible :: Int -> Form -> Bool
esTeorema n = refuta n . skolem . Neg
satisfacible n = not . refuta n . skolem
```

Por ejemplo tomando `tautologia1` y la ya usada anteriormente `formula2`

```
tautologia1 :: Form
tautologia1 = Disy [Atom "P" [tx], Neg (Atom "P" [tx])]
```

se tiene

```
ghci> tautologia1
(P[x]⋁¬P[x])
ghci> esTeorema 1 tautologia1
True
ghci> formula2
∀x ∀y (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
ghci> esTeorema 20 formula2
False
ghci> tab1
¬((p⋁q)⟹(p⋀q))
ghci> esTeorema 20 tab1
False
ghci> satisfacible 1 tab1
True
ghci> tab2
((p⟹q)⋀((q⟹r)⋀¬(p⟹r)))
ghci> esTeorema 20 tab2
False
ghci> satisfacible 20 tab2
False
```

Teorema 4.4.2. *El cálculo de tableros semánticos es adecuado y completo.*

Definición 4.4.9. Una fórmula F es **deducible** a partir del conjunto de fórmulas S si exite un tablero completo cerrado de la conjunción de S y $\neg F$. Se representa por $S \vdash_{Tab} F$.

Explicar más el método de tableros con polaridad.

Comparar la implementación con la de Ben Ari que se encuentra en https://github.com/motib/mlcs/blob/master/fol/tabl.pro

Capítulo 5

Modelos de Herbrand

En este capítulo se pretende formalizar los modelos de Herbrand. Herbrand propuso un método constructivo para generar interpretaciones de fórmulas o conjuntos de fórmulas.

El contenido de este capítulo se encuentra en el módulo `Herbrand`.

```
module Herbrand where
import Data.List
import PFH
import LPH
import PTLP
```

Definición 5.0.1. Una **fórmula básica** es una fórmula sin variables ni cuantificadores.

5.1. Universo de Herbrand

Definición 5.1.1. El **universo de Herbrand** de L es el conjunto de términos básicos de F. Se reprenta por $\mathcal{UH}(L)$.

Proposición 5.1.1. $\mathcal{UH}(L) = \bigcup_{i\geq 0} H_i(L)$, donde $H_i(L)$ es el nivel i del $\mathcal{UH}(L)$ definido por

$$H_0(L) = \begin{cases} C, & \text{si } C \neq \emptyset \\ a, & \text{en caso contrario (a nueva constante)} \end{cases}$$

$$H_{i+1}(L) = H_i(L) \cup \{f(t_1, \ldots, t_n) : f \in \mathcal{F}_n \text{ y } t_i \in H_i(L)\}$$

A continuación caracterizamos las constantes. Definimos la función `(esConstante c)`.

```
esConstante :: Termino -> Bool
esConstante (Ter _ []) = True
esConstante _          = False
```

Un par de ejemplos

```
esConstante a   ==  True
esConstante tx  ==  False
```

Definimos la función `(constDeTerm t)` que determine las de un término `t`.

```
constDeTerm :: Termino -> [Termino]
constDeTerm (Var _) = []
constDeTerm c@(Ter _ []) = [c]
constDeTerm (Ter str (t:ts)) | esConstante t = t: aux (Ter str ts)
                             | otherwise =
                                 constDeTerm t ++ aux (Ter str ts)
                            where
                              aux (Ter _ []) = []
                              aux t = constDeTerm t
```

Nota 5.1.1. La función `aux` evita que en la recursión consideremos un término funcional como constante al quedarse vacía la lista de elementos a los que se aplica.

Por ejemplo

```
ghci> Ter "f" [Ter "a" [] , Ter "b" [] , Ter "g" [tx, Ter "c" []]]
f[a,b,g[x,c]]
ghci> constDeTerm (Ter "f" [Ter "a" [] , Ter "b" [] , Ter "g" [tx, Ter "c" []]])
[a,b,c]
```

Definimos `(constForm f)` para determinar las constantes de `f`.

```
constForm :: Form -> [Termino]
constForm  (Atom _ ts)   =  nub (concat [constDeTerm t | t <- ts])
constForm  (Neg f)       = constForm f
constForm  (Impl f1 f2)  = constForm f1 `union` constForm f2
constForm  (Equiv f1 f2) = constForm f1 `union` constForm f2
constForm  (Conj fs)     = nub (concatMap constForm fs)
constForm  (Disy fs)     = nub (concatMap constForm fs)
constForm  (PTodo x f)   = constForm  f
constForm  (Ex x f)      = constForm  f
```

Si como función auxiliar de `constForm` empleamos `constDeTerm` obtendremos las constantes de la fórmula. Por ejemplo

```
ghci> Atom "P" [a,tx]
P[a,x]
ghci> constForm (Atom "P" [a,tx])
[a]
ghci> Conj [Atom "P" [a, Ter "f" [tx,b]], Atom "R" [Ter "g" [tx,ty],c]]
```

```
(P[a,f[x,b]]/\R[g[x,y],c])
ghci> constForm  (Conj [Atom "P" [a, Ter "f" [tx,b]],
                  Atom "R" [Ter "g" [tx,ty],c]])
[a,b,c]
```

Definimos `(esFuncion f)` y `(funForm f)` para obtener todos los símbolos funcionales que aparezcan en la fórmula `f`.

```
esFuncion :: Termino -> Bool
esFuncion (Ter _ xs) | not (null xs) = True
                     | otherwise = False
esFuncion _ = False
```

```
funForm :: Form -> [Termino]
funForm (Atom _ ts)   = nub [ t | t <- ts, esFuncion t]
funForm (Neg f)       = funForm f
funForm (Impl f1 f2)  = funForm f1 `union` funForm f2
funForm (Equiv f1 f2) = funForm f1 `union` funForm f2
funForm (Conj fs)     = nub (concatMap funForm fs)
funForm (Disy fs)     = nub (concatMap funForm fs)
funForm (PTodo x f)   = funForm f
funForm (Ex x f)      = funForm f
```

Por ejemplo

```
ghci> funForm (Conj [Atom "P" [a, Ter "f" [tx,b]], Atom "R" [Ter "g" [tx,ty],c]])
[f[x,b],g[x,y]]
```

Definición 5.1.2. La **aridad** de un operador $f(x_1, \ldots, x_n)$ es el número número de argumentos a los que se aplica.

Definimos `(aridadF t)` para calcular la aridad del término `t`.

```
aridadF :: Termino -> Int
aridadF (Ter _ ts) = length ts
```

También necesitamos definir dos funciones auxiliares que apliquen los símbolos funcionales a las constantes del universo de Herbrand. Las funciones son `(aplicaFunAConst f c)` que aplica el símbolo funcional `f` a la constante `c` y `(aplicaFun fs cs)` que es una generalización a listas de la anterior.

```
aplicaFunAConst (Ter s _) = Ter s

aplicaFun [] cs = []
aplicaFun (f:fs) cs =
    map (aplicaFunAConst f) (subconjuntosTam (aridadF f) cs)
                            ++ aplicaFun fs cs
```

Así podemos ya obtener el universo de Herbrand de una fórmula `f` definiendo `(univHerbrand n f)`

```
univHerbrand :: (Eq a, Num a) => a -> Form -> Universo Termino
univHerbrand 0 f = if  constForm  f /= [] then constForm f
                   else [a]
univHerbrand 1 f =
    nub (univHerbrand 0 f ++ aplicaFun (funForm f) (univHerbrand 0 f))
univHerbrand n f =
    nub (univHerbrand (n-1) f ++ aplicaFun (funForm f)
        (univHerbrand (n-1) f))
```

Por ejemplo

```
u = Variable "u" []
tu = Var u
```

```
ghci> formula2
∀x ∀y (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
ghci> univHerbrand 0 formula2
[a]
ghci>
ghci> Conj [Disy [Atom "P" [a],Atom "P" [b]],
            Disy [Neg (Atom "P" [b]),Atom "P" [c]],
            Impl (Atom "P" [a]) (Atom "P" [c])]
((P[a]⋁P[b])⋀((¬P[b]⋁P[c])⋀(P[a]⟹P[c])))
ghci> univHerbrand 0 (Conj [Disy [Atom "P" [a],Atom "P" [b]],
                            Disy [Neg (Atom "P" [b]),Atom "P" [c]],
                            Impl (Atom "P" [a]) (Atom "P" [c])])
[a,b,c]

ghci> Conj [PTodo x (PTodo y (Impl (Atom "Q" [b,tx])
                   (Disy [Atom "P" [a],Atom "R" [ty]]))),
                    Impl (Atom "P" [b]) (Neg (Ex z (Ex u (Atom "Q" [tz,tu])))))]
(∀x ∀y (Q[b,x]⟹(P[a]⋁R[y]))⋀(P[b]⟹¬∃z ∃u Q[z,u]))
ghci> univHerbrand 0 (Conj [PTodo x (PTodo y (Impl (Atom "Q" [b,tx])
                    (Disy [Atom "P" [a],Atom "R" [ty]]))),
                     Impl (Atom "P" [b]) (Neg (Ex z (Ex u (Atom "Q" [tz,tu])))))])
[b,a]
```

Proposición 5.1.2. $\mathcal{UH}$ es finito si y sólo si no tiene símbolos de función.

Definimos fórmulas con términos funcionales para el ejemplo

```
formula6, formula7 :: Form
formula6 = PTodo x (Atom "P" [Ter "f" [tx]])
formula7 = PTodo x (Atom "P" [Ter "f" [a,b]])
```

quedando como ejemplo

```
ghci> formula6
∀x P[f[x]]
ghci> univHerbrand 5 formula6
[a,f[a],f[f[a]],f[f[f[a]]],f[f[f[f[a]]]],f[f[f[f[f[a]]]]]]
ghci> univHerbrand 0 formula7
[a,b]
ghci> univHerbrand 1 formula7
[a,b,f[a,b],f[b,a]]
ghci> univHerbrand 2 formula7
[a,b,f[a,b],f[b,a],f[f[a,b],f[b,a]],f[f[b,a],f[a,b]],f[b,f[b,a]],
f[f[b,a],b],f[b,f[a,b]],f[f[a,b],b],f[a,f[b,a]],f[f[b,a],a],f[a,f[a,b]],
f[f[a,b],a]]
```

Hay que tener en cuenta que se dispara la cantidad de elementos del universo de Herbrand ante términos funcionales con aridad grande.

```
length (univHerbrand 0 formula7)  ==  2
length (univHerbrand 1 formula7)  ==  4
length (univHerbrand 2 formula7)  ==  14
length (univHerbrand 3 formula7)  ==  184
```

5.2. Base de Herbrand

Definición 5.2.1. La **base de Herbrand** $\mathcal{BH}(L)$ de un lenguaje L es el conjunto de átomos básicos de L.

Con el objetivo de definir una función que obtenga la base de Herbrand; necesitamos poder calcular el conjunto de símbolos de predicado de una fórmula.

Definimos `(aridadP f)` que determina la aridad del predicado de la fórmula atómica `f`.

```
aridadP :: Form -> Int
aridadP (Atom _ xs) = length xs
```

Definimos `(esPredicado f)` que determina si `f` es un predicado.

```
esPredicado :: Form -> Bool
esPredicado (Atom _ []) = False
esPredicado (Atom _ _)  = True
esPredicado _           = False
```

Calculamos el conjunto de los predicados de una fórmula `f` definiendo la función `(predicadosForm f)`.

```
predicadosForm :: Form -> [Form]
predicadosForm p@(Atom _ _)    = [p | esPredicado p]
predicadosForm (Neg f)         = predicadosForm f
predicadosForm (Impl f1 f2)    =
  predicadosForm f1 `union` predicadosForm f2
predicadosForm (Equiv f1 f2)   =
  predicadosForm f1 `union` predicadosForm f2
predicadosForm (Conj fs)       = nub (concatMap predicadosForm fs)
predicadosForm (Disy fs)       = nub (concatMap predicadosForm fs)
predicadosForm (PTodo x f)     = predicadosForm f
predicadosForm (Ex x f)        = predicadosForm f
```

Esta función repite el mismo predicado si tiene distintos argumentos, como por ejemplo

```
ghci> formula2
∀x ∀y (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
ghci> predicadosForm formula2
[R[x,y],R[x,z],R[z,y]]
```

Por lo tanto, definimos `(predForm f)` que obtiene los símbolos de predicado sin repeticiones.

```
predForm :: Form -> [Form]
predForm = noRep . predicadosForm
    where
      noRep [] = []
      noRep (Atom st t : ps) =
          if null [Atom str ts  | (Atom str ts) <- ps, str== st]
          then Atom st t : noRep ps else noRep ps
```

Así obtenemos

```
ghci> predForm formula2
[R[z,y]]
```

Podemos tambien obtener una lista de los símbolos de predicado definiendo `(simbolosPred f)`

```
simbolosPred :: Form -> [Nombre]
simbolosPred f = [str | (Atom str _) <- ps]
    where ps = predForm f
```

Definir directamente simbolosPred sin usar predForm y eliminar predForm y predicadosForm. Respuesta a comentario: La razón de no haber definido simbolosPred sin predForm y predicadosForm es por la necesidad de mantener la aridad del predicado para la definición de base de Herbrand.

Finalmente, necesitamos aplicar los símbolos de predicado al universo de Herbrand correspondiente.

Definimos las funciones `(aplicaPred p ts)` y su generalización `(apPred ps ts)` para aplicar los simbolos de predicado.

```
aplicaPred :: Form -> [Termino] -> Form
aplicaPred (Atom str _) = Atom str

apPred :: [Form] -> [Termino] -> [Form]
apPred [] ts = []
apPred (p:ps) ts = map (aplicaPred p) (subconjuntosTam (aridadP p) ts)
                   ++ apPred ps ts
```

Algunos ejemplos son

```
ghci> aplicaPred (Atom "P" [tx]) [ty]
P[y]
ghci> aplicaPred (Atom "R" [tx,ty]) [tz,ty]
R[z,y]
ghci> apPred [Atom "P" [tx]] [tx,ty,tz]
[P[z],P[y],P[x]]
ghci> apPred [Atom "P" [tx], Atom "R" [tx,ty]] [tx,ty,tz]
[P[z],P[y],P[x],R[y,z],R[z,y],R[x,z],R[z,x],R[x,y],R[y,x]]
```

Definimos la función `(baseHerbrand n f)`

```
baseHerbrand :: (Eq a, Num a) => a -> Form -> [Form]
baseHerbrand n f = apPred (predForm f) (univHerbrand n f)
```

Algunos ejemplos

Aplicar baseHerbrand a los ejemplos de LMF. Faltan aquellos elementos que repiten constante. PE: Q(a,a)

Podemos hacer un análisis de la fórmula 6, calculando sus constantes, símbolos funcionales y símbolos de predicado. Así como el universo de Herbrand y la base de Herbrand.

Corregir el análisis de la base de Herbrand.

Teorema 5.2.1. *$\mathcal{BH}(L)$ es finita si y sólo si L no tiene símbolos de función.*

5.3. Interpretaciones de Herbrand

Definición 5.3.1. Una **interpretación de Herbrand** de una fórmula F es una interpretación $\mathcal{I} = (\mathcal{U}, I)$ tal que

- $\mathcal{U}$ es el universo de Herbrand de F.
- $I(c) = c$, para constante c de F.
- $I(f) = f$, para cada símbolo funcional de F.

5.4. Modelos de Herbrand

Definición 5.4.1. Un **modelo de Herbrand** de una fórmula F es una interpretación de Herbrand de F que es modelo de F.

Un **modelo de Herbrand** de un conjunto de fórmulas S es una interpretación de Herbrand de S que es modelo de S.

Falta la definición de interpretación de Herbrand de un conjunto de fórmulas.

Nota 5.4.1. Los conjuntos de fórmulas pueden ser representados mediante **cláusulas** que no son más que fórmulas entre comas, donde las comas representan la conjunción. Posteriormente definiremos las cláusulas pero por ahora representamos conjuntos como conjunción de funciones.

Aclarar la nota sobre cláusulas.

Definimos `(valHerbrand f)` para determinar si existe algún subconjunto de la base de Herbrand que sea modelo de la fórmula `f`.

Para la recursión necesitamos la fórmula `(valorHerbrand f f)` pues hace recursión en una de las `f` para luego calcular la base de Herbrand de la otra.

Aclarar el significado de valorHerbrand y añadir ejemplos.

```
valorHerbrand :: Form -> Form -> Int -> Bool
valorHerbrand p@(Atom str ts) f n =
  p `elem` baseHerbrand n f
valorHerbrand (Neg g) f n =
  not (valorHerbrand g f n)
valorHerbrand (Impl f1 f2) f n =
  valorHerbrand f1 f n <= valorHerbrand f2 f n
valorHerbrand (Equiv f1 f2) f n =
  valorHerbrand f1 f n == valorHerbrand f2 f n
valorHerbrand (Conj fs) f n =
  all (==True) [valorHerbrand g f n | g <- fs]
valorHerbrand (Disy fs) f n =
```

```
  True `elem` [valorHerbrand g f n | g <- fs]
valorHerbrand (PTodo v g) f n =
  valorHerbrand g f n
valorHerbrand (Ex v g)  f n =
  valorHerbrand g f n
```

Nota 5.4.2. Se puede cambiar la `n` de la base de Herbrand a la que se calcula la existencia de modelo. Eso es interesante para fórmulas con símbolos funcionales.

```
valHerbrand :: Form -> Int -> Bool
valHerbrand g = valorHerbrand f f
  where f = skolem g
```

Añadimos un par de fórmulas para los ejemplos

```
formula8 = Impl (Atom "P" [tx]) (Atom "Q" [tx])
formula9 = Conj [Atom "P" [tx], Neg (Atom "P" [tx])]
```

```
ghci> valHerbrand formula8 0
True
ghci> valHerbrand formula9 0
False
ghci> valHerbrand formula6 0
False
ghci> valHerbrand formula6 1
True
ghci> valHerbrand formula2 0
False
ghci> valHerbrand formula2 1
True
```

La fórmula 9 es una contradicción, es decir, no tiene ningún modelo. Podemos comprobarlo mediante tableros

```
ghci> satisfacible 1 formula9
False
```

Capítulo 6

Resolución en lógica de primer orden

En este capítulo se introducirá la resolución en la lógica de primer orden y se implementará en Haskell. El contenido de este capítulo se encuentra en el módulo `RES`

```
module RES where
import LHS
```

Parte II

Sistemas utilizados

Capítulo 7

Apéndice: GitHub

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

7.1. 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.

7.2. 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`

7.3. 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>"
```

7.4. 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.

7.5. 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

Lista de tareas pendientes