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 7 de junio de 2017)

Índice general

Introducción

La introducción tiene que ser más amplia de forma que casi se corresponda con la presentación del TFG. Puedes ver la de Dani y la de María.

Para Haskell se recomienda [7] y [10].

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 implementación en Haskell de la teoría impartida en la asignatura Lógica matemática y fundamentos [1] ([2]). Para ello, se lleva a cabo la adaptación de los programas del libro de J. van Eijck Computational semantics and type theory [2] ([13]) y su correspondiente teoría.
- En la segunda parte se comentan aquellos sistemas empleados en la elaboración del trabajo.

La lógica de primer orden o lógica de predicados nace como una extensión de la lógica proposicional ante algunas carencias que ésta presenta. La lógica proposicional tiene como objetivo modelizar el razonamiento y nos aporta una manera de formalizar las demostraciones. El problema surge cuando aparecen propiedades universales o nos cuestionamos la existencia de elementos con una cierta propiedad, es decir, aparecen razonamientos del tipo

> Todos los sevillanos van a la feria.
> Yo soy sevillano.
> Por lo tanto, voy a la feria

A pesar de ser yo una clara prueba de que la proposición no es cierta, sí que se infieren las carencias de la lógica proposicional.

Una vez que vemos la necesidad de la existencia de la lógica de primer orden, nos damos cuenta de que el progreso nos lleva a la mejora y automatización del razonamiento, y eso desemboca en el empleo de lenguajes informáticos para pasar del

[1] https://www.cs.us.es/~jalonso/cursos/lmf-15

[2] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.467.1441&rep=rep1&type=pdf

"hombre" a la "máquina". Con este fin, se ha elegido como lenguaje para la implementación Haskell, un lenguaje de programación funcional, y nos vemos motivados a esta elección debido a su cercanía a la representación matemática en su "enfoque funcional".

Como se ha comentado antes, se implementarán aquellas cuestiones impartidas en la asignatura de Lógica matemática y fundamentos. Cuestiones como la semántica de la lógica proposicional, resolución, prueba de teoremas, modelos de Herbrand, ... Todas estas implementaciones desembocan en una pregunta mayor, ¿es la lógica y la programación lo mismo? O más allá, ¿existe una equivalencia entre la programación y las matemáticas? Estas preguntas se ven motivadas ante la posibilidad de implementar distintos ámbitos de las matemáticas, el preguntarnos si cada rama de las matemáticas tiene su clon en formato código, desembocando en el último capítulo de la primera parte, la correspondencia de Curry-Howard, que nos da una satisfactoria relación entre ambos campos.

Finalmente, la segunda parte del trabajo consta de una guía sobre GitHub, para el trabajo común entre varias personas a través de esta plataforma y el uso de `git` en emacs.

Indicar que se ha escrito en Haskell literario (mezclando código Haskell y LaTeX) y se ha desarrollado como un proyecto en GitHub (poniendo un enlace).

Comentar el uso de la librería `doctest` para la comprobación de las definiciones y ejemplos.

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]).

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

```
module PFH where
import Data.List
```

1.1. Introducción a Haskell

En esta sección se introducirán funciones básicas para la programación en Haskell. Como método didáctico, se empleará la definición de funciones ejemplos, así como la redefinición de funciones que Haskell ya tiene predefinidas, con el objetivo de que el lector aprenda "*a montar en bici, montando*".

A continuación se muestra la definición `(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 enteros y booleanos, respectivamente. Pero también se puede trabajar con otros tipos de datos como son las cadenas de caracteres que son tipo `[Char]` o `String`.Definimos la función `(contieneLaLetra xs l)` que 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 parecida 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, de una mayor dificultad,es la construcción de variaciones con repeticiones de una lista. Se define `(variacionesR n xs)`

```
variacionesR :: Int -> [a] -> [[a]]
variacionesR _ [] = [[]]
variacionesR 0 _  = [[]]
variacionesR k xs =
    [z:ys | z <- xs, ys <- variacionesR (k-1) xs]
```

Nota 1.1.2. La función `variacionesR` será util en capítulos posteriores.

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 los pares (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 1.1.3. 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 1.1.4. $\otimes$ 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 ([11])

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 se emplean en la comprobación de propiedades con QuickCheck.

```
module Generadores where
import PFH
import Test.QuickCheck
import Control.Monad
```

Teoría básica de generadores

Cuando se pretende realizar pruebas aleatorias mediante QuickCheck, es necesario generar casos aleatorios. Dos buenas fuentes de información sobre generadores son [14] y [9]. Dado un tipo de dato, se puede hacer miembro de la llamada clase `Arbitrary`.

```
class Arbitrary a where
  arbitrary :: Gen a
```

Nota 1.1.5. `Gen a` es un generador del tipo de dato `a`.

Cuando tenemos un generador de un determinado tipo de dato podemos hacerlo funcionar mediante

```
generate :: Gen a -> IO a
```

Veamos algunos ejemplos fáciles de tipos de datos que ya pertenecen a la clase `Arbitrary`:

```
-- >>> generate arbitrary :: IO Int
-- 28
-- >>> generate arbitrary :: IO Char
-- '\228'
-- >>> generate arbitrary :: IO [Int]
-- [5,-9,-27,15,-10,-23,20,-23,5,6,-29,-6,25,-3,-1,4,20,15,7,15]
```

Para construir generadores que dependan del tamaño, como por ejemplo los árboles que dependen de la profundidad, empleamos `sized`:

```
sized :: (Int -> Gen a) -> Gen a
```

Veamos un ejemplo de generadores que dependen del tamaño en los árboles binarios. En el caso de que no tuviéramos en cuenta el tamaño, el generador podría no acabar al generar un árbol infinito. En [8] se puede ver no sólo la implementación de generadores si no una lista de funciones y algoritmos aplicados a los árboles binarios.

Primero definimos el tipo de dato abstracto del árbol binario:

```
data Arbol a = Hoja | Nodo Int  (Arbol a) (Arbol a)
  deriving (Show,Eq)
```

Posteriormente añadimos el tipo `(Arbol a)` a la clase `Arbitrary`:

```
instance Arbitrary a => Arbitrary (Arbol a) where
arbitrary = sized arbol
```

Finalmente se construye el generador de árboles:

```
arbol 0 = return Hoja
arbol n = do
  x <- Test.QuickCheck.arbitrary
  t1 <- subarbol
  t2 <- subarbol
  return (Nodo x t1 t2)
  where subarbol = arbol (n `div` 2)
```

A continuación, introduzcamos una serie de funciones

`choose`	Escoge de manera pseudoaleatoria un elemento de un intervalo `(a,b)`.
`oneof`	Escoge de manera pseudoaleatoria un elemento de una lista.
`listOf`	Elabora una lista de elementos pseudoaleatorios de un tipo determinado.
`liftM`	Convierte una función en una mónada.
`liftM2`	Convierte una función en una mónada escaneando los argumentos de izquierda a derecha.

Por ejemplo,

```
ghci> let g = choose (0,20)
ghci> generate g :: IO Int
14
ghci> let h =listOf g
ghci> generate h :: IO [Int]
[2,6,9,2,3,12,18,13,19,2,14,0,10,13,10,4,13,1,8]
ghci> let i = oneof [h,h]
ghci> generate i :: IO [Int]
[18,15,7,6,12,5,9,1,2,12,8,5,12,0,12,14,6,1,3]
```

En una futura sección se implementarán generadores de los distintos tipos de datos que se establecen para las fórmulas de la lógica, y así poder hacer comprobaciones con QuickCheck de todo aquello que implementemos.

1.2. Librería Data.Map

Introducimos la librería `Data.Map` cuya función es el trabajo con diccionarios, permitiendo tanto la construcción de estos diccionarios, como su modificación y acceso a la información.

```
module Map where
import Data.List
import Data.Map (Map)
import qualified Data.Map as M
import Text.PrettyPrint
import Text.PrettyPrint.GenericPretty
```

Debido a que mucha de sus funciones tienen nombres coincidentes con algunas ya definidas en `Prelude`, es necesario importarla renombrándola de la siguiente manera: `import qualified Data.Map as M`. Eso implica que cuando llamemos a una función de esta librería, tendremos que hacerlo poniendo `M.(función)`.

Los diccionarios son del tipo `Map k a` y la forma de construirlos es mediante la función `(M.fromList)` seguida de una lista de pares.

```
-- | Ejemplos:
-- >>> :type M.fromList
-- M.fromList :: Ord k => [(k, a)] -> Map k a
--
-- >>> M.fromList [(1,"Pablo"),(10,"Elisabeth"),(7,"Cristina"),(0,"Luis")]
-- fromList [(0,"Luis"),(1,"Pablo"),(7,"Cristina"),(10,"Elisabeth")]
```

Una vez creado un diccionario, podemos acceder a la información registrada en él y modificarla.

El operador `(M.!)` sirve para acceder a elementos del diccionario.

```
-- | Ejemplos:
-- >>> let l = M.fromList [(1,"Pablo"),(10,"Elisabeth"),(7,"Cristina"),(0,"Luis")]
-- >>> l M.! 1
-- "Pablo"
-- >>> l M.! 10
-- "Elisabeth"
```

La función `(M.size)` devuelve el tamaño del diccionario; es decir, su número de elementos.

```
-- | Ejemplos
-- >>> let l = M.fromList [(1,"Pablo"),(10,"Elisabeth"),(7,"Cristina"),(0,"Luis")]
-- >>> M.size l
-- 4
```

La función `(M.insert)` registra un elemento en el diccionario.

```
-- | Ejemplos
-- >>> let l = M.fromList [(1,"Pablo"),(10,"Elisabeth"),(7,"Cristina"),(0,"Luis")]
-- >>> M.insert 8 "Jesus" l
-- fromList [(0,"Luis"),(1,"Pablo"),(7,"Cristina"),(8,"Jesus"),(10,"Elisabeth")]
```

El operador `(M.\\)` realiza la diferencia entre dos diccionarios.

```
-- | Ejemplos
-- >>> let l = M.fromList [(1,"Pablo"),(10,"Elisabeth"),(7,"Cristina"),(0,"Luis")]
-- >>> let l' = M.fromList [(1,"Pablo"),(7,"Cristina"),(0,"Luis")]
-- >>> l M.\\ l
-- fromList []
-- >>> l M.\\ l'
-- fromList [(10,"Elisabeth")]
```

Para determinar si un elemento pertenece a un diccionario se emplea `(M.member)`

```
-- | Ejemplos
-- >>> let l = M.fromList [(1,"Pablo"),(10,"Elisabeth"),(7,"Cristina"),(0,"Luis")]
-- >>> M.member 10 l
-- True
-- >>> M.member 3 l
-- False
```

Para localizar la definición de un elemento en un diccionario usamos la función `(M.lookup)`

```
-- | Ejemplos
-- >>> let l = M.fromList [(1,"Pablo"),(10,"Elisabeth"),(7,"Cristina"),(0,"Luis")]
-- >>> M.lookup 10 l
-- Just "Elisabeth"
```

La función `(M.adjust)` actualiza una entrada de un diccionario aplicando una función determinada.

```
-- | Ejemplos
-- >>> let l = M.fromList [(1,"Pablo"),(10,"Elizabeth"),(7,"Cristina"),(0,"Luis")]
-- >>> M.adjust (++ " the Queen") 10 l
-- fromList [(0,"Luis"),(1,"Pablo"),(7,"Cristina"),(10,"Elizabeth the Queen")]
```

.

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.

```
{-# LANGUAGE DeriveGeneric #-}

module LPH where

import Dominio
import Modelo
import Data.List
import Test.QuickCheck
import Text.PrettyPrint
import Text.PrettyPrint.GenericPretty
```

Los elementos básicos de las fórmulas en la lógica de primer orden, así como en la lógica proposicional son las variables. Por ello, definimos un tipo de dato para las variables.

Una variable estará compuesta por un nombre y un índice; es decir, nombraremos las variables como `x1`, `a1`, ...

El tipo de dato para el `nombre` lo definimos como una lista de caracteres

```
type Nombre = String
```

El tipo de dato para los índices lo definimos como lista de enteros.

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

Quedando el tipo de dato compuesto `Variable` como

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

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'

instance Out Variable where
  doc       = text . show
  docPrec _ = doc
```

Mostramos algunos ejemplos de definición de variables

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

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 se obtiene mediante la siguiente definición recursiva

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`.

```
-- | Ejemplos
-- >>> reflexiva
-- ∀x R[x,x]
-- >>> simetrica
-- ∀x ∀y (R[x,y]⟹R[y,x])
reflexiva, simetrica :: Formula
reflexiva = ParaTodo x (Atomo "R" [x,x])
simetrica = ParaTodo x (ParaTodo y ( Atomo "R" [x,y] `Implica`
                                     Atomo "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 $I(f) : \mathcal{U}^n \to \mathcal{U}$
- si P es un símbolo de relación 0–aria de L, entonces $I(P) \in \{0, 1, \}$
- si R es un símbolo de relación n–aria de L, entonces $I(R) \subseteq \mathcal{U}^n$

Para el manejo de estructuras del lenguaje, vamos a definir tipos de datos para cada uno de sus elementos.

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`. `P` y `Q` tienen dos posibles interpretaciones: Falso o verdadero. 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

El valor de `(sustituye s x d)` es la asignación obtenida a partir de la asignación `s` pero con `x` interpretado como `d`,

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

```
-- | Ejemplos
-- >>> sustituye asignacion y B z
-- A
-- >>> sustituye asignacion y B y
-- B
sustituye :: Asignacion a -> Variable -> a -> Asignacion a
sustituye s x d v | v == x    = d
                  | otherwise = s v
```

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 r vs)        = i r (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`, y dos posibles interpretaciones `interpretacion1` e `interpretacion2`.

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

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

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

Tomando como universo todas las entidades, menos la que denotamos `Inespecífico`, se tienen las siguientes evaluaciones

```
-- | Evaluaciones
-- >>> valor (take 26 entidades) interpretacion1 asignacion formula1
-- False
-- >>> 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 aquellas fórmulas de la lógica de primer orden que no están compuestas sólo por variables.

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

Algunos ejemplos de variables como términos

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

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.

```
-- | Ejemplo
-- >>> Ter "f" [tx,ty]
-- f[x,y]

instance Show Termino where
  show (Var v)    = show v
  show (Ter c []) = c
  show (Ter f ts) = f ++ show ts

instance Out Termino where
  doc       = text . show
  docPrec _ = doc
```

La propiedad `(esVariable t)` se verifica si el término `t` es una variable.

```
-- | Ejemplos
-- >>> esVariable tx
-- True
-- >>> esVariable (Ter "f" [tx,ty])
-- False
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, 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,Generic)
```

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]]))
```

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 [])     = "verdadero"
    show (Conj [f])    = show f
    show (Conj (f:fs)) = "(" ++ show f ++ "⋀" ++ show (Conj fs) ++ ")"
    show (Disy [])     = "falso"
    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)

instance Out Form where
  doc       = text . show
  docPrec _ = doc
```

Quedando las fórmulas ejemplo antes definidas de la siguiente manera

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

Previamente hemos definido `InterpretacionR`, ahora para la interpretación de los símbolos funcionales se define un nuevo tipo de dato, `InterpretacionF`.

```
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".

El valor de `(valorT i a t)` es el valor del término `t` en la interpretación `i` respecto de la asignación `a`.

```
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)
```

Para los ejemplos de evaluación de términos usaremos la siguiente interpretación de los símbolos de función

```
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
```

y la siguiente asignación

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

Con ello, se tiene

```
-- | Evaluaciones
-- >>> let t1 = Ter "cero" []
-- >>> valorT interpretacionF1 asignacion1 t1
-- 0
-- >>> let t2 = Ter "s" [t1]
-- >>> t2
-- s[cero]
-- >>> valorT interpretacionF1 asignacion1 t2
-- 1
-- >>> let t3 = Ter "mas" [t2,t2]
-- >>> t3
-- mas[s[cero],s[cero]]
-- >>> valorT interpretacionF1 asignacion1 t3
-- 2
-- >>> let t4 = Ter "por" [t3,t3]
-- >>> t4
-- por[mas[s[cero],s[cero]],mas[s[cero],s[cero]]]
-- >>> valorT interpretacionF1 asignacion1 t4
-- 4
```

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`.

```
-- | Ejemplos
-- >>> formula4
-- ∃x R[cero,x]
-- >>> formula5
-- (∀x P[x]⟹∀y Q[x,y])
formula4, formula5 :: Form
formula4 = Ex x (Atom "R" [cero,tx])
formula5 = Impl (PTodo x (Atom "P" [tx])) (PTodo y (Atom "Q" [tx,ty]))
```

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
```

En este caso se tiene las siguientes evaluaciones

```
-- | Evaluaciones
-- >>> 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

Pendiente de revisión.

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

```
module GeneradoresForm where
import PFH
import LPH
import Test.QuickCheck
import Control.Monad
```

Nuestro primer objetivo será la generación de variables. Recordemos cómo estaban definidas las variables:

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

Comprobamos que una variable está compuesta de un nombre y un índice. Así que comenzamos generando el nombre y, con este fin, generamos letras al azar entre unas cuantas elegidas de la `x` a la `w`, y cada una compondrá un nombre para una variable.

```
vars :: Nombre
vars =  "xyzuvw"

genLetra :: Gen Char
genLetra = elements vars
```

Podemos emplear como dijimos en la sección introductoria a los generadores de tipos la función `generate` para generar algunas letras al azar.

```
ghci> generate genLetra :: IO Char
'r'
ghci> generate genLetra :: IO Char
'p'
```

Con esto podemos definir el generador de nombres `genNombre`.

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

Una definición alternativa a la anterior es `genNombre2` como sigue

```
genNombre2 :: Gen Nombre
genNombre2 = do
  c <- elements vars
  return [c]
```

Finalmente, generemos algunos ejemplos de nombres para variables.

```
ghci> generate genNombre :: IO Nombre
"i"
ghci> generate genNombre :: IO Nombre
"y"
ghci> generate genNombre2 :: IO Nombre
"y"
```

Una vez que tenemos los nombres de nuestras variables, podemos generar índices. En nuestro caso limitaremos los índices al rango del 0 al 10, pues no necesitamos una cantidad mayor de variables.

Definimos un generador de enteros `genNumero` en el rango requerido.

```
genNumero :: Gen Int
genNumero = choose (0,10)
```

Y construimos el generador de índices `genIndice`.

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

Comprobemos que podemos tomar índices al azar.

```
ghci> generate genIndice :: IO Indice
[2]
ghci> generate genIndice :: IO Indice
[0]
ghci> generate genIndice :: IO Indice
[9]
```

Una vez que hemos establecido generadores de los tipos abstractos que componen las variables, podemos combinarlos en un generador de variables `generaVariable`.

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

Introducimos nuestro generador en la clase `Arbitrary`.

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

Ya podemos generar variables pseudoaleatorias, comprobemos lo que obtenemos.

```
ghci> generate generaVariable :: IO Variable
t4
ghci> generate generaVariable :: IO Variable
x5
ghci> generate generaVariable :: IO Variable
y6
ghci> generate generaVariable :: IO Variable
x4
ghci> generate generaVariable :: IO Variable
z2
```

Como ya vimos al definir las bases de la lógica, una vez que hemos definido las variables, el siguiente nivel es trabajar con términos.

Incluimos el tipo de dato `Termino` en la clase `Arbitrary`.

```
instance Arbitrary (Termino) where
    arbitrary = resize 3 (sized termino)
```

Definimos el generador de términos que tiene en cuenta el tamaño `termino n`.

```
termino :: (Num a, Ord a) => a -> Gen Termino
termino 0 = liftM Var generaVariable
termino n | n <=1 =
              liftM2 Ter genNombre (resize 3 (listOf1 (generaTermino)))
          | n > 1 =
              termino 1
    where
      generaTermino = termino (n-1)
```

Nota 2.4.3. `(resize n)` redimensiona un generador para ajustarlo a una escala.

Nota 2.4.4. Se ha acotado tanto la dimensión del generador porque no nos compensa tener términos de gran cantidad de variables o con muchos términos dentros unos de otros.

Generemos algunos términos que nos sirvan de ejemplo para comprobar el funcionamiento de nuestro generador.

```
ghci> generate (termino 0) :: IO Termino
z7
ghci> generate (termino 1) :: IO Termino
v[z1,u6]
ghci> generate (termino 2) :: IO Termino
x[z0,z10]
ghci> generate (termino 3) :: IO Termino
y[v2]
ghci> generate (termino 4) :: IO Termino
u[x0,z5,z9]
```

Para finalizar debemos implementar un generador de fórmulas. Primero lo añadiremos a la clase `Arbitrary`, y finalmente se construye el generador de fórmulas.

```
instance Arbitrary (Form) where
    arbitrary = sized formula

formula 0 = liftM2 Atom genNombre (resize 3 (listOf (termino 3)))
formula n = oneof [liftM  Neg generaFormula,
                   liftM2 Impl generaFormula generaFormula,
                   liftM2 Equiv generaFormula generaFormula,
                   liftM Conj (listOf generaFormula),
                   liftM Disy (listOf generaFormula),
                   liftM2 PTodo generaVariable generaFormula,
                   liftM2 Ex   generaVariable generaFormula]
              where
                generaFormula = formula (div n 4)
```

Algunos ejemplos de generación de fórmulas serían los siguientes.

```
ghci> generate (formula 0)
y[v[u9,x6,x8]]
ghci> generate (formula 2)
∀x4 z[x[z10]]
ghci> generate (formula 3)
¬x[x[w6,u2]]
ghci> generate (formula 4)
(¬w[w[x6,v4]]⟹∀y7 v[y[u2]])
```

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.

```
-- Ejemplos
-- >>> let t1 = Ter "f" [tx,a]
-- >>> varEnTerm t1
-- [x]
-- >>> let t2 = Ter "g" [tx,a,ty]
-- >>> varEnTerm t2
-- [x,y]
-- >>> varEnTerms [t1,t2]
-- [x,y]
varEnTerm :: Termino -> [Variable]
varEnTerm (Var v)    = [v]
varEnTerm (Ter _ ts) = varEnTerms ts

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

Nota 2.4.5. La función `(nub xs)` elimina elementos repetidos en una lista `xs`. Se encuentra en el paquete `Data.List`.

Nota 2.4.6. Se emplea un tipo de recursión cruzada entre funciones. Las funciones se llaman la una a la otra.

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

```
-- | Ejemplos
-- >>> formula2
-- ∀x ∀y (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
-- >>> varEnForm formula2
-- [x,y,z]
```

```
-- >>> formula3
-- (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
-- >>> varEnForm formula3
-- [x,y,z]
-- >>> formula4
-- ∃x R[cero,x]
-- >>> varEnForm formula4
-- [x]
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)   = varEnForm f
varEnForm (Ex x f)      = varEnForm f
```

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`.

```
-- | Ejemplos
-- >>> formula2
-- ∀x ∀y (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
-- >>> variablesLibres formula2
-- []
-- >>> formula3
-- (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
-- >>> variablesLibres formula3
-- [x,y]
-- >>> formula4
-- ∃x R[cero,x]
-- >>> variablesLibres formula4
-- []
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 en alguna subfórmula de la forma $\forall x$ o $\exists x$.

Definición 2.4.7. Una **fórmula abierta** es una fórmula con variables libres.

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

```
-- Ejemplos
-- >>> formula3
-- (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
-- >>> formulaAbierta formula3
-- True
-- >>> formula2
-- ∀x ∀y (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
-- >>> formulaAbierta formula2
-- False
formulaAbierta :: Form -> Bool
formulaAbierta = not . null . variablesLibres
```

.

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`.

```
{-# LANGUAGE DeriveGeneric #-}
module PTLP where
import LPH
import Data.List
import Test.QuickCheck -- Para ejemplos
import Generadores     -- Para ejemplos
import Text.PrettyPrint
import Text.PrettyPrint.GenericPretty
```

Antes de comenzar, se definen los átomos `p`,`q` y `r` para comodidad en los ejemplos.

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

3.1. Sustitución

Definición 3.1.1. Una **sustitución** es una aplicación $S : \mathit{Variable} \to \mathit{Termino}$.

Nota 3.1.1. $[x_1/t_1, x_2/t_2, \ldots, x_n/t_n]$ 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 xs = [x | x <- xs, Var (fst x) /= snd x]
```

Por ejemplo,

```
-- | Ejemplos
-- >>> 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,

```
-- | Ejemplos
-- >>> dominio [(x,tx)]
-- [x]
-- >>> dominio [(x,tx),(z,ty)]
-- [x,z]
-- >>> recorrido [(x,tx)]
-- [x]
-- >>> recorrido [(x,tx),(z,ty)]
-- [x,y]
```

Posteriormente, se define una función que aplica la sustitución a 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 3.1.2. $t[x_1/t_1,\ldots,x_n/t_n]$ es el término obtenido sustituyendo en t las apariciones de x_i por t_i.

Definición 3.1.3. 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 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,

```
-- | Ejemplos
-- >>> susTerm  [(x,ty)] tx
-- y
-- >>> susTerms [(x,ty),(y,tx)] [tx,ty]
-- [y,x]
```

Definición 3.1.4. $F[x_1/t_1,\ldots,x_n/t_n]$ es la fórmula obtenida sustituyendo en F las apariciones libres de x_i por t_i.

Definición 3.1.5. 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,

```
-- | Ejemplos
-- >>> formula3
-- (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
-- >>> 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,

```
-- | Ejemplos
-- >>> 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 3.1.2. Las comprobaciones con QuickCheck emplean código del módulo `Generadores`.

Definición 3.1.6. 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

```
-- | Ejemplo
-- >>> let f1 = Ex x (Atom "R" [tx,ty])
-- >>> f1
-- ∃x R[x,y]
-- >>> variablesLibres f1
-- [y]
-- >>> sustitucionForm [(y,tx)] f1
-- ∃x R[x,x]
-- >>> variablesLibres (sustitucionForm [(y,tx)] f1)
-- []
```

Un ejemplo de una sustitución libre

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

3.2. Sustitucion mediante diccionarios

Pendiente de decidir su inclusión.

En esta sección definiremos las sustituciones, de una manera alternativa, mediante la librería `Data.Map`.

```
module SustDiccionario where
import LPH
import Data.List
import Data.Map (Map)
import qualified Data.Map as M
import Text.PrettyPrint
import Text.PrettyPrint.GenericPretty
```

Debido a que muchas funciones de esta librería coinciden con funciones definidas en `Prelude`, ésta se suele importar `qualified`.

No compatible con nuestra definición de término, hay que adaptarlo

3.3. Unificación

Definición 3.3.1. Un **unificador** de dos términos t_1 y t_2 es una sustitución S tal que $S(t_1) = S(t_2)$.

Definición 3.3.2. La sustitución σ_1 es **más general** que la σ_2 si existe una sustitución σ_3 tal que $\sigma_2 = \sigma_1\sigma_3$.

Definición 3.3.3. La sustitución σ es un **unificador de máxima generalidad** (UMG) de los términos t_1 y t_2 si

- σ es un unificador de t_1 y t_2.
- σ es más general que cualquier unificador de t_1 y t_2.

Definición 3.3.4. Dadas dos listas de términos s y t son **unificables** si tienen algún unificador.

Se define la función (`unificadoresTerminos t1 t2`) que devuelve los unificadores de los términos de entrada.

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

3.4. Skolem

3.4.1. Formas normales

Definición 3.4.1. Una fórmula está en **forma normal conjuntiva** si es una conjunción de disyunciones de literales.

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

Nota 3.4.1. La forma normal conjuntiva es propia de la lógica proposicional. Por ello las fórmulas aquí definidas sólo se aplicarán a fórmulas sin cuantificadores.

Definimos la función `(enFormaNC f)` para determinar si una fórmula está en su forma normal conjuntiva.

```
enFormaNC :: Form -> Bool
enFormaNC (Atom _ _) = True
enFormaNC (Conj fs) = and [(literal f) || (esConj f) | f <- fs]
    where
      esConj (Disy fs) = all (literal) fs
      esConj _ = False
enFormaNC _ = False
```

Por ejemplo

```
-- | Ejemplos
-- >>> enFormaNC (Conj [p, Disy [q,r]])
-- True
-- >>> enFormaNC (Conj [Impl p r, Disy [q, Neg r]])
-- False
-- >>> enFormaNC (Conj [p, Disy [q, Neg r]])
-- True
```

Aplicando a una fórmula F el siguiente algoritmo se obtiene una forma normal conjuntiva de F.

1. Eliminar los bicondicionales usando la equivalencia

$$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$$

2. Eliminar los condicionales usando la equivalencia

$$A \to B \equiv \neg A \vee B$$

3. Interiorizar las negaciones usando las equivalencias

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$

$$\neg\neg A \equiv A$$

4. Interiorizar las disyunciones usando las equivalencias

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

$$(A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C)$$

Implementamos estos pasos en Haskell

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 (elimImpEquiv f1), (elimImpEquiv f2)]
elimImpEquiv (Neg f) =
  Neg (elimImpEquiv f)
elimImpEquiv (Disy fs) =
  Disy (map elimImpEquiv fs)
elimImpEquiv (Conj fs) =
  Conj (map elimImpEquiv fs)
elimImpEquiv (Ex x f) =
    Ex x (elimImpEquiv f)
elimImpEquiv (PTodo x f) =
    PTodo x (elimImpEquiv f)
```

Nota 3.4.2. Se aplica a fórmulas con cuantificadores pues la función será empleada en la sección de la forma de Skolem.

Por ejemplo,

```
-- | Ejemplo
-- >>> elimImpEquiv (Neg (Conj [p, Impl q r]))
-- ¬(p⋀(¬q⋁r))
```

Interiorizamos las negaciones mediante la función `(interiorizaNeg f)`.

```
interiorizaNeg :: Form -> Form
interiorizaNeg p@(Atom f xs) = p
interiorizaNeg (Equiv f1 f2) =
    Equiv (interiorizaNeg f1) (interiorizaNeg f2)
interiorizaNeg (Impl f1 f2) = Impl (interiorizaNeg f1) (interiorizaNeg f2)
interiorizaNeg (Neg (Disy fs)) = Conj (map (interiorizaNeg) (map (Neg) fs))
interiorizaNeg (Neg (Conj fs)) = Disy (map (interiorizaNeg) (map (Neg) fs))
interiorizaNeg (Neg (Neg f)) = interiorizaNeg f
interiorizaNeg (Neg f) = Neg (interiorizaNeg f)
interiorizaNeg (Disy fs) = Disy (map interiorizaNeg fs)
interiorizaNeg (Conj fs) = Conj (map interiorizaNeg fs)
```

Definimos `(interiorizaDisy f)` para interiorizar las disyunciones

```
interiorizaDisy :: Form -> Form
interiorizaDisy (Disy fs)
    | all (literal) fs = Disy fs
    | otherwise =
        Conj (map (aux2) (map (Disy) (concat (aux [ aux1 f | f <- fs]))))
            where
              aux [] = []
              aux (xs:xss) = map (combina xs) xss ++ aux xss
              aux1 p | literal p = [p]
              aux1 (Conj xs) = xs
              aux1 (Disy xs) = xs
              combina [] ys = []
              combina xs [] = []
              combina xs ys = [[x,y] | x <- xs, y <- ys]
              aux2 f | enFormaNC f = f
                     | otherwise = interiorizaDisy f

interiorizaDisy f = f
```

Nota 3.4.3. Explicación de las funciones auxiliares

- La función `aux` aplica la función `combina` las listas de las conjunciones.

- La función `aux1` toma las listas de las conjunciones, construye una lista de un literal o unifica disyunciones.
- La función `combina xs ys` elabora listas de dos elementos de las listas `xs` e `ys`.
- La función `aux2` itera para interiorizar todas las disyunciones.

Debido a la representación que hemos elegido, pueden darse conjunciones de conjunciones, lo cual no nos interesa. Por ello, definimos `unificacionConjuncion` que extrae la conjunción al exterior.

```
unificaConjuncion :: Form -> Form
unificaConjuncion p@(Atom _ _) = p
unificaConjuncion (Disy fs) = Disy fs
unificaConjuncion (Conj fs) = Conj (concat (map (aux) (concat xs)))
    where
      xs = [ aux f | f <- fs]
      aux (Conj xs) = xs
      aux f = [f]
```

Por ejemplo,

```
-- | Ejemplos
-- >>> let f1 = Conj [p, Conj [r,q]]
-- >>> f1
-- (p∧(r∧q))
-- >>> unificaConjuncion f1
-- (p∧(r∧q))
-- >>> unificaConjuncion f1 == (Conj [p, Conj [r,q]])
-- False
-- >>> unificaConjuncion f1 == (Conj [p,r,q])
-- True
```

Nota 3.4.4. La representación "visual" por pantalla de una conjunción de conjunciones y su unificación puede ser la misma, como en el ejemplo anterior.

Así, hemos construido el algoritmo para el cálculo de formas normales conjuntivas. Definimos la función `(formaNormalConjuntiva f)`

```
formaNormalConjuntiva :: Form -> Form
formaNormalConjuntiva =
    unificaConjuncion . interiorizaDisy . interiorizaNeg . elimImpEquiv
```

Por ejemplo

```
-- | Ejemplos
-- >>> let f1 = Neg (Conj [p, Impl q r])
-- >>> f1
-- ¬(p⋀(q⟹r))
-- >>> formaNormalConjuntiva f1
-- ((¬p⋁q)⋀(¬p⋁¬r))
-- >>> enFormaNC (formaNormalConjuntiva f1)
-- True
--
-- >>> let f2 = Neg (Conj [Disy [p,q],r])
-- >>> f2
-- ¬((p⋁q)⋀r)
-- >>> formaNormalConjuntiva f2
-- ((¬p⋁¬r)⋀(¬q⋁¬r))
-- >>> enFormaNC (formaNormalConjuntiva f2)
-- True
-- >>> let f3 = (Impl (Conj [p,q]) (Disy [Conj [Disy [r,q],Neg p], Neg r]))
-- >>> f3
-- ((p⋀q)⟹(((r⋁q)⋀¬p)⋁¬r))
-- >>> formaNormalConjuntiva f3
-- ((¬p⋁r)⋀((¬p⋁q)⋀((¬p⋁¬p)⋀((¬p⋁¬r)⋀((¬q⋁r)⋀((¬q⋁q)⋀((¬q⋁¬p)⋀(¬q⋁¬r))))
-- >>> enFormaNC (formaNormalConjuntiva f3)
-- True
```

Definición 3.4.2. Una fórmula está en **forma normal disyuntiva** si es una disyunción de conjunciones de literales.

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

3.4.2. Forma rectificada

Definición 3.4.3. Una fórmula F está en forma **rectificada** si ninguna variable aparece de manera simultánea libre y ligada y cada cuantificador se refiere a una variable diferente.

Para proceder a su implementación definimos una función auxiliar previa que denotamos `(sustAux n v f)` que efectúa una sustitución de la variable `v` por x_n.

```
sustAux :: Int -> Variable -> Form -> Form
sustAux n v (PTodo var f)
    | var == v =
        PTodo (Variable "x" [n])
      (sustAux n v (sustitucionForm [(v, Var (Variable "x" [n]))] f))
    | otherwise = sustAux (n+1) var (PTodo var f)
```

```
sustAux n v (Ex var f)
    | var == v =
        Ex (Variable "x" [n])
      (sustAux n v (sustitucionForm [(v, Var (Variable "x" [n]))] f))
    | otherwise = sustAux (n+1) var (Ex var f)
sustAux n v (Impl f1 f2) =
    Impl (sustAux n v f1) (sustAux (n+k) v f2)
    where
      k = length (varEnForm f1)
sustAux n v (Conj fs) = Conj (map (sustAux n v) fs)
sustAux n v (Disy fs) = Disy (map (sustAux n v) fs)
sustAux n v (Neg f) = Neg (sustAux n v f)
sustAux n v f = sustitucionForm [(v, Var (Variable "x" [n]))] f
```

Añadimos ejemplos

```
-- | Ejemplo
-- >>> let f1 = PTodo x (Impl (Atom "P" [tx]) (Atom "Q" [tx,ty]))
-- >>> f1
-- ∀x (P[x]⟹Q[x,y])
-- >>> sustAux 0 x f1
-- ∀x0 (P[x0]⟹Q[x0,y])
```

Definimos `(formRec n f)` que calcula la forma rectificada de la fórmula `f` empezando a renombrar las variables desde el índice `n`.

```
formRec :: Int -> Form -> Form
formRec n (PTodo v form) = sustAux n v (PTodo v (formRec (n+1) form))
formRec n (Ex v form) = sustAux n v (Ex v (formRec (n+1) form))
formRec n (Impl f1 f2) = Impl (formRec n f1) (formRec (n+k) f2)
    where
      k = length (recolectaCuant f1)
formRec n (Conj fs) = Conj [formRec (n+(k f fs)) f | f <- fs]
    where
      k f fs = length (concat (map (recolectaCuant) (takeWhile (/=f)
                                                               fs)))
formRec n (Disy fs) = Disy [formRec (n+(k f fs)) f | f <- fs]
    where
      k f fs = length (concat (map (recolectaCuant) (takeWhile (/=f)
                                                               fs)))
formRec n (Neg f) = Neg (formRec n f)
formRec n f = f
```

Por ejemplo

```
-- | Ejemplos
-- >>> formula2
-- ∀x ∀y (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
-- >>> formRec 0 formula2
-- ∀x0 ∀x1 (R[x0,x1]⟹∃x4 (R[x0,x4]⋀R[x4,x1]))
-- >>> formula3
-- (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
-- >>> formRec 0 formula3
-- (R[x,y]⟹∃x0 (R[x,x0]⋀R[x0,y]))
```

3.4.3. Forma normal prenexa

Definición 3.4.4. Una fórmula F está en forma **normal prenexa** si es de la forma $Q_1x_1 \dots Q_nx_nG$ donde $Q_i \in \{\forall, \exists\}$ y G no tiene cuantificadores.

Para la definición de la forma normal prenexa requerimos de dos funciones previas. Una que elimine los cuantificadores, `(eliminaCuant f)`, y otra que los recolecta en una lista ,`(recolectaCuant f)`.

La definición de `eliminaCuant f` es

```
eliminaCuant :: Form -> Form
eliminaCuant (Ex x f) = eliminaCuant f
eliminaCuant (PTodo x f) = eliminaCuant f
eliminaCuant (Conj fs) = Conj (map eliminaCuant fs)
eliminaCuant (Disy fs) = Disy (map eliminaCuant fs)
eliminaCuant (Neg f) = Neg (eliminaCuant f)
eliminaCuant (Impl f1 f2) = Impl (eliminaCuant f1) (eliminaCuant f2)
eliminaCuant (Equiv f1 f2) = Equiv (eliminaCuant f1) (eliminaCuant f2)
eliminaCuant p@(Atom _ _) = p
```

Algunos ejemplos

```
-- | Ejemplos
-- >>> eliminaCuant formula2
-- (R[x,y]⟹(R[x,z]⋀R[z,y]))
-- >>> eliminaCuant formula3
-- (R[x,y]⟹(R[x,z]⋀R[z,y]))
```

La implementación de `(recolectaCuant f)` es

```
recolectaCuant :: Form -> [Form]
recolectaCuant (Ex x f) = (Ex x p): recolectaCuant f
recolectaCuant (PTodo x f) = (PTodo x p): recolectaCuant f
```

```
recolectaCuant (Conj fs) = concat (map recolectaCuant fs)
recolectaCuant (Disy fs) = concat (map recolectaCuant fs)
recolectaCuant (Neg f) = recolectaCuant f
recolectaCuant (Impl f1 f2) = recolectaCuant f1 ++ recolectaCuant f2
recolectaCuant p@(Atom _ _) = []
recolectaCuant (Equiv f1 f2) = recolectaCuant f1 ++ recolectaCuant f2
```

Por ejemplo,

```
-- | Ejemplos
-- >>> recolectaCuant formula2
-- [∀x p,∀y p,∃z p]
-- >>> recolectaCuant formula3
-- [∃z p]
```

Definimos la función `formaNormalPrenexa f` que calcula la forma normal prenexa de la fórmula `f`

```
formaNormalPrenexa :: Form -> Form
formaNormalPrenexa f = aplica cs (eliminaCuant (formRec 0 f))
    where
      aplica [] f = f
      aplica ((PTodo x _):fs) f = aplica fs (PTodo x f)
      aplica ((Ex x _):fs) f = aplica fs (Ex x f)
      cs = reverse (recolectaCuant (formRec 0 f))
```

Por ejemplo,

```
-- | Ejemplos
-- >>> formaNormalPrenexa formula2
-- ∀x0 ∀x1 ∃x4 (R[x0,x1]⟹(R[x0,x4]⋀R[x4,x1]))
-- >>> formaNormalPrenexa formula3
-- ∃x0 (R[x,y]⟹(R[x,x0]⋀R[x0,y]))
```

3.4.4. Forma normal prenexa conjuntiva

Definición 3.4.5. Una fórmula F está en **forma normal prenexa conjuntiva** si está en forma normal prenexa con G en forma normal conjuntiva.

La implementamos en `Haskell` mediante la función `(formaNPConjuntiva f)`

```
formaNPConjuntiva :: Form -> Form
formaNPConjuntiva f = aux (formaNormalPrenexa f)
    where
      aux (PTodo v f) = PTodo v (aux f)
      aux (Ex v f) = Ex v (aux f)
      aux f = formaNormalConjuntiva f
```

Por ejemplo,

```
-- | Ejemplos
-- >>> formula2
-- ∀x ∀y (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
-- >>> formaNPConjuntiva formula2
-- ∀x0 ∀x1 ∃x4 ((¬R[x0,x1]⋁R[x0,x4])⋀(¬R[x0,x1]⋁R[x4,x1]))
-- >>> formula3
-- (R[x,y]⟹∃z (R[x,z]⋀R[z,y]))
-- >>> formaNPConjuntiva formula3
-- ∃x0 ((¬R[x,y]⋁R[x,x0])⋀(¬R[x,y]⋁R[x0,y]))
```

3.4.5. Forma de Skolem

Definición 3.4.6. 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 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 3.4.7. Las fórmulas `F` y `G` son **equivalentes** si para toda interpretación valen lo mismo.

Definición 3.4.8. Las fórmulas `F` y `G` son **equisatisfacibles** si se cumple que ambas son satisfacibles o ninguna lo es.

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 están compuestos por las siglas "sk" y un entero que lo identifique.

Nota 3.4.5. 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 3.4.9. 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  = aux. sk . elimImpEquiv
    where
      aux (Neg (Neg f)) = f
      aux (Neg (Ex v f)) = (PTodo v f)
      aux (Disy fs) = Disy (map aux fs)
      aux f = f
```

Por ejemplo,

```
-- | Ejemplos
-- >>> skolem formula2
-- ∀x ∀y (¬R[x,y]⋁(R[x,sk0[x,y]]⋀R[sk0[x,y],y]))
-- >>> skolem formula3
-- (¬R[x,y]⋁(R[x,sk0]⋀R[sk0,y]))
-- >>> skolem formula4
-- R[cero,sk0]
-- >>> skolem formula5
-- (¬P[sk0]⋁∀y Q[x,y])
-- >>> skolem (Neg (Ex x (Impl (Atom "P" [tx]) (PTodo x (Atom "P" [tx])))))
-- ∀x (¬P[x]⋁P[sk0[x]])
-- >>> let f1 = Impl (Neg (Disy [PTodo x (Impl (Atom "P" [tx]) (Atom "Q" [tx])),PTodo
-- >>> skolem f1
-- ((∀x (¬P[x]⋁Q[x])⋁∀x (¬Q[x]⋁R[x]))⋁∀x (¬P[x]⋁R[x]))
```

3.5. Tableros semánticos

Definición 3.5.1. Un conjunto de fórmulas cerradas 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 en el que se pueden acometer para una simplicación eficiente de una fórmula a otras. Así se obtendrán fórmulas 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 3.5.1. Cada elemento de la izquierda de las tablas es equivalente a la entrada de la derecha de la tabla que esté en su misma altura.

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 3.5.2. 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
```

Estas definiciones de literales nos permiten distinguir entre literales positivos y literales negativos, lo cual será necesario a la hora de construir un tablero como veremos muy pronto.

Necesitamos también poder reconocer las dobles negaciones, para ello definimos la función `dobleNeg f`.

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

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

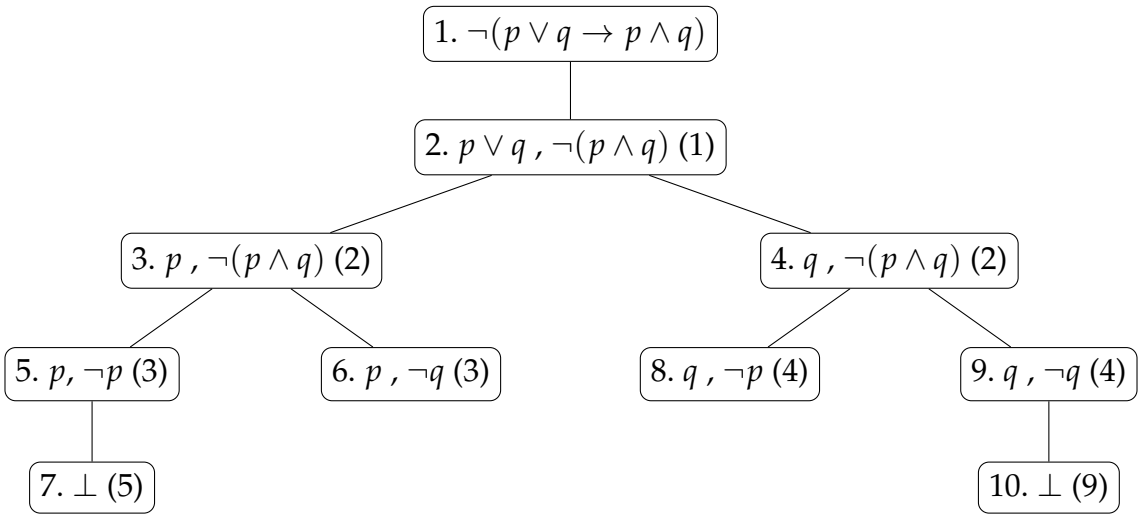
- El árbol cuyo único nodo tiene como etiqueta S es un tablero de S.
- Sea $\mathcal{T}$ un tablero de S y S_1 la etiqueta de una hoja de $\mathcal{T}$.
  1. Si S_1 contiene una fórmula y su negación, entonces el árbol obtenido añadiendo como hijo de S_1 el nodo etiquetado con $\{\bot\}$ es un tablero 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 3.5.3. Se dice que una hoja es **cerrada** si contiene una fórmula y su negación. Se representa $\bot$

Definición 3.5.4. Se dice que una hoja es **abierta** si es un conjunto de literales y no contiene un literal y su negación.

Definición 3.5.5. Un **tablero completo** es un tablero tal que todas sus hojas son abiertas o cerradas.

Ejemplo de tablero completo



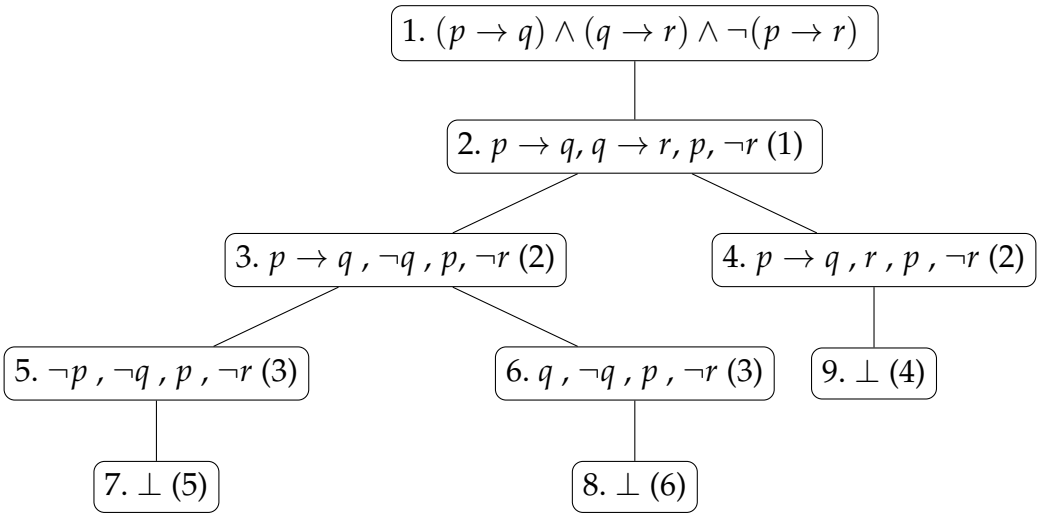
Representamos la fórmula de este tablero en Haskell.

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

-- | Representación
-- >>> tab1
-- ¬((p⋁q)⟹(p⋀q))
```

Definición 3.5.6. 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)]
-- | Representación
-- >>> tab2
-- ((p⟹q)⋀((q⟹r)⋀¬(p⟹r)))
```

Teorema 3.5.1. *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`. 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.

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 representa los negativos, y la lista de fórmulas son aquellas ligadas a los términos de las listas anteriores. Esta separación de los literales por signo es necesaria para la unificación en tableros.

Una vez definidos los nodos, definimos los tableros como una lista de ellos.

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

Una función auxiliar de conversión de literales a términos es `listeralATer t`.

```
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,

```
-- | Ejemplos
-- >>> componentes (skolem (tab1))
-- [¬¬(p⋁q),¬(p⋀q)]
-- >>> 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,

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

Definimos `(ramificacion nodo)` que ramifica un nodo aplicando las equivalencias adecuadas.

```
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 puede 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 γ, algo así como el número de veces que se aplica una regla γ.

```
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 3.5.7. Un nodo está completamente **expandido** si no se puede seguir ramificando

Caracterizamos cuando un nodo está expandido mediante la función `(nodoExpandido nd)`.

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

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

Nota 3.5.2. Se aplica el paso de expansión al primer nodo que lo necesite hasta que la profundidad γ se vea reducida y cuando ésto sucedese cambia de nodo al siguiente. Así los nodos son expandidos de manera regular. Este proceso se sigue recursivamente hasta que `n` llega a 0 o el tablero está completamente expandido.

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
```

Un nodo "cierra" cuando es posible unificar uno de sus literales positivos con uno de los literales negativos, así que es interesante y necesario poder coleccionar todas aquellas sustituciones para la unificación. Aquí vemos la motivación de la separación que anteriormente comentamos en dos listas, una de literales positivos y otra con los negativos.

Definimos la función `(sustDeUnifTab)` para construir la lista de sustituciones de unificación de un nodo.

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

Nota 3.5.3. Como los literales se han representado como términos, hemos podido aplicar la función `unificadoresTerminos`.

Como hemos definido una función para generar la lista de unificaciones, ahora tiene sentido definir las funciones auxiliares `(sustNodo sust nd)` y `(sustTab sut 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 t)` para determinar si un tablero es cerrado. Esta función construye una lista de sustituciones de unificación para todos sus nodos, así que un tablero será cerrado si su lista generada por esta función es no vacía.

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

Dada una fórmula a la que queremos construir su tablero asociado, 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,

```
-- | Ejemplos
-- >>> tab1
-- ¬((p⋁q)⟹(p⋀q))
-- >>> 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)
--  where tab = expandeTableroG k (tableroInicial f)
```

Nota 3.5.4. Se puede emplear tambien `expandeTableroG`, por ello se deja comentado para su posible uso.

Definición 3.5.8. 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)`, respectivamente.

```
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

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

Teorema 3.5.2. *El cálculo de tableros semánticos es adecuado y completo.*

Definición 3.5.9. Una fórmula F es **deducible** a partir del conjunto de fórmulas S si existe un tablero completo cerrado de la conjunción de S y $\neg F$. Se representa por $S \vdash_{Tab} F$.

Capítulo 4

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`

```
{-# LANGUAGE DeriveGeneric #-}
module Herbrand where
import Data.List
import Text.PrettyPrint.GenericPretty
import PFH
import LPH
import PTLP
import Tableros
```

4.1. Universo de Herbrand

Definición 4.1.1. La **aridad** de un operador $f(x_1, \ldots, x_n)$ es el número número de argumentos a los que se aplica.

Definición 4.1.2. Una **signatura** es una terna formada por las constantes, símbolos funcionales y símbolos de relación. Teniendo en cuenta la aridad tanto de los símbolos funcionales como los de relación.

Se define un tipo de dato para la signatura, cuya estrucura es

```
( constantes, (funciones, aridad) , (relaciones, aridad) )
```

```
type Signatura = ([Nombre],[(Nombre,Int)],[(Nombre,Int)])
```

Dada una signatura, se procede a definir la función `(unionSignatura s1 s2)` que une las signaturas `s1` y `s2`.

```
-- | Ejemplos
-- >>> let s1 = (["a"],[("f",1)],[])
-- >>> let s2 = (["b","c"],[("f",1),("g",2)],[("R",2)])
-- >>> unionSignatura s1 s2
-- (["a","b","c"],[("f",1),("g",2)],[("R",2)])
```

Su definición es

```
unionSignatura :: Signatura -> Signatura -> Signatura
unionSignatura (cs1,fs1,rs1) (cs2,fs2,rs2) =
    ( cs1 `union` cs2
    , fs1 `union` fs2
    , rs1 `union` rs2 )
```

Generalizamos la función anterior a la unión de una lista de signaturas mediante la función `(unionSignaturas ss)`.

```
-- | Ejemplos
-- >>> let s1 = (["a"],[("f",1)],[])
-- >>> let s2 = (["b","c"],[("f",1),("g",2)],[("R",2)])
-- >>> let s3 = (["a","c"],[],[("P",1)])
-- >>> unionSignaturas [s1,s2,s3]
-- (["a","b","c"],[("f",1),("g",2)],[("R",2),("P",1)])
```

Su definición es

```
unionSignaturas :: [Signatura] -> Signatura
unionSignaturas = foldr unionSignatura ([], [], [])
```

Se define la función `(signaturaTerm t)` que determina la signatura del término `t`.

```
-- | Ejemplos
-- >>> signaturaTerm tx
-- ([],[],[])
-- >>> signaturaTerm a
-- (["a"],[],[])
-- >>> let t1 = Ter "f" [a,tx,Ter "g" [b,a]]
-- >>> t1
-- f[a,x,g[b,a]]
-- >>> signaturaTerm t1
-- (["a","b"],[("f",3),("g",2)],[])
```

Su definición es

```
signaturaTerm :: Termino -> Signatura
signaturaTerm (Var _) = ([], [], [])
signaturaTerm (Ter c []) = ([c], [], [])
signaturaTerm (Ter f ts) = (cs,[(f,n)] `union` fs, rs)
    where
      (cs,fs,rs) = unionSignaturas (map signaturaTerm ts)
      n          = length ts
```

Una vez que podemos calcular la signatura de términos de una fórmula, se define la signatura de una fórmula `f` mediante la función `(signaturaForm f)`.

```
-- | Ejemplos
-- >>> let f1 = Atom "R" [a,tx,Ter "g" [b,a]]
-- >>> f1
-- R[a,x,g[b,a]]
-- >>> signaturaForm f1
-- (["a","b"],[("g",2)],[("R",3)])
-- >>> signaturaForm (Neg f1)
-- (["a","b"],[("g",2)],[("R",3)])
-- >>> let f2 = Atom "P" [b]
-- >>> let f3 = Impl f1 f2
-- >>> f3
-- (R[a,x,g[b,a]]⟹P[b])
-- >>> signaturaForm f3
-- (["a","b"],[("g",2)],[("R",3),("P",1)])
-- >>> let f4 = Conj [f1,f2,f3]
-- >>> f4
-- (R[a,x,g[b,a]]⋀(P[b]⋀(R[a,x,g[b,a]]⟹P[b])))
-- >>> signaturaForm f4
-- (["a","b"],[("g",2)],[("R",3),("P",1)])
-- >>> let f5 = PTodo x f4
-- >>> f5
-- ∀x (R[a,x,g[b,a]]⋀(P[b]⋀(R[a,x,g[b,a]]⟹P[b])))
-- >>> signaturaForm f5
-- (["a","b"],[("g",2)],[("R",3),("P",1)])
```

Su definición es

```
signaturaForm :: Form -> Signatura
signaturaForm (Atom r ts) =
  (cs,fs, [(r,n)] `union` rs)
  where (cs,fs,rs) = unionSignaturas (map signaturaTerm ts)
        n          = length ts
signaturaForm (Neg f) =
  signaturaForm f
signaturaForm (Impl f1 f2) =
```

```
  signaturaForm f1 `unionSignatura` signaturaForm f2
signaturaForm (Equiv f1 f2) =
  signaturaForm f1 `unionSignatura` signaturaForm f2
signaturaForm (Conj fs) =
  unionSignaturas (map signaturaForm fs)
signaturaForm (Disy fs) =
  unionSignaturas (map signaturaForm fs)
signaturaForm (PTodo _ f) =
  signaturaForm f
signaturaForm (Ex _ f) =
  signaturaForm f
```

Generalizamos la función anterior a una lista de fórmulas con la función `(signaturaForms fs)`.

```
-- | Ejemplos
-- >>> let f1 = Atom "R" [Ter "f" [tx]]
-- >>> let f2 = Impl f1 (Atom "Q" [a,Ter "f" [b]])
-- >>> let f3 = Atom "S" [Ter "g" [a,b]]
-- >>> signaturaForms [f1,f2,f3]
-- (["a","b"],[("f",2),("g",2)],[("R",1),("Q",2),("S",1)])
```

Su definición es

```
signaturaForms :: [Form] -> Signatura
signaturaForms fs =
  unionSignaturas (map signaturaForm fs)
```

El cálculo de la signatura de fórmulas y listas de ellas nos permite definir posteriormente el cálculo del universo de Herbrand.

Definición 4.1.3. El **universo de Herbrand** de L es el conjunto de términos básicos de F. Se reprenta por $\mathcal{UH}(L)$.

Proposición 4.1.1. $\mathcal{UH}(L) = \bigcup_{i\geq 0} H_i(L)$, donde $H_i(L)$ es el nivel i del $\mathcal{UH}(L)$ definido por

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

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

Definimos la función `(universoHerbrand n s)` que es el universo de Herbrand de la signatura `s` a nivel `n`.

```
-- | Ejemplos
-- >>> let s1 = (["a","b","c"],[],[])
```

```
-- >>> universoHerbrand 0 s1
-- [a,b,c]
-- >>> universoHerbrand 1 s1
-- [a,b,c]
-- >>> let s2 = ([],[("f",1)],[])
-- >>> universoHerbrand 0 s2
-- [a]
-- >>> universoHerbrand 1 s2
-- [a,f[a]]
-- >>> universoHerbrand 2 s2
-- [a,f[a],f[f[a]]]
-- >>> let s3 = (["a","b"],[("f",1),("g",1)],[])
-- >>> universoHerbrand 0 s3
-- [a,b]
-- >>> universoHerbrand 1 s3
-- [a,b,f[a],f[b],g[a],g[b]]
-- >>> pp $ universoHerbrand 2 s3
-- [a,b,f[a],f[b],g[a],g[b],f[f[a]],f[f[b]],f[g[a]],
--  f[g[b]],g[f[a]],g[f[b]],g[g[a]],g[g[b]]]
-- >>> universoHerbrand 3 (["a"],[("f",1)],[("R",1)])
-- [a,f[a],f[f[a]],f[f[f[a]]]]
-- >>> pp $ universoHerbrand 3 (["a","b"],[("f",1),("g",1)],[])
-- [a,b,f[a],f[b],g[a],g[b],f[f[a]],f[f[b]],f[g[a]],
--  f[g[b]],g[f[a]],g[f[b]],g[g[a]],g[g[b]],f[f[f[a]]],
--  f[f[f[b]]],f[f[g[a]]],f[f[g[b]]],f[g[f[a]]],
--  f[g[f[b]]],f[g[g[a]]],f[g[g[b]]],g[f[f[a]]],
--  g[f[f[b]]],g[f[g[a]]],g[f[g[b]]],g[g[f[a]]],
--  g[g[f[b]]],g[g[g[a]]],g[g[g[b]]]]
-- >>> let s4 = (["a","b"],[("f",2)],[])
-- >>> universoHerbrand 0 s4
-- [a,b]
-- >>> universoHerbrand 1 s4
-- [a,b,f[a,a],f[a,b],f[b,a],f[b,b]]
-- >>> pp $ universoHerbrand 2 s4
-- [a,b,f[a,a],f[a,b],f[b,a],f[b,b],f[a,f[a,a]],
--  f[a,f[a,b]],f[a,f[b,a]],f[a,f[b,b]],f[b,f[a,a]],
--  f[b,f[a,b]],f[b,f[b,a]],f[b,f[b,b]],f[f[a,a],a],
--  f[f[a,a],b],f[f[a,a],f[a,a]],f[f[a,a],f[a,b]],
--  f[f[a,a],f[b,a]],f[f[a,a],f[b,b]],f[f[a,b],a],
--  f[f[a,b],b],f[f[a,b],f[a,a]],f[f[a,b],f[a,b]],
--  f[f[a,b],f[b,a]],f[f[a,b],f[b,b]],f[f[b,a],a],
--  f[f[b,a],b],f[f[b,a],f[a,a]],f[f[b,a],f[a,b]],
--  f[f[b,a],f[b,a]],f[f[b,a],f[b,b]],f[f[b,b],a],
--  f[f[b,b],b],f[f[b,b],f[a,a]],f[f[b,b],f[a,b]],
--  f[f[b,b],f[b,a]],f[f[b,b],f[b,b]]]
```

Su implementación es

```
universoHerbrand :: Int -> Signatura -> [Termino]
universoHerbrand 0 (cs,_,_)
  | null cs   = [a]
  | otherwise = [Ter c [] | c <- cs]
universoHerbrand n s@(_,fs,_) =
  u `union`
  [Ter f ts | (f,k) <- fs
            , ts <- variacionesR k u]
  where u = universoHerbrand (n-1) s
```

Se define el universo de Herbrand de una fórmula `f` a nivel `n` mediante la función `(universoHerbrandForm n f)`.

```
-- | Ejemplos
-- >>> let f1 = Atom "R" [a,b,c]
-- >>> universoHerbrandForm 1 f1
-- [a,b,c]
-- >>> let f2 = Atom "R" [Ter "f" [tx]]
-- >>> universoHerbrandForm 2 f2
-- [a,f[a],f[f[a]]]
-- >>> let f3 = Impl f2 (Atom "Q" [a,Ter "g" [b]])
-- >>> f3
-- (R[f[x]]⟹Q[a,g[b]])
-- >>> pp $ universoHerbrandForm 2 f3
-- [a,b,f[a],f[b],g[a],g[b],f[f[a]],f[f[b]],f[g[a]],
--  f[g[b]],g[f[a]],g[f[b]],g[g[a]],g[g[b]]]
-- >>> let f4 = Atom "R" [Ter "f" [a,b]]
-- >>> signaturaForm f4
-- (["a","b"],[("f",2)],[("R",1)])
-- >>> pp $ universoHerbrandForm 2 f4
-- [a,b,f[a,a],f[a,b],f[b,a],f[b,b],f[a,f[a,a]],
--  f[a,f[a,b]],f[a,f[b,a]],f[a,f[b,b]],f[b,f[a,a]],
--  f[b,f[a,b]],f[b,f[b,a]],f[b,f[b,b]],f[f[a,a],a],
--  f[f[a,a],b],f[f[a,a],f[a,a]],f[f[a,a],f[a,b]],
--  f[f[a,a],f[b,a]],f[f[a,a],f[b,b]],f[f[a,b],a],
--  f[f[a,b],b],f[f[a,b],f[a,a]],f[f[a,b],f[a,b]],
--  f[f[a,b],f[b,a]],f[f[a,b],f[b,b]],f[f[b,a],a],
--  f[f[b,a],b],f[f[b,a],f[a,a]],f[f[b,a],f[a,b]],
--  f[f[b,a],f[b,a]],f[f[b,a],f[b,b]],f[f[b,b],a],
--  f[f[b,b],b],f[f[b,b],f[a,a]],f[f[b,b],f[a,b]],
--  f[f[b,b],f[b,a]],f[f[b,b],f[b,b]]]
```

A continuación, su definición es

```
universoHerbrandForm :: Int -> Form -> [Termino]
universoHerbrandForm n f =
  universoHerbrand n (signaturaForm f)
```

Se generaliza la definición anterior a una lista de fórmulas mediante la función `(universoHerbrandForms n fs)`

```
-- | Ejemplos
-- >>> let f1 = Atom "R" [Ter "f" [tx]]
-- >>> let f2 = Impl f1 (Atom "Q" [a,Ter "f" [b]])
-- >>> let f3 = Atom "S" [Ter "g" [a,b]]
-- >>> universoHerbrandForms 1 [f1,f2,f3]
-- [a,f[a],b,f[b],g[a,a],g[a,b],g[b,a],g[b,b]]
```

Siendo su definición

```
universoHerbrandForms :: Int -> [Form] -> [Termino]
universoHerbrandForms n fs =
  nub (concatMap (universoHerbrandForm n) fs)
```

Proposición 4.1.2. $\mathcal{UH}$ es finito si y sólo si no tiene símbolos de función.

4.2. Base de Herbrand

Definición 4.2.1. Una **fórmula básica** es una fórmula sin variables ni cuantificadores.

Definición 4.2.2. La **base de Herbrand** $\mathcal{BH}(L)$ de un lenguaje L es el conjunto de átomos básicos de L.

Definimos un tipo de dato para las bases de Herbrand

```
type BaseH = [Form]
```

Implementamos la base de Herbrand a nivel `n` de la signatura `s` mediante la función `(baseHerbrand n s)`

```
-- | Ejemplos
-- >>> let s1 = (["a","b","c"],[],[("P",1)])
-- >>> baseHerbrand 0 s1
-- [P[a],P[b],P[c]]
-- >>> let s2 = (["a","b","c"],[],[("P",1),("Q",1),("R",1)])
-- >>> let s2 = (["a","b","c"],[("f",1)],[("P",1),("Q",1),("R",1)])
-- >>> baseHerbrand 0 s2
-- [P[a],P[b],P[c],Q[a],Q[b],Q[c],R[a],R[b],R[c]]
-- >>> pp $ baseHerbrand 1 s2
-- [P[a],P[b],P[c],P[f[a]],P[f[b]],P[f[c]],Q[a],Q[b],
--  Q[c],Q[f[a]],Q[f[b]],Q[f[c]],R[a],R[b],R[c],R[f[a]],
--  R[f[b]],R[f[c]]]
```

Se implementa en Haskell a continuación

```
baseHerbrand :: Int -> Signatura -> BaseH
baseHerbrand n s@(_,_,rs) =
  [Atom r ts | (r,k) <- rs
             , ts <- variacionesR k u]
  where u = universoHerbrand n s
```

Se define la base de Herbrand de una fórmula `f` a nivel `n` mediante `(baseHerbrandForm n f)`.

```
-- | Ejemplos
-- >>> let f1 = Atom "P" [Ter "f" [tx]]
-- >>> f1
-- P[f[x]]
-- >>> baseHerbrandForm 2 f1
-- [P[a],P[f[a]],P[f[f[a]]]]
```

Su definición es

```
baseHerbrandForm :: Int -> Form -> BaseH
baseHerbrandForm n f =
  baseHerbrand n (signaturaForm f)
```

Generalizamos la función anterior a una lista de fórmulas definiendo `(baseHerbrandForms n fs)`

```
-- | Ejemplos
-- >>> let f1 = Atom "P" [Ter "f" [tx]]
-- >>> let f2 = Atom "Q" [Ter "g" [b]]
-- >>> baseHerbrandForms 1 [f1,f2]
-- [P[b],P[f[b]],P[g[b]],Q[b],Q[f[b]],Q[g[b]]]
```

Su definición es

```
baseHerbrandForms :: Int -> [Form] -> BaseH
baseHerbrandForms n fs =
  baseHerbrandForm n (Conj fs)
```

4.3. Interpretacion de Herbrand

Definición 4.3.1. Una **interpretación de Herbrand** 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.

Definimos un tipo de dato para los elementos que componen la interpretación de Herbrand.

```
type UniversoH = Universo Termino

type InterpretacionHR = InterpretacionR Termino

type InterpretacionHF = InterpretacionF Termino

type InterpretacionH = (InterpretacionHR, InterpretacionHF)
```

```
type AtomosH = [Form]
```

Se define la interpretación de Herbrand de un conjunto de átomos de Herbrand a través de `(interpretacionH fs)`

```
-- | Ejemplos
-- >>> let f1 = Atom "P" [a]
-- >>> let f2 = Atom "P" [Ter "f" [a,b]]
-- >>> let fs = [f1,f2]
-- >>> let (iR,iF) = interpretacionH fs
-- >>> iF "f" [a,c]
-- f[a,c]
-- >>> iR "P" [a]
-- True
-- >>> iR "P" [b]
-- False
-- >>> iR "P" [Ter "f" [a,b]]
-- True
-- >>> iR "P" [Ter "f" [a,a]]
-- False
```

Se implementa en Haskell

```
interpretacionH :: AtomosH -> InterpretacionH
interpretacionH fs = (iR,iF)
  where iF f ts   = Ter  f ts
        iR r ts   = Atom r ts `elem` fs
```

Proposición 4.3.1. Una interpretación de Herbrand queda determinada por un subconjunto de la base de Herbrand.

Evaluamos una fórmula a través de una interpertación de Herbrand. Para ello definimos la función `(valorH u i f)`; donde `u` representa el universo, `i` la interpretación, y `f` la fórmula.

```
valorH :: UniversoH -> InterpretacionH -> Form -> Bool
valorH u i f =
  valorF u i s f
  where s _ = a
```

4.4. Modelos de Herbrand

Definición 4.4.1. 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.

Implementamos los subconjuntos del n-ésimo nivel de la base de Herbrand de la fórmula `f` que son modelos de `f` con la función `(modelosHForm n f)`.

```
-- | Ejemplos
-- >>> let f1 = Disy [Atom "P" [a], Atom "P" [b]]
-- >>> f1
-- (P[a]⋁P[b])
-- >>> modelosHForm 0 f1
-- [[P[a]],[P[b]],[P[a],P[b]]]
-- >>> let f2 = Impl (Atom "P" [a]) (Atom "P" [b])
-- >>> f2
-- (P[a]⟹P[b])
-- >>> modelosHForm 0 f2
-- [[],[P[b]],[P[a],P[b]]]
-- >>> let f3 = Conj [Atom "P" [a], Atom "P" [b]]
-- >>> f3
-- (P[a]⋀P[b])
-- >>> modelosHForm 0 f3
```

```
-- [[P[a],P[b]]]
-- >>> let f4 = PTodo x (Impl (Atom "P" [tx]) (Atom "Q" [Ter "f" [tx]]))
-- >>> f4
-- ∀x (P[x]⟹Q[f[x]])
-- >>> modelosHForm 0 f4
-- [[],[Q[a]]]
-- >>> modelosHForm 1 f4
-- [[],[Q[a]],[Q[f[a]]],[P[a],Q[f[a]]],[Q[a],Q[f[a]]],[P[a],Q[a],Q[f[a]]]]
-- >>> length (modelosHForm 2 f4)
-- 18
```

Lo definimos en Haskell

```
modelosHForm :: Int -> Form -> [AtomosH]
modelosHForm n f =
 [fs | fs <- subsequences bH
     , valorH uH (interpretacionH fs) f]
  where uH = universoHerbrandForm n f
        bH = baseHerbrandForm n f
```

Generalizamos la definición anterior a una lista de fórmulas mediante la función `(modelosH n fs)`.

```
-- | Ejemplos
-- >>> let f1 = PTodo x (Impl (Atom "P" [tx]) (Atom "Q" [Ter "f" [tx]]))
-- >>> f1
-- ∀x (P[x]⟹Q[f[x]])
-- >>> let f2 = Ex x (Atom "P" [tx])
-- >>> f2
-- ∃x P[x]
-- >>> modelosH 0 [f1,f2]
-- []
-- >>> modelosH 1 [f1,f2]
-- [[P[a],Q[f[a]]],[P[a],Q[a],Q[f[a]]]]
```

Su definición es

```
modelosH :: Int -> [Form] -> [AtomosH]
modelosH n fs =
  [gs | gs <- subsequences bH
      , and [valorH uH (interpretacionH gs) f | f <- fs]]
  where uH = universoHerbrandForms n fs
        bH = baseHerbrandForms n fs
```

4.5. Consistencia mediante modelos de Herbrand

Proposición 4.5.1. Sea S un conjunto de fórmulas básicas. Son equivalentes:

1. S es consistente.
2. S tiene un modelo de Herbrand.

Proposición 4.5.2. Existen conjuntos de fórmulas consistentes sin modelos de Herbrand.

Un ejemplo de fórmula consistente sin modelo de Herbrand

```
formula10 :: Form
formula10 = Conj [Ex x (Atom "P" [tx]), Neg (Atom "P" [a])]
```

Como podemos ver aplicando `modelosH`

```
ghci> formula10
(∃x P[x]⋀¬P[a])
ghci> modelosH 0 [formula10]
[]
ghci> modelosH 1 [formula10]
[]
ghci> modelosH 2 [formula10]
[]
ghci> modelosH 3 [formula10]
[]
```

Pero es satisfacible

```
ghci> satisfacible 0 formula10
True
```

4.6. Extensiones de Herbrand

Definición 4.6.1. Sea $C = \{L_1, ; L_n\}$ una cláusula de L y σ una sustitución de L. Entonces, $C\sigma = \{L_1\sigma, \ldots, L_n\sigma\}$ es una **instancia** de C.

Definición 4.6.2. $C\sigma$ es una **instancia básica** de C si todos los literales de $C\sigma$ son básicos.

Por ejemplo, si tenemos $C = \{P(x,a), \neg P(x, f(y))\}$, una instancia básica sería

$$C[x/a, y/f(a)] = \{P(a,a), \neg P(x, f(f(a)))\}$$

Que en haskell lo habríamos representado por

```
ghci> Conj [Atom "P" [tx,a],Neg (Atom "P" [tx,Ter "f" [ty]])]
(P[x,a]⋀¬P[x,f[y]])
ghci> sustitucionForm [(x,a),(y,Ter "f" [a])]
        (Conj [Atom "P" [tx,a], Neg (Atom "P" [tx,Ter "f" [ty]])])
(P[a,a]⋀¬P[a,f[f[a]]])
```

Definición 4.6.3. La **extensión de Herbrand** de un conjunto de cláusulas Cs es el conjunto de fórmulas

$$EH(Cs) = \{C\sigma : C \in Cs \text{ y }, \forall x \in C, \sigma(x) \in UH(Cs)\}$$

Proposición 4.6.1. $EH(L) = \cup_{i \geq 0} EH_i(L)$, donde $EH_i(L)$ es el nivel i de la $EH(L)$

$$EH_i(Cs) = \{C\sigma : C \in Cs \text{ y }, \forall x \in C, \sigma(x) \in UH_i(Cs)\}$$

Capítulo 5

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 Data.List
import LPH
import PTLP
```

5.1. Forma clausal

Para la implementación de la resolución, primero debemos definir una serie de conceptos y construir la forma clausal.

Definición 5.1.1. Un **literal** es un átomo o la negación de un átomo.

Definición 5.1.2. Una **cláusula** es un conjunto finito de literales.

Definición 5.1.3. Una **forma clausal** de una fórmula F es un conjunto de cláusulas equivalente a F.

Proposición 5.1.1. Si $(p_1 \vee \cdots \vee p_n) \wedge \cdots \wedge (q_1 \vee \cdots \vee q_m)$ es una forma normal conjuntiva de la fórmula F. Entonces, una forma clausal de F es $\{(p_1 \vee \cdots \vee p_n), \ldots, (q_1 \vee \cdots \vee q_m)\}$.

Nota 5.1.1. Una forma clausal de $\neg(p \wedge (q \to r))$ es $\{\{\neg p, q\}, \{\neg p, \neg r\}\}$.

Ahora que ya conocemos los conceptos básicos, debemos comenzar la implementación.

Definimos los tipos de dato `Clausula` y `Clausulas` para la representación de una cláusula o un conjunto de ellas, respectivamente.

```
data Clausula  = C [Form]
data Clausulas = Cs [Clausula]
```

Definimos su representación en la clase `Show`

```
instance Show Clausula where
    show (C []) = "[]"
    show (C fs) = "{" ++  init (tail (show fs)) ++ "}"
instance Show Clausulas where
    show (Cs []) = "[]"
    show (Cs cs) = "{" ++  init (tail (show cs)) ++ "}"
```

Si consideramos la siguiente fórmula,

```
ghci> Neg (Conj [p,Impl q r])
¬(p⋀(q⟹r))
```

Su forma clausal sería la siguiente:

```
ghci> Cs [C [Neg p,q], C [Neg p, Neg r]]
¬p,q,¬p,¬r
```

Para el cálculo de la forma clausal tenemos el siguiente algoritmo:

1. Sea $F_1 = \exists y_1 \ldots \exists y_n F$, donde y_i con $i = 1, \ldots, n$ son las variables libres de F.

2. Sea F_2 una forma normal prenexa conjuntiva rectificada de F_1.

3. Sea $F_3 =$ `Skolem` (F_2), que tiene la forma

$$\forall x_1 \ldots \forall x_p[(L_1 \vee \cdots \vee L_n) \wedge \cdots \wedge (M_1 \vee \cdots \vee M_m)]$$

Entonces, una forma clausal es

$$S = \{\{L_1, \ldots, L_n\}, \ldots, \{M_1, \ldots, M_m\}\}$$

Dada una fórmula que está en la forma del paso 3 del algoritmo, es decir,

$$\texttt{f} = \forall x_1 \ldots \forall x_p[(L_1 \vee \cdots \vee L_n) \wedge \cdots \wedge (M_1 \vee \cdots \vee M_m)],$$

podemos convertirla a su forma causal por medio de la función `(form3AC f)`

```
form3CAC :: Form -> Clausulas
form3CAC (Disy fs) = Cs [C fs]
form3CAC p@(Atom _ _) = Cs [C [p]]
form3CAC (PTodo x f) = form3CAC f
form3CAC (Conj fs) = Cs (map disyAClau fs)
    where
      disyAClau p@(Atom _ _) = C [p]
      disyAClau p@(Neg (Atom _ _)) = C [p]
      disyAClau (Disy fs) = C fs
```

Por ejemplo,

```
-- | Ejemplo
-- >>> Conj [p, Disy [q,r]]
-- (p⋀(q⋁r))
-- >>> form3CAC (Conj [p, Disy [q,r]])
-- {{p},{q,r}}
```

Definimos `(formaClausal f)` que transforma una fórmula `f` a su forma clausal.

```
formaClausal :: Form -> Clausulas
formaClausal  = form3CAC . skolem .formaNPConjuntiva
```

Por ejemplo,

```
-- | Ejemplos
-- >>> formaClausal (Neg (Conj [p, Impl q r]))
-- {{¬p,q},{¬p,¬r}}
-- >>> formaClausal (Disy [PTodo x (Atom "P" [tx]),Ex y (Atom "Q" [ty])])
-- {{P[x0],Q[sk0[x0]]}}
-- >>> let f = Neg (PTodo x (Ex y (Neg (Equiv (Atom "P" [ty,tx]) (Neg (Atom "P" [ty,t
-- >>> formaClausal f
-- {{¬P[sk0[x0],x0],¬P[sk0[x0],sk0[x0]]},{P[sk0[x0],sk0[x0]],P[sk0[x0],x0]}}
```

Definimos la unión clausal mediante el operador infijo `(++!)`.

```
(++!) :: Clausulas -> Clausulas -> Clausulas
(Cs cs) ++! (Cs cs') = Cs (cs++cs')
```

```
-- | Ejemplo
-- >>> let c1 = formaClausal (Impl p q)
-- >>> let c2 = formaClausal (Impl q r)
-- >>> c1 ++! c2
-- {{¬p,q},{¬q,r}}
```

Definimos otro operador infijo que puede resultar útil al hacer resolución en un conjunto de cláusulas. `(!!!)` devuelve la cláusula n-ésima de un conjunto de cláusulas.

```
(!!!) :: Clausulas -> Int -> Clausula
(Cs cs) !!! n = cs !! n
```

Definimos la eliminación de un literal en una cláusula mediante `(borra l c)`.

```
borra :: Form -> Clausula -> Clausula
borra p (C fs) = C (delete p fs)
```

Definimos el operador infijo `(+!)` que une cláusulas.

```
(+!) :: Clausula -> Clausula -> Clausula
(C fs)+! (C gs) = C (nub (fs++gs))
```

Una serie de ejemplos de estas funciones definidas podrían ser.

```
-- | Ejemplos
-- >>> let c = C [Atom "P" [tx],q]
-- >>> c
-- {P[x],q}
-- >>> borra q c
-- {P[x]}
-- >>> let c' = C [Atom "P" [tx],q,Atom "Q" [Ter "f" [tx]]]
-- >>> c+!c'
-- {P[x],q,Q[f[x]]}
-- >>> let f = Neg (Impl (Conj [(PTodo x (Impl (Atom "P" [tx]) (Atom "Q" [tx]))),PTod
-- >>> f
-- ¬((∀x (P[x]⟹Q[x])⋀∀x (Q[x]⟹R[x]))⟹∀x (P[x]⟹R[x]))
-- >>> formaClausal f
-- {{¬P[x0],Q[x0]},{¬Q[x1],R[x1]},{P[x2]},{¬R[x2]}}
-- >>> (formaClausal f)!!! 3
-- {¬R[x2]}
```

5.2. Interpretación y modelos de la forma clausal

Primero implementemos un tipo de dato adecuado para las interpretaciones de cláusulas, `InterpretacionC`.

```
type InterpretacionC = [(Form,Int)]
```

Definición 5.2.1. El **valor** de una cláusula C en una interpretación I es

$$I(C) = \begin{cases} 1, & \text{si existe un } L \in C \text{ tal que } I(L) = 1, \\ 0, & \text{en caso contrario} \end{cases}$$

Implementamos el valor de una cláusula `c` por una interpretación `is` mediante la función `(valorC c is)`.

```
valorC :: Clausula -> InterpretacionC -> Int
valorC (C fs) is =
    if ([1 | (f,1) <- is, elem f fs] ++
        [1 | (f,0) <- is, elem (Neg f) fs]) /= []
    then 1 else 0
```

Definición 5.2.2. El **valor** de un conjunto de cláusulas S en una interpretación I es

$$I(S) = \begin{cases} 1, & \text{si para toda } C \in S, I(C) = 1, \\ 0, & \text{en caso contrario} \end{cases}$$

Implementamos el valor de un conjunto de cláusulas mediante la función `(valorCs cs is)`

```
valorCs :: Clausulas -> InterpretacionC -> Int
valorCs (Cs cs) is =
    if (all (==1) [valorC c is | c <- cs]) then 1 else 0
```

Nota 5.2.1. Cualquier interpretación de la cláusula vacía es 0.

Definición 5.2.3. Una cláusula C y una fórmula F son **equivalentes** si $I(C) = I(F)$ para cualquier interpretación I.

Veamos algunos ejemplos que nos ilustren lo definido hasta ahora:

```
-- | Ejemplos
-- >>> let c = Cs [C [Neg p,p],C [Neg p,Neg r]]
-- >>> c
-- {{¬p,p},{¬p,¬r}}
-- >>> valorCs c [(p,1),(r,0)]
-- 1
-- >>> valorCs c [(p,1),(r,1)]
-- 0
```

Definición 5.2.4. Una interpretación I es **modelo** de un conjunto de cláusulas S si $I(S) = 1$.

Caracterizamos el concepto "modelo de un conjunto de cláusulas" mediante la función `(is 'esModeloDe' cs)`.

```
esModeloDe :: InterpretacionC -> Clausulas -> Bool
esModeloDe is cs = valorCs cs is  == 1
```

```
-- | Ejemplos
-- >>> let c = Cs [C [Neg p,p],C [Neg p,Neg r]]
-- >>> let is = [(p,1),(r,0)]
-- >>> is `esModeloDe` c
-- True
```

Definición 5.2.5. Un conjunto de cláusulas es **consistente** si tiene modelos e **inconsistente**, en caso contrario.

Definamos una serie de funciones necesarias para determinar si un conjunto de cláusulas es consistente.

Primero definimos las funciones `(atomosC c)` y `(atomosCs cs)` que obtienen una lista de los átomos que aparecen en la cláusula o conjuntos de cláusulas `c` y `cs`, respectivamente.

```
atomosC :: Clausula -> [Form]
atomosC (C fs) = nub ([f | f <- fs, esAtomo f] ++ [f | (Neg f) <- fs])
    where
      esAtomo (Atom _ _) = True
      esAtomo _ = False

atomosCs :: Clausulas -> [Form]
atomosCs (Cs cs) = nub (concat [atomosC c | c <- cs])
```

A continuación, mediante la implementación de `(interPosibles cs)` obtenemos una lista de todas las posibles interpretaciones que podemos obtener de los átomos de `cs`.

```
interPosibles :: Clausulas -> [InterpretacionC]
interPosibles = sequence . aux2 . aux1 . atomosCs
    where
      aux1 fs = [(a,b) | a <- fs, b <- [0,1]]
      aux2 [] = []
      aux2 fs = [take 2 fs] ++ (aux2 (drop 2 fs))
```

Finalmente, comprobamos con la función `(esConsistente cs)` que alguna de las interpretaciones posibles es modelo del conjunto de cláusulas.

```
esConsistente :: Clausulas -> Bool
esConsistente cs = or [i `esModeloDe` cs | i <- is]
    where
      is = interPosibles cs
```

Ahora, como acostumbramos, veamos algunos ejemplos de las funciones definidas.

```
-- | Ejemplos
-- >>> let c = Cs [C [Neg p,p],C [Neg p,Neg r]]
-- >>> atomosCs c
-- [p,r]
-- >>> interPosibles c
-- [[(p,0),(r,0)],[(p,0),(r,1)],[(p,1),(r,0)],[(p,1),(r,1)]]
-- >>> esConsistente c
-- True
-- >>> let c' = Cs [C [p],C [Neg p,q],C [Neg q]]
-- >>> c'
-- {{p},{¬p,q},{¬q}}
-- >>> esConsistente c'
-- False
```

Definición 5.2.6. $S \models C$ si para todo modelo I de S, $I(C) = 1$.

Para su implementación en Haskell definimos la lista de las interpretaciones que son modelo de un conjunto de cláusulas mediante la función `(modelosDe cs)`

```
modelosDe :: Clausulas -> [InterpretacionC]
modelosDe cs = [i | i <- is, i `esModeloDe` cs]
    where
      is = interPosibles cs
```

Caracterizamos cuando una cláusula es consecuencia de un conjunto de cláusulas mediante la función `(c `esConsecuenciaDe` cs)`

```
esConsecuenciaDe :: Clausulas -> Clausulas ->  Bool
esConsecuenciaDe c cs = and [i `esModeloDe` c |i <-  modelosDe cs]
```

Veamos por ejemplo que si tenemos $p \to q$ y $q \to r$ se tiene como consecuencia que $p \to r$.

```
-- | Ejemplo
-- >>> let c1 = formaClausal (Impl p q)
-- >>> let c2 = formaClausal (Impl q r)
-- >>> let c3 = formaClausal (Impl p r)
-- >>> esConsecuenciaDe c3 (c1++!c2)
-- True
```

Proposición 5.2.1. Sean $S_1, \ldots, S_n$ formas clausales de las fórmulas $F_1, \ldots, F_n$:

- $\{F_1, \ldots, F_n\}$ es consistente si y sólo si $S_1 \cup \cdots \cup S_n$ es consistente.
- Si S es una forma clausal de $\neg G$, entonces son equivalentes:
  1. $\{F_1, \ldots, F_n\} \models G$.
  2. $\{F_1, \ldots, F_n, \neg G\}$ es inconsistente.
  3. $S_1 \cup \cdots \cup S_n \cup S$ es inconsistente.

Si continuamos con el ejemplo anterior, una aplicación de esta proposición sería ver que

$$\{p \to q, q \to r\} \models p \to r \Leftrightarrow \{\{\neg p, q\}, \{\neg q, r\}, \{p\}, \{\neg r\}\} \text{ es inconsistente.}$$

Hemos comprobado que lo primero es cierto, es decir, que se tiene la consecuencia. Nos faltaría comprobar que la expresión a la derecha del "si y sólo si" es inconsistente. Lo comprobamos a continuación.

```
-- | Ejemplo
-- >>> let c1 = formaClausal (Impl p q)
-- >>> let c2 = formaClausal (Impl q r)
-- >>> let c3 = Cs [C [p],C [Neg r]]
-- >>> c1++!c2++!c3
-- {{¬p,q},{¬q,r},{p},{¬r}}
-- >>> esConsistente (c1++!c2++!c3)
-- False
```

5.3. Resolución proposicional

Definición 5.3.1. Sean C_1 una cláusula, L un literal de C_1 y C_2 una cláusula que contiene el complementario de L. La **resolvente de** C_1 **y** C_2 **respecto de** L es

$$Res_L(C_1, C_2) = (C_1 \backslash \{L\}) \cup (C_2 \backslash \{L^c\})$$

Implementamos la función `(res c1 c2 l)` que calcula la resolvente de `c1` y `c2` respecto del literal `l`.

```
res :: Clausula -> Clausula -> Form -> Clausula
res (C fs) (C gs) l | p = C (nub (delete (Neg l) ((delete l (fs++gs)))))
                    | otherwise =
                        error "l no pertenece a alguna de las cláusulas"
```

```
                    where
                      p = ((elem l fs) && (elem (Neg l) gs)) ||
                          ((elem l gs) && (elem (Neg l) fs))
```

Nota 5.3.1. Consideraremos que `l` siempre será un átomo.

```
-- | Ejemplos
-- >>> res (C [p,q]) (C [Neg q,r]) q
-- {p,r}
-- >>> res (C [p]) (C [Neg p]) p
-- []
```

Definición 5.3.2. Sean C_1 y C_2 cláusulas, se define $Res(C_1, C_2)$ como el conjunto de las resolventes entre C_1 y C_2.

Se define la función `(ress c1 c2)` que calcula el conjunto de las resolventes de las cláusulas `c1` y `c2`.

```
ress :: Clausula -> Clausula -> [[Form]]
ress (C []) (C gs) = []
ress (C ((Neg f):fs)) (C gs) | elem f gs = [f,Neg f]:(ress (C fs) (C
                                                                gs))
                             | otherwise = ress (C fs) (C gs)
ress (C (f:fs)) (C gs) | elem (Neg f) gs = [f,Neg f]:(ress (C fs) (C
                                                                gs))
                       | otherwise = ress (C fs) (C gs)
```

Algunos ejemplos

```
-- | Ejemplos
-- >>> ress (C [p,q,Neg r]) (C [Neg q,r])
-- [[q,¬q],[r,¬r]]
-- >>> ress (C [p]) (C [Neg q,r])
-- []
```

5.3.1. Resolvente binaria

En esta sección implementaremos la resolución binaria entre dos cláusulas. Con este objetivo definimos inicialmente la función `(listaTerms f)` que calcula los términos de una fórmula dada.

Nota 5.3.2. La fórmula de entrada siempre será un literal, pues se aplicará a formas clausales.

```
listaTerms :: Form -> [Termino]
listaTerms (Atom _ ts) = ts
listaTerms (Neg (Atom _ ts)) = ts
```

Ahora calculamos la resolución entre dos cláusulas mediante la función `(resolucion c1 c2 f1 f2)`, donde `c1` y `c2` son cláusulas y, `f1` y `f2` serán fórmulas de `c1` y `c2`, respectivamente, tales que se podrá efectuar resolución entre ellas mediante la unificación adecuada.

```
resolucion :: Clausula -> Clausula -> Form -> Form -> Clausula
resolucion c1@(C fs) c2@(C gs) f1 f2 =  aux c1' c2'
    where
      sust = unificadoresListas (listaTerms f1) (listaTerms f2)
      c1' = C (sustitucionForms (head sust) fs)
      c2' = C (sustitucionForms (head sust) gs)
      aux (C ((Neg f):fs)) (C gs) | elem f gs = C (fs++(delete f gs))
                                  | otherwise = aux (C fs) (C gs)
      aux (C (f:fs)) (C gs) | elem (Neg f) gs = C (fs ++ (delete (Neg f) gs))
                            | otherwise = aux (C fs) (C gs)
```

```
-- | Ejemplos
-- >>> let c1 = C [Neg (Atom "P" [tx, Ter "f" [tx,ty]])]
-- >>> c1
-- {¬P[x,f[x,y]]}
-- >>> let c2 = C [Atom "P" [a,tz],Neg (Atom "Q" [tz,tu])]
-- >>> c2
-- {P[a,z],¬Q[z,u]}
-- >>> resolucion c1 c2 (Neg (Atom "P" [tx, Ter "f" [tx,ty]])) (Atom "P" [a,tz])
-- {¬Q[f[a,y],u]}
```

5.4. Resolución de primer orden

5.4.1. Separación de variables

A continuación definamos una serie de conceptos importantes para la resolución.

Definición 5.4.1. La sustitución $[x_1/t_1, \ldots, x_n/t_n]$ es un **renombramiento** si todos los t_i son variables.

Nota 5.4.1. Mediante un renombramiento se obtiene una cláusula equivalente a la que teníamos.

```
renombramiento :: Clausula -> Sust -> Clausula
renombramiento (C fs) sust = C [sustitucionForm sust f | f <- fs]
```

Definición 5.4.2. Las cláusulas C_1 y C_2 **están separadas** si no tienen ninguna variable común.

La función `(varsClaus c)` obtiene la lista de las variables que aparecen en la cláusula `c`.

```
varsClaus :: Clausula -> [Variable]
varsClaus (C fs) = concat [varEnForm f | f <- fs]
```

Definición 5.4.3. Una **separación de las variables** de C_1 y C_2 es un par de renombramientos (θ_1, θ_2) tales que $C_1\theta_1$ y $C_2\theta_2$ están separadas.

Vayamos definiendo funciones de manera progresiva para el cálculo de la resolvente binaria de dos cláusulas.

Definimos `(mismoNombre l1 l2)` que determina si dos literales son iguales en nombre aunque no tengan las mismos términos. Nos interesan aquellos que sean negacione suno del otro, por ello sólo tenemos en cuenta dos casos.

```
mismoNombre (Neg (Atom n1 _)) (Atom n2 _) = n1 == n2
mismoNombre (Atom n1 _) (Neg (Atom n2 _)) = n1 == n2
mismoNombre _ _ = False
```

Un par de ejemplos que ilustran la función.

```
-- | Ejemplo
-- >>> mismoNombre (Atom "P" [tx]) (Atom "P" [Ter "f" [tx]])
-- False
-- >>> mismoNombre (Atom "P" [tx]) (Neg (Atom "P" [Ter "f" [tx]]))
-- True
```

Definimos `(RenAux n str vs)` que dada una lista de variables `vs` obtiene una sustitución de las variables nombrándolas según un nombre `str` y una secuencia de números empezando en `n`.

```
renAux :: Int -> String -> [Variable] -> Sust
renAux _ _ [] = []
renAux n str (v:vs) = (v,Var (Variable str [n])): (renAux (n+1) str vs)
```

Definimos `(separacionVars c1 c2)` que separa las variables de ambas cláusulas, devolviendo un par con las cláusulas ya separadas.

```
separacionVars :: Clausula -> Clausula -> (Clausula,Clausula)
separacionVars c1 c2 = (renombramiento c1 s1, renombramiento c2 s2)
    where
      s1 = renAux 1 "x" (varsClaus c1)
      s2 = renAux 1 "y" (varsClaus c2)
```

En el siguiente ejemplo vemos la separación de variables de las cláusulas

$$C_1 = \{P(x), Q(x,y)\} \text{ y } \{\neg Q(x), R(g(x))\}$$

```
-- | Ejemplos
-- >>> let c1 = C [Neg (Atom "P" [tx]), Atom "Q" [Ter "f" [tx]]]
-- >>> let c2 = C [Neg (Atom "Q" [tx]), Atom "R" [Ter "g" [tx]]]
-- >>> separacionVars c1 c2
-- ({¬P[x1],Q[f[x1]]},{¬Q[y1],R[g[y1]]})
```

5.4.2. Resolvente binaria

En esta sección definiremos la resolvente binaria en lógica de primer orden. Para ello definiremos una serie de funciones auxiliares hasta alcanzar la meta de la resolvente de dos cláusulas.

Definimos `(litMisNom c1 c2)` que determina los literales comunes en ambas cláusulas.

```
litMisNom :: Clausula -> Clausula -> (Form, Form)
litMisNom (C fs) (C gs) = head [(f,g) | f <- fs,g <- gs, mismoNombre f g]
```

Definimos `(unifClau l1 l2)` que determina las unificaciones posibles entre dos literales.

```
unifClau (Atom _ ts) (Atom _ ts') = unificadoresListas ts ts'
unifClau (Atom _ ts) (Neg (Atom _ ts')) = unificadoresListas ts ts'
unifClau (Neg (Atom _ ts)) (Atom _ ts') = unificadoresListas ts ts'
```

Definición 5.4.4. La cláusula C es una **resolvente binaria** de las cláusulas C_1 y C_2 si existen una separación de variables (θ_1, θ_2) de C_1 y C_2, un literal $L_1 \in C_1$, un literal $L_2 \in C_2$ y un UMG σ de $L_1\sigma_1$ y $L_2^c\theta_2$ tales que

$$C = (C_1\theta_1\sigma \backslash \{L_1\theta_2\sigma_1\}) \cup (C_2\theta_2\sigma \backslash \{L_2\theta_2\sigma\})$$

Definimos la resolvente binaria de dos cláusulas mediante la función `(resolBin c1 c2)`.

```
resolBin c1 c2 | s /= [] = renombramiento (c1'' +! c2'') (head s)
               | otherwise = error "No se puede unificar"
    where
      (c1', c2') = separacionVars c1 c2
      (p,q) = (litMisNom c1' c2')
      s = unifClau p q
      c1'' = borra p c1'
      c2'' = borra q c2'
```

Empleando las cláusulas de antes, tenemos la siguiente resolvente binaria.

```
-- | Ejemplo
-- >>> let c1 = C [Neg (Atom "P" [tx]), Atom "Q" [Ter "f" [tx]]]
-- >>> let c2 = C [Neg (Atom "Q" [tx]), Atom "R" [Ter "g" [tx]]]
-- >>> resolBin c1 c2
-- {¬P[x1],R[g[f[x1]]]}
```

Capítulo 6

Correspondencia de Curry-Howard

A lo largo de este trabajo hemos implementado la lógica de primer orden en Haskell, así como distintos métodos y aspectos de la misma. Hemos traducido la lógica a un sistema de programación funcional, al lambda-cálculo. Mediante esta implementación que hemos desarrollado, sin tener una conciencia plena de ello, hemos establecido una relación entre dos campos, la lógica y un determinado lenguaje de programación. ¿Las matemáticas y la programación son lo mismo? ¿Hasta donde alcanza su similitud? ¿Todo lo que yo cree en matemáticas, proposiciones, teoremas, demostraciones... es implementable y tiene su hermano en el campo de la programación? La correspondencia de Curry Howard nos arrojará un poco de luz al respecto.

La correspondencia de Curry Howard, también llamada isomorfismo de Curry-Howard fue publicada en 1980 en honor a Curry. El lema principal que reivindica el isomorfismo es: **las proposiciones son tipos**. Es decir, se describe una correspondencia que asocia a cada proposición en la lógica un tipo en un lenguaje de programación, así como el recíproco. Se habla ampliamente sobre esta correspondencia en el siguiente artículo [15], así como en [12].

Una vez que se define dicha correspondencia nos toca profundizar. Cuando enunciamos una proposición se requiere su demostración. ¿Cuál es el hermano de las demostraciones que habita en la programación? Esta pregunta nos lleva al segundo lema: **las demostraciones son programas**. Es decir, dada una prueba de una proposición existe un programa del tipo correspondiente.

En 1934, Curry observó que toda función del tipo (`A` $\rightarrow$ `B`) puede ser leída como la proposición $(A \supset B)$ y, que leyendo el tipo de cualquier función siempre existe una proposición demostrable asociada. Howard, notando que existe una correspondencia similar entre deducción natural y el lambda-cálculo, probó la extensión de la correspondencia al resto de conectivas lógicas.

- La **conjunción** $(A \wedge B)$ corresponde al producto cartesiano. En Haskell dicho tipo corresponde al par (a, b).
- La **disyunción** $(A \vee B)$ corresponde a la suma disjunta de A y B. En Haskell tenemos su tipo de dato abstracto hermano equivalente que es `Either a b`.
- La **implicación** $(A \Rightarrow B)$ corresponde a las funciones de tipo $A \to B$. Su hermano en Haskell es evidente, se trata de los tipos de función que van de un tipo de dato A a otro B.
- **Los cuantificadores existencial** $(\exists)$ y **universal** $(\forall)$ corresponden a los llamados tipos dependientes.

De hecho, la correspondencia va más allá. Las reglas de demostración en la deducción natural y las reglas de tipos tienen sus similitudes. La primera se basa en reglas de introducción y reglas de eliminación. Las reglas de introducción se ven representadas en la programación en las maneras de definir o construir valores de un determinado tipo, así como las reglas de eliminación son representadas por las formas de usar o deconstruir valores de un determinado tipo.

Un nivel más profundo de identificación entre la programación y la lógica que el isomorfismo de Curry establece es que **las simplificaciones de demostraciones son evaluaciones de programas**. Por lo tanto, a pesar de no establecer una biyección entre ambos ámbitos, se establece un isomorfismo que preserva la estructura entre programas y pruebas, simplificación y evaluación.

Para terminar veamos algunos ejemplos entre la lógica y la programación en Haskell, se pueden ver más ejemplo, así como análisis sobre la correspondencia en [4], [6] y [3] . Para ello, creemos un módulo que contenga nuestro ejemplos.

```
module CHC where
import Data.Either
```

Vayamos escalando la montaña con mesura. Si tenemos la implicación $a \to a$, no es dificil intuir la función que la demuestra y el tipo que define la proposición en Haskell, no es más que la función `identidad`.

```
identidad :: a -> a
identidad x = x
```

Sigamos componiendo conectivas para ir construyendo ejemplos poco a poco más complejos. La implicación $a \to b \to a \wedge b$ se vería representada en Haskell por el tipo

```
a -> b -> (a, b)
```

Y demostrada por la existencia de una función con ese tipo, en este caso el operador `(,)`.

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

Otro ejemplo podría ser la regla de la eliminación de la conjunción por la izquierda, es decir, $a \wedge b \rightarrow a$ cuya representación y demostración ponemos a continuación.

```
fst :: (a,b) -> a
fst (x,y) = x
```

Hasta ahora hemos demostrado distintas proposiciones encontrando la función adecuada pero, ¿qué sucede si tomamos una que no sea cierta? Por ejemplo, $a \rightarrow a \wedge b$, ¿existe alguna función con el tipo de dato `a ->(a,b)`? La respuesta es no, un alivio porque en caso contrario no sería consistente la teoría.

Procedamos ahora con la siguiente proposición $(a \rightarrow b \rightarrow c) \rightarrow (b \rightarrow a \rightarrow c)$ siendo su tipo y función demostradora los siguientes.

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

¿ Existirá alguna función con el tipo análogo a la proposición $a \wedge b \rightarrow a$? El tipo de la función buscada sería.

```
Either a b -> a
```

Y podemos pasarlo por nuestro demostrador basado en tableros semánticos y determinar si debería existit o no una función con este tipo.

```
ghci> esTeorema 5 (Impl (Disy [p,q]) q)
False
```

Por lo tanto, no debería y, de hecho, no existe ninguna función con el tipo buscado.

Anteriormente no hemos hablado anteriormente del análogo de la negación lógica en Haskell. Para ello, primero tenemos que añadir las siguientes líneas y así extender el lenguaje.

```
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE ImpredicativeTypes #-}
{-# LANGUAGE ExistentialQuantification #-}
```

Posteriormente definimos el tipo `Not` como que para cualquier `a` se puede inferir cualquier cosa.

```
type Not x = (forall a. x -> a)
```

Ahora podemos definir una serie de ejemplos que nos convenzan de la correspondencia.

Nuestro primer ejemplo consistirá en las leyes de deMorgan, cuya formalización matemática es:

$$\neg(A \wedge B) \leftrightarrow (\neg A) \vee (\neg B) \text{ y } \neg(A \vee B) \leftrightarrow (\neg A) \vee (\neg B)$$

A estas proposiciones les corresponde un tipo de dato en Haskell.

Nota 6.0.1. Nosotros nos limitaremos a una de las implicaciones.

```
leydeMorgan1 :: (Not a, Not b) -> Not (Either a b)

leydeMorgan2 :: Either (Not a) (Not b) -> Not (a,b)
```

Y podríamos demostrar dichas proposiciones pero eso será equivalente a la existencia de un programa asociado a los tipos de datos antes señalados.

```
leydeMorgan1 (noA, _) (Left a) = noA a
leydeMorgan1 (_, noB) (Right b) = noB b

leydeMorgan2 (Left noA) (a, _) = noA a
leydeMorgan2 (Right noB) (_, b) = noB b
```

Veamos un último ejemplo con motivación inversa. Si tenemos la función implementada en Haskell `curry f a b`, intentemos deducir su análogo lógico. Veamos primero su uso con unos ejemplos:

```
-- | Ejemplos
-- >>> curry fst 2 3
-- 2
-- >>> curry fst 'a' 'b'
-- 'a'
-- >>> curry snd 'a' 'b'
-- 'b'
```

Nota 6.0.2. `curry` 'currifica' una función, es decir, dada una función `f` y dos elementos `a` y `b` se da la igualdad informal `curry (f,a,b) = f(a,b)`.

Una vez conocido su uso, preguntemos a Haskell cuál es su tipo:

```
ghci> :t curry
curry :: ((a, b) -> c) -> a -> b -> c
```

Vista la respuesta de Haskell no es dificil inferir la proposición asociada,

$$((A \wedge B) \to C) \to (A \to B \to C)$$

Por lo tanto, hemos establecido de nuevo una correspondencia entre lógica y, en este caso, el lenguaje de programación funcional Haskell.

Apéndice A

Trabajando con GitHub

En este apéndice se pretende introducir al lector en el empleo de GitHub, sistema remoto de versiones, así como el uso de `magit` en emacs.

A.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.

A.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`

A.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>"
```

A.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.

A.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)`

Apéndice B

Usando Doctest

En este apéndice se introducirá el uso del paquete doctest. Fue implementado basado en el paquete doctest para Python.

Cuando se realizan trabajos de programación de gran extensión en el que muchos programas dependen unos de otros, el trabajo de documentación y comprobación puede volverse un caos. Un sistema para llevar a cabo estas penosas tareas es empleando el paquete doctest. Su documentación se encuentra en [5], así como una guía más amplia de la que aquí se expone.

El paquete `doctest` se puede encontrar en Hackage, y por lo tanto, se puede instalar mediante `cabal install doctest`.

En cuanto a su uso, los ejemplos deben ser estructurados tal y como se ha realizado en todo el trabajo. Por ejemplo, en resolución lo hicimos de la siguiente manera:

```
-- | Ejemplos
-- >>> let c = Cs [C [Neg p,p],C [Neg p,Neg r]]
-- >>> atomosCs c
-- [p,r]
-- >>> interPosibles c
-- [[(p,0),(r,0)],[(p,0),(r,1)],[(p,1),(r,0)],[(p,1),(r,1)]]
-- >>> esConsistente c
-- True
-- >>> let c' = Cs [C [p],C [Neg p,q],C [Neg q]]
-- >>> c'
-- {{p},{¬p,q},{¬q}}
-- >>> esConsistente c'
-- False
```

Finalmente, se abre la carpeta en la que está el archivo en un terminal y se invoca el paquete mediante `doctest RES.lhs`, y en caso de que todos los ejemplos sean correctos, devolverá:

```
   >doctest RES.lhs
Examples: 163 Tried: 163 Errors: 0 Failures: 0
```

En el caso de que hubiera un error devuelve dónde lo ha encontrado, así como lo que él esperaba encontrar.

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] V. autores. Curry-howard-lambek correspondence. Technical report, Haskell Wiki, 2010. En https://wiki.haskell.org/Curry-Howard-Lambek_correspondence.

[4] V. autores. Haskell/the curry–howard isomorphism. Technical report, wikibooks, 2016. En https://en.wikibooks.org/wiki/Haskell/The_Curry-Howard_isomorphism.

[5] V. autores. Doctest: Test interactive haskell. Technical report, github, 2017. En https://github.com/sol/doctest#readme.

[6] G. Gonzalez. The curry-howard correspondence between programs and proofs. Technical report, Haskell for all, 2017. En http://www.haskellforall.com/2017/02/the-curry-howard-correspondence-between.html.

[7] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2016.

[8] G. Jovanovski. Haskell binary tree manipulation. Technical report, github, 2015. En https://gist.github.com/jovanovski/0ce88e66dbed59099dbc.

[9] J. H. K. Claessen. Quickcheck: A lightweight tool for random testing of haskell programs. Technical report, Chalmers University of Technology, 2009. En http://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf.

[10] S. Marlow. Haskell 2010 language report. Technical report, haskell.org, 2010. En https://www.haskell.org/onlinereport/haskell2010.

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

[12] M. Román. Curry-howard. Technical report, Universidad de Granada, 2014. En https://github.com/libreim/curryHoward/blob/master/CurryHoward.pdf.

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

[14] J. P. Villa. A quickcheck tutorial: Generators. Technical report, stackbuilders.com, 2015. En https://www.stackbuilders.com/es/news/a-quickcheck-tutorial-generators.

[15] P. Wadler. Propositions as types. Technical report, University of Edinburgh, 2014. En http://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf.

Índice alfabético

Lista de tareas pendientes