DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Grupo *Pollo al uncurry*

23 de Abril de 2015 — Paradigmas De Programación

Integrante	LU	Correo electrónico
Benitti, Raul	592/08	raulbenitti@gmail.com
Giordano, Francisco	429/13	frangio.1@gmail.com
Vallejo, Nicolás Agustín	500/10	nico_pr08@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Tipos

```
module Tipos where

import Grafo

type Mundo = Integer
type Prop = String
data Modelo = K (Grafo Mundo) (Prop -> [Mundo])

data Exp = Var Prop | Not Exp | Or Exp Exp | And Exp Exp | D Exp | B Exp deriving (Show, Eq)
```

2. Grafo

```
module Grafo
    (Grafo, vacio, nodos, vecinos, agNodo, sacarNodo, agEje, lineal, union, clausura) where

import qualified Data.List as List (delete, union, nub)

-- Invariante: ninguna de las listas (nodos y vecinos de nodos) tienen repetidos
data Grafo a = G [a] (a -> [a])

instance (Show a) => Show (Grafo a) where
  show (G n e) =
    "[\n" ++ concat (map (\x -> " " ++ show x ++ " -> " ++ show (e x) ++ "\n") n) ++ "]"


-- Ejercicio 1
-- Crea un grafo sin nodos.

vacio :: Grafo a
vacio = G [] (\_ -> [])

-- Ejercicio 2
-- Devuelve los nodos de un grafo.

nodos :: Grafo a -> [a]
nodos (G ns fv) = ns

-- Ejercicio 3
-- Devuelve los vecinos de un nodo en un grafo dado.

vecinos :: Grafo a -> a -> [a]
vecinos (G ns fv) = fv

-- Ejercicio 4
-- Devuelve un grafo con un nodo agregado, sin vecinos.
-- Si el nodo ya está presente en el grafo, no tendrá vecinos en nuevo grafo.

agNodo :: Eq a => a -> Grafo a -> Grafo a
agNodo n (G ns fv) = G ns' fv'
        where ns' = List.union [n] ns
              fv' e | e == n    = []
                    | otherwise = fv e


-- Ejercicio 5
-- Devuelve un grafo sin un nodo dado.

sacarNodo :: Eq a => a -> Grafo a -> Grafo a
sacarNodo n (G ns fv) = G ns' fv'
        where ns' = List.delete n ns
              fv' e | e == n    = []
                    | otherwise = List.delete n $ fv e
```

```
-- Ejercicio 6
-- Devuelve un grafo con un eje agregado.

agEje :: Eq a =>  (a,a) -> Grafo a -> Grafo a
agEje (n1, n2) (G ns fv) = G ns fv'
        where fv' e | e == n1   = n2 : fv e
                    | otherwise = fv e


-- Ejercicio 7
-- Devuelve un grafo donde los nodos son todos los de la lista pasada por
-- argumento, y cada nodo tiene como unico vecino al elemento que lo sigue en
-- dicha lista.

lineal :: Eq a => [a] -> Grafo a
lineal ns = G ns fv
        where fv e = take 1 $ drop 1 $ dropWhile (/= e) ns


-- Ejercicio 8
-- Devuelve la unión de dos grafos.

union :: Eq a => Grafo a -> Grafo a -> Grafo a
union (G ns1 fv1) (G ns2 fv2) = G ns' fv'
        where ns' = List.union ns1 ns2
              fv' e = List.union (fv1 e) (fv2 e)

-- Ejercicio 9
-- Devuelve la clausura reflexo transitiva de un grafo.

clausura :: (Eq a) => Grafo a -> Grafo a
clausura (G ns fv) = G ns fv'
        where fv' e = puntofijo extenderConVecinos [e]
              extenderConVecinos xs = foldl List.union xs (map fv xs)

-- Punto fijo
-- Devuelve f(f(...f(x)...)) = f^n(x), tal que f(f^n(x)) = f^n(x), es decir que
-- es un punto fijo de la función.

puntofijo :: (Eq a) => (a -> a) -> (a -> a)
puntofijo f x = head $ dropWhile (\e -> (f e) /= e) $ iterate f x
```

3. Lomoba

```
module Lomoba where
import Grafo
import Tipos
import qualified Data.List as List (union, intersect)


-- Ejercicio 10
-- Realiza recursión estructural sobre expresiónes.

foldExp ::
  (Prop -> b) -> (b -> b) -> (b -> b -> b) -> (b -> b -> b) -> (b -> b) -> (b -> b) -> Exp -> b
foldExp fVar fNot fOr fAnd fD fB ei =
        let frec = foldExp fVar fNot fOr fAnd fD fB
        in case ei of
                Var p -> fVar p
                Not e -> fNot (frec e)
                Or e1 e2 -> fOr (frec e1) (frec e2)
                And e3 e4 -> fAnd (frec e3) (frec e4)
                D e5 -> fD (frec e5)
                B e6 -> fB (frec e6)


-- Ejercicio 11
-- Devuelve un la visibilidad de una fórmula, es decir cuánto del grafo se
-- utiliza para evaluarla.

visibilidad :: Exp -> Integer
visibilidad = foldExp fVar fNot fOr fAnd fD fB
        where   fVar = const 0
                fNot = id
                fOr = max
                fAnd = max
                fD = (1+)
                fB = (1+)


-- Ejercicio 12
-- Lista las variables que aparecen en la fórmula.

extraer :: Exp -> [Prop]
extraer = foldExp fVar fNot fOr fAnd fD fB
        where   fVar p = [p]
                fNot = id
                fOr = List.union
                fAnd = List.union
                fD = id
                fB = id

-- Ejercicio 13
-- Dado un modelo, decide si una fórmula es verdadera en un mundo.

eval :: Modelo -> Mundo -> Exp -> Bool
eval m w e = eval' m e w
```

```
eval' :: Modelo -> Exp -> Mundo -> Bool
eval' m@(K g v) = foldExp fVar fNot fOr fAnd fD fB
        where        fVar x w = elem w (v x)
                fNot rec w = not (rec w)
                fOr recI recD w = (recI w) || (recD w)
                fAnd recI recD w = (recI w) && (recD w)
                fD rec w = any rec (vecinos g w)
                fB rec w = all rec (vecinos g w)


-- Ejercicio 14
-- Devuelve todos los mundos de un modelo para los que vale una expresión.

valeEn :: Exp -> Modelo -> [Mundo]
valeEn e m@(K g v) = filter (eval' m e) (nodos g)


-- Ejercicio 15
-- Devuelve un modelo en el que para todos los mundos vale la expresión dada.

quitar :: Exp -> Modelo -> Modelo
quitar e m@(K g v) = K g' v'
        where   g' = foldl (flip sacarNodo) g (noValeEn e m)
                v' p = List.intersect (nodos g') (v p)

noValeEn :: Exp -> Modelo ->[Mundo]
noValeEn e m@(K g v) = filter (not . eval' m e) (nodos g)

-- Ejercicio 16
-- Dado un modelo, indica si para todos sus mundos vale una expresión.

cierto :: Modelo -> Exp -> Bool
cierto m@(K g v) e = all (eval' m e) (nodos g)
```

4. Tests

```
import Grafo
import Tipos
import Lomoba
import Parser
import Test.HUnit

-- evaluar t para correr todos los tests
t = runTestTT allTests

allTests = test [
"parser" ~: testsParser,
"grafo" ~: testsGrafo,
"lomoba" ~: testsLomoba
]

testsParser = test [
(Var "p")  ~=? (parse "p"),
(And (Var "p") (Var "q"))  ~=? (parse "p && q"),
(Or (Var "p") (Var "q"))  ~=? (parse "p || q"),
(Or (Not (Var "p")) (Var "q")) ~=? (parse "!p || q"),
(And (D (Var "p")) (Var "q"))  ~=? (parse "<>p && q"),
(And (B (Var "p")) (Var "q"))  ~=? (parse "[]p && q"),
(D (And (Var "p") (Var "q")))  ~=? (parse "<>(p && q)"),
(B (And (Var "p") (Var "q")))  ~=? (parse "[](p && q)")]

testsGrafo = test [
--nodos y agNodo
[1] ~~? (nodos (agNodo 1 vacio)),
[1,2] ~~? (nodos (agNodo 2 (agNodo 1 vacio))),

--vecinos y egEje
[1] ~~? (vecinos (agEje (2,1) (agNodo 2 (agNodo 1 vacio))) 2 ),
[] ~~? (vecinos (agEje (2,1) (agNodo 2 (agNodo 1 vacio))) 1 ),
[1,2,3] ~~? (nodos((agEje(3,2) (agEje (1,3) (agEje (1,2)(agNodo 3 (agNodo 2 (agNodo 1 vacio)))))))),
[2,3] ~~? (vecinos (agEje(3,2) (agEje (1,3) (agEje (1,2)(agNodo 3 (agNodo 2 (agNodo 1 vacio)))))) 1)
[2] ~~? (vecinos (agEje(3,2) (agEje (1,3) (agEje (1,2)(agNodo 3 (agNodo 2 (agNodo 1 vacio)))))) 3),
[] ~~? (vecinos (agEje(3,2) (agEje (1,3) (agEje (1,2)(agNodo 3 (agNodo 2 (agNodo 1 vacio)))))) 2),

--sacarNodo
[2] ~~? (nodos(sacarNodo 1 ((agNodo 2 (agNodo 1 vacio))))),
[] ~~? (vecinos (sacarNodo 2 (agEje(3,2) (agEje (1,3) (agEje (1,2)(agNodo 3 (agNodo 2 (agNodo 1 vaci
[3] ~~? (vecinos (sacarNodo 2 (agEje(3,2) (agEje (1,3) (agEje (1,2)(agNodo 3 (agNodo 2 (agNodo 1 vac
[1,3] ~~? (nodos (sacarNodo 2 (agEje(3,2) (agEje (1,3) (agEje (1,2)(agNodo 3 (agNodo 2 (agNodo 1 vac

--lineal
[1,2,3,4] ~~? (nodos (lineal [1,2,3,4])),
[2] ~~? (vecinos (lineal [1,2,3,4]) 1),
[3] ~~? (vecinos (lineal [1,2,3,4]) 2),
[4] ~~? (vecinos (lineal [1,2,3,4]) 3),
[] ~~? (vecinos (lineal [1,2,3,4]) 4),

--union
[1,2,3] ~~? (nodos (union (agEje (1,2) (agNodo 2 (agNodo 1 vacio))) (agEje (3,1) (agEje (1,3) (agNod
[2,3] ~~? (vecinos (union (agEje (1,2) (agNodo 2 (agNodo 1 vacio))) (agEje (3,1) (agEje (1,3) (agNod
[] ~~? (vecinos (union (agEje (1,2) (agNodo 2 (agNodo 1 vacio))) (agEje (3,1) (agEje (1,3) (agNodo 3
[1] ~~? (vecinos (union (agEje (1,2) (agNodo 2 (agNodo 1 vacio))) (agEje (3,1) (agEje (1,3) (agNodo
```

```
[1,2,3,4] ~~? (nodos (union (vacio) (lineal [1,2,3,4]))),

--clasura
[1,2,3,4] ~~? (nodos (clausura (lineal [1,2,3,4]))),
[1,2,3,4] ~~? (vecinos (clausura (lineal [1,2,3,4])) 1),
[2,3,4] ~~? (vecinos (clausura (lineal [1,2,3,4])) 2),
[3,4] ~~? (vecinos (clausura (lineal [1,2,3,4])) 3),
[4] ~~?(vecinos (clausura (lineal [1,2,3,4])) 4)
]

testsLomoba = test [
-- foldExp
        exp1 ~=? (foldExp Var Not Or And D B exp1),

-- visibilidad
0 ~=? (visibilidad (parse "p")),
1 ~=? (visibilidad (parse "<>p")),
1 ~=? (visibilidad (parse "!<>p")),
2 ~=? (visibilidad (parse "<>!<>p")),
2 ~=? (visibilidad (parse "<><>p || <><>q")),
3 ~=? (visibilidad (parse "<>(<>p || <><>q)")),
3 ~=? (visibilidad (parse "[](<>p && <>[]q)")),

-- extraer
["p"] ~~? (extraer (parse "p")),
["p"] ~~? (extraer (parse "<>p")),
["p"] ~~? (extraer (parse "[]p")),
["p", "q"] ~~? (extraer (parse "p||q")),
["p", "q"] ~~? (extraer (parse "p&&q")),
["p"] ~~? (extraer (parse "<><>p || <><>p")),
["p", "q", "r"] ~~? (extraer (parse "(p||q)&&[]<>r")),
["p", "q", "r"] ~~? (extraer (parse "<>((p||q)&&[]<>r)")),

-- eval
True ~=? eval modelo1 1 (parse "p"),
False ~=? eval modelo1 1 (parse "q"),

True ~=? eval modelo1 4 (parse "r && q"),
False ~=? eval modelo1 4 (parse "p && q"),

True ~=? eval modelo1 1 (parse "p || q"),
False ~=? eval modelo1 2 (parse "p || q"),

True ~=? eval modelo1 2 (parse "!p || q"),
False ~=? eval modelo1 1 (parse "!p || q"),
True ~=? eval modelo1 3 (parse "!p || q"),

False ~=? eval modelo1 1 (parse "<>p && p"),
True ~=? eval modelo1 1 (parse "<>q && p"),

True ~=? eval modelo1 1 (parse "[]r"),
False ~=? eval modelo1 1 (parse "[]q"),

False ~=? eval modelo1 1 (parse "<>(p && q)"),
True ~=? eval modelo1 1 (parse "<>(r && q)"),

True ~=? eval modelo1 2 (parse "[](!p && q)"),
```

```
-- valeEn
[1] ~~? valeEn (parse "p") modelo1,
[2,3,4] ~~? valeEn (parse "!p") modelo1,
[3,4] ~~? valeEn (parse "q") modelo1,
[1,2] ~~? valeEn (parse "<>q") modelo1,
[2,3,4] ~~? valeEn (parse "[]q") modelo1,
[1,3,4] ~~? valeEn (parse "[]r") modelo1,
[4] ~~? valeEn (parse "r && q") modelo1,
[1,2,3,4] ~~? valeEn (parse "r || (q || p)") modelo1,
[1,3] ~~? valeEn (parse "!r && (q || p)") modelo1,
[] ~~? valeEn (parse "s") modelo1,


--quitar
[1] ~~? let (K g _) = quitar (parse "p") modelo1 in nodos g,
[4,2] ~~? let (K g _) = quitar (parse "r") modelo1 in nodos g,
[4] ~~? let (K g _) = quitar (parse "r&&q") modelo1 in nodos g,
[1,3,4] ~~? let (K g _) = quitar (parse "[]r") modelo1 in nodos g,
[1] ~~? let (K g _) = quitar (parse "<>r") modelo1 in nodos g,

--cierto
True ~=? cierto (K vacio (const [])) (parse "p"),
True ~=? cierto (K vacio (const [])) (parse "[]p"),
False ~=? cierto modelo1 (parse "p"),
True ~=? cierto modelo1 (parse "[][]q")
]

modelo1 = K (union (lineal [1,2,3]) (lineal [1,4]) ) v1
where v1 "p" = [1]
v1 "q" = [3,4]
v1 "r" = [2,4]
v1 _ = []

exp1 = parse "<>[]p || q && r"

---------------
--  helpers  --
---------------

-- idem ~=? pero sin importar el orden
(~~?) :: (Ord a, Eq a, Show a) => [a] -> [a] -> Test
expected ~~? actual = (sort expected) ~=? (sort actual)
where
sort = foldl (\r e -> push r e) []
push r e = (filter (e<=) r) ++ [e] ++ (filter (e>) r)
```