Recuperatorio de Trabajo Práctico de Programación Funcional

Jueves 9 de Noviembre de 2017 — Paradigmas de Lenguajes de Programación

Integrante	LU	Correo electrónico
Francisco Demartino	348/14	demartino.francisco@gmail.com
Martín Mongi Badía	422/13	martinmongi@gmail.com
Grupo: "Demo"		

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

1. Main.hs

```
import Diccionario
import Data.Maybe
import Data.List
import Arbol23
import Test.HUnit

--Este módulo sirve para utilizar el diccionario sin acceder a su
--estructura interna. Pueden agregar otras funciones o casos de prueba.

{- Función a implementar. -}

búsquedaDelTesoro :: Eq a => a -> (a->Bool) -> Diccionario a a -> Maybe a
búsquedaDelTesoro p f d = until nadaOTesoro buscarProximaPista (Just p)
  where
    nadaOTesoro = maybe True f
    buscarProximaPista = maybe Nothing (flip obtener $ d)

--Ejecución de los tests
main :: IO Counts
main = do runTestTT allTests

allTests = test [
  "ejercicio2" ~: testsEj2,
  "ejercicio3" ~: testsEj3,
  "ejercicio4" ~: testsEj4,
  "ejercicio5" ~: testsEj5,
  "ejercicio6" ~: testsEj6,
  "ejercicio7" ~: testsEj7,
  "ejercicio8" ~: testsEj8,
  "ejercicio9" ~: testsEj9,
  "ejercicio10" ~: testsEj10
  ]

testsEj2 = test [
  [0,1,2,3,4,5,6,7] ~=? internos arbolito1,
  "abcdefghi" ~=? hojas arbolito1,
  [True,False,True] ~=? internos arbolito2,
  [1,2,3,2,3,4,3,4,5,4] ~=? take 10 (hojas arbolito3)
  ]

testsEj3 = test [
  [0,1,-1,5] ~=? hojas (incrementarHojas arbolito2)
  ]

testsEj4 = test [
  [1,2,3,2,3,4,3,4,5,4,5,6,0,0,0,0,0] ~=? hojas (truncar 0 6 arbolito3),
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0] ~=? hojas (truncar 0 3 arbolito4)
```

```
  ]

testsEj5 = test [
  22 ~=? evaluar (truncar 0 6 arbolito3)
  ]

diccVacio::Diccionario Int String
diccVacio = vacio (<)

testsEj6 = test [
  True ~=? isNothing (estructura diccVacio),
  [] ~=? claves diccVacio,
  Nothing ~=? obtener 42 diccVacio
  ]

testsEj7 = test [
  isNothing (estructura (definir 1 2 (vacio (\x-> \y-> False)))) ~=? False,
  isJust (estructura (definir 1 2 (vacio (\x-> \y-> False)))) ~=? True,
  "perro" ~=? fromJust (obtener 1 (definir 1 "perro" diccVacio))
  ]

testsEj8 = test [
  "Hola" ~=? fromJust (obtener 0 dicc1),
  "Chau" ~=? fromJust (obtener (-10) dicc1),
  "Felicidades" ~=? fromJust (obtener 15 dicc1),
  True ~=? isNothing (obtener 27 dicc1)
  ]

testsEj9 = test [
  [-10,0,2,9,15] ~=? sort(claves dicc1)
  ]

testsEj10 = test [
  Just "alfajor" ~=? búsquedaDelTesoro "inicio" ((=='a').head) dicc2,
  Nothing ~=? búsquedaDelTesoro "auto" (=="alfajor") dicc2,
  Just "alfajor" ~=? búsquedaDelTesoro "ropero" ((=='a').head) dicc2
  ]
```

2. Arbol23.hs

```
module Arbol23 where

data Arbol23 a b = Hoja a
                 | Dos  b   (Arbol23 a b) (Arbol23 a b)
                 | Tres b b (Arbol23 a b) (Arbol23 a b) (Arbol23 a b)

{- Funciones para mostrar el árbol. -}
```

```
instance (Show a, Show b) => Show (Arbol23 a b) where
    show = ("\n" ++) . (padTree 0 0 False)

padlength :: Int
padlength = 5

padTree:: (Show a, Show b) => Int -> Int -> Bool -> (Arbol23 a b)-> String
padTree nivel acum doPad t = case t of
  (Hoja x) -> initialPad ++ stuff x
  (Dos x i d) -> initialPad ++ stuff x ++
                 pad padlength ++ rec x False i ++ "\n" ++
                 rec x True d ++ "\n"
  (Tres x y i m d) -> initialPad ++ stuff x ++ --(' ':tail (stuff y)) ++
                      pad padlength ++ rec x False i ++ "\n" ++
                      pad levelPad ++ stuff y ++ pad padlength
                      ++ rec x False m ++ "\n" ++
                      rec x True d ++ "\n"
  where l = length . stuff
        levelPad = (padlength*nivel + acum)
        initialPad = (if doPad then pad levelPad else "")
        rec x = padTree (nivel+1) (acum+l x)

stuff :: Show a => a -> String
stuff x = if n > l then pad (n-l) ++ s else s
  where s = show x
        l = length s
        n = padlength

pad :: Int -> String
pad i = replicate i ' '

{- Funciones pedidas. -}

foldA23 :: (a -> c) -> (b -> c -> c -> c) -> (b -> b -> c -> c -> c -> c)->
  Arbol23 a b -> c
foldA23 fHoja fDos fTres t = case t of
    Hoja x              -> fHoja x
    Dos  k r1 r2        -> fDos  k     (f r1) (f r2)
    Tres k1 k2 r1 r2 r3 -> fTres k1 k2 (f r1) (f r2) (f r3)
  where f = foldA23 fHoja fDos fTres

--Lista en preorden de los internos del árbol.
internos :: Arbol23 a b -> [b]
internos = foldA23 (const [])
                   (\ k     x y   -> [k] ++ x ++ y)
                   (\ k1 k2 x y z -> [k1, k2] ++ x ++ y ++ z)

--Lista las hojas de izquierda a derecha.
```

```
hojas :: Arbol23 a b -> [a]
hojas = foldA23 (:[])
                (\ _ x y -> x ++ y)
                (\ _ _ x y z -> x ++ y ++ z)


-- True si el arbol es hoja, falso si es Dos o Tres
esHoja :: Arbol23 a b -> Bool
esHoja a = case a of
  Hoja _    -> True
  otherwise -> False

-- Map de un Arbol23 a otro
mapA23 :: (a -> c) -> (b -> d) -> Arbol23 a b -> Arbol23 c d
mapA23 f g = foldA23 (Hoja . f)
                     (Dos . g)
                     (\ x -> Tres (g x) . g)

--Ejemplo de uso de mapA23.
--Incrementa en 1 el valor de las hojas.
incrementarHojas :: Num a => Arbol23 a b -> Arbol23 a b
incrementarHojas = mapA23 (+1) id

-- aplica sucesivamente n veces la funcion f, con neutro z.
foldNat :: a -> (a -> a) -> Integer -> a
foldNat z f n = case n of
  0 -> z
  _ -> f (foldNat z f (n-1))

--Trunca el árbol hasta un determinado nivel. Cuando llega a 0, reemplaza el
--resto del árbol por una hoja con el valor indicado.
--Funciona para árboles infinitos.
truncar :: a -> Integer -> (Arbol23 a b -> Arbol23 a b)
truncar z = foldNat (const (Hoja z)) crecer

-- dada f, devuelve una funcion que aplica f a los subarboles de un nodo
crecer :: (Arbol23 a b -> Arbol23 a b) -> (Arbol23 a b -> Arbol23 a b)
crecer f = foldA23 Hoja fDos fTres
  where fDos  x   r1 r2    = Dos  x   (f r1) (f r2)
        fTres x y r1 r2 r3 = Tres x y (f r1) (f r2) (f r3)

-- truncar pero mucho más rápido. testeado con truncar' 0 20 arbolito3'
truncar' :: a -> Integer -> Arbol23 a b -> Arbol23 a b
truncar' z n a = hastaNivel z n (conNiveles a)

-- decora un arbol con su numero de nivel,
-- la raiz es 1, el primer nivel es 2, etc.
conNiveles :: Arbol23 a b -> Arbol23 (a, Integer) (b, Integer)
conNiveles = foldA23 h d t
```

```
  where h x            =         Hoja (z x)
        d x   r1 r2    = mapInc (Dos  (z x)       r1 r2)
        t x y r1 r2 r3 = mapInc (Tres (z x) (z y) r1 r2 r3)
        z x = (x, 0)
        mapInc = mapA23 inc inc
        inc (x, n) = (x, n+1)

-- devuelve el arbol decorado hasta el nivel indicado,
-- reemplazando el nivel cortado por (Hoja z)
hastaNivel :: a -> Integer -> (Arbol23 (a, Integer) (b, Integer) -> Arbol23 a b)
hastaNivel z n = foldA23 h d t
  where h (x, xn)                 = f xn (Hoja x)             z'
        d (x, xn)        r1 r2    = f xn (Dos  x   r1 r2)    (Dos  x   z' z')
        t (x, xn) (y, _) r1 r2 r3 = f xn (Tres x y r1 r2 r3) (Tres x y z' z' z')
        f nivel orig trunco = if nivel < n then orig else trunco
        z' = Hoja z

--Evalúa las funciones tomando los valores de los hijos como argumentos.
--En el caso de que haya 3 hijos, asocia a izquierda.
evaluar :: Arbol23 a (a -> a -> a) -> a
evaluar = foldA23 id id (\ f g a b -> g (f a b))

--Ejemplo:
--evaluar (truncar 0 6 arbolito3) = 22 = (1*2-3)+(2*3-4)+(3*4-5)+(4*5-6)

{- Árboles de ejemplo. -}
arbolito1 :: Arbol23 Char Int
arbolito1 = Tres 0 1
        (Dos 2 (Hoja 'a') (Hoja 'b'))
        (Tres 3 4 (Hoja 'c') (Hoja 'd') (Dos 5 (Hoja 'e') (Hoja 'f')))
        (Dos 6 (Hoja 'g') (Dos 7 (Hoja 'h') (Hoja 'i')))

arbolito2 :: Arbol23 Int Bool
arbolito2 = Dos True (Hoja (-1)) (Tres False True (Hoja 0) (Hoja (-2)) (Hoja 4))

arbolito3 :: Arbol23 Int (Int->Int->Int)
arbolito3 = Dos (+) (Tres (*) (-) (Hoja 1) (Hoja 2) (Hoja 3))
                    (incrementarHojas arbolito3)

arbolito3' :: Arbol23 Int String
arbolito3' = Dos "(+)" (Tres "(*)" "(-)" (Hoja 1) (Hoja 2) (Hoja 3))
                 (incrementarHojas arbolito3')

arbolito4 :: Arbol23 Int Char
arbolito4 = Dos 'p' (Dos 'l' (Dos 'g' (Hoja 5) (Hoja 2))
                             (Tres 'r' 'a' (Hoja 0)(Hoja 1)(Hoja 12)))
                    (Dos 'p' (Tres 'n' 'd' (Hoja (-3))(Hoja 4)(Hoja 9))
                             (Dos 'e' (Hoja 20)(Hoja 7)))
```

3. Diccionario.hs

```
module Diccionario (Diccionario, vacio, definir, definirVarias,
obtener, claves, cmp, estructura, dicc1, dicc2, dicc3) where

import Data.Maybe
import Data.List
import Arbol23

{- Definiciones de tipos. -}

type Comp clave = clave->clave->Bool
type Estr clave valor = Arbol23 (clave,valor) clave

data Diccionario clave valor =
    Dicc {cmp :: Comp clave, estructura :: Maybe (Estr clave valor)}
--El comparador es por menor.

{- Funciones provistas por la cátedra. -}

-- Inserta un nuevo par clave,
-- valor en una estructura que ya tiene al menos un dato.
insertar :: clave -> valor -> Comp clave -> Estr clave valor -> Estr clave valor
insertar c v comp a23 =
    interceptar (insertarYPropagar c v comp a23) id (\s1 (c1, s2)->Dos c1 s1 s2)

--Maneja el caso de que la segunda componente sea Nothing.
interceptar :: (a, Maybe b) -> (a -> c) -> (a -> b -> c) -> c
interceptar (x, y) f1 f2 = maybe (f1 x) (f2 x) y

{- Inserta una clave con su valor correspondiente. Si se actualiza el índice,
   el cambio se propaga hacia arriba para mantener balanceado el árbol.
   Usamos recursión explícita porque este tipo de recursión no es estructural
   (no se aplica a todos los hijos). -}
insertarYPropagar::clave->valor->Comp clave->Estr clave valor->
    (Estr clave valor, Maybe (clave, Estr clave valor))
insertarYPropagar c v comp a23 =
    let rec = insertarYPropagar c v comp in case a23 of
    --Si es hoja, elegimos la máxima de las claves
    --y propagamos el balanceo hacia arriba.
    Hoja (ch,vh) -> if comp c ch
                then (Hoja (c,v), Just (ch, Hoja (ch,vh)))
                else (Hoja (ch, vh), Just (c, Hoja (c,v)))
    {- Si el actual es Nodo-Dos, o se queda en forma Nodo-Dos o se transforma en
       Nodo-Tres; no puede ocurrir que haya propagación hacia arriba
       (retornamos Nothing). -}
    Dos c1 a1 a2 -> (if comp c c1
                then
                -- La clave c va del lado izquierdo.
```

```
                    interceptar (rec a1)
                        (\s1 -> Dos c1 s1 a2)
                        (\s1 (c3, s2) -> Tres c3 c1 s1 s2 a2)
                else
                -- La clave c va del lado derecho.
                    interceptar (rec a2)
                        (\s1 -> Dos c1 a1 s1)
                        (\s1 (c3, s2) -> Tres c1 c3 a1 s1 s2), Nothing)
    {- Nodo-tres sólo propaga si de abajo propagan, los tres casos
    son muy similares. Sólo cambia en que árbol se inserta. -}
    Tres c1 c2 a1 a2 a3 -> if comp c c1
                then
                    -- La clave debe ir en el primer árbol.
                    interceptar (rec a1)
                        (\s1 -> (Tres c1 c2 s1 a2 a3, Nothing))
                        (\s1 (c3, s2) -> (Dos c3 s1 s2, Just(c1, Dos c2 a2 a3)))
                else if comp c c2
                then
                    -- La clave debe ir en el árbol del medio.
                    interceptar (rec a2)
                        (\s1 -> (Tres c1 c2 a1 s1 a3, Nothing))
                        (\s1 (c3, s2) -> (Dos c1 a1 s1, Just(c3, Dos c2 s2 a3)))
                else
                    --La clave debe ir en el último árbol.
                    interceptar (rec a3)
                        (\s1 -> (Tres c1 c2 a1 a2 s1, Nothing))
                        (\s1 (c3, s2) -> (Dos c1 a1 a2, Just(c2, Dos c3 s1 s2)))

--Se asume que la lista no tiene claves repetidas.
definirVarias :: [(clave, valor)] -> Diccionario clave valor ->
    Diccionario clave valor
definirVarias = (flip.foldr.uncurry) definir

{- Funciones a implementar. -}

vacio :: Comp clave -> Diccionario clave valor
vacio c = Dicc c Nothing

definir :: clave -> valor -> Diccionario clave valor -> Diccionario clave valor
definir k v d = Dicc comp (Just (definir' (estructura d)))
  where
    definir' = maybe (Hoja (k, v)) (insertar k v comp)
    comp = cmp d

obtener :: Eq clave => clave -> Diccionario clave valor -> Maybe valor
obtener k d = maybe Nothing obtener' (estructura d)
  where
    obtener' = valor . until esHoja (bajarNivel (cmp d) k)
    valor (Hoja (k', v)) = if k == k' then Just v else Nothing
```

```
bajarNivel :: Eq clave => Comp clave -> clave ->
    (Estr clave valor -> Estr clave valor)
bajarNivel comp k a23 = case a23 of
  Hoja h                          -> Hoja h
  Dos  k1    a1 a2    | comp k k1 -> a1
  Dos  k1    a1 a2    | otherwise -> a2
  Tres k1 k2 a1 a2 a3 | comp k k1 -> a1
  Tres k1 k2 a1 a2 a3 | comp k k2 -> a2
  Tres k1 k2 a1 a2 a3 | otherwise -> a3

claves :: Diccionario clave valor -> [clave]
claves d = maybe [] ((map fst) . hojas) (estructura d)

{- Diccionarios de prueba: -}

dicc1 :: Diccionario Int String
dicc1 = definirVarias [
    (0,"Hola"),
    (-10,"Chau"),
    (15,"Felicidades"),
    (2,"etc."),
    (9,"a")
    ] (vacio (<))

dicc2 :: Diccionario String String
dicc2 = definirVarias [
    ("inicio","casa"),
    ("auto","flores"),
    ("calle","auto"),
    ("casa","escalera"),
    ("ropero","alfajor"),
    ("escalera","ropero")
    ] (vacio (<))

dicc3 :: Diccionario Int String
dicc3 = definirVarias [
    (0,"Hola"),
    (-10,"Chau"),
    (15,"Felicidades"),
    (2,"etc."),
    (9,"a")
    ] (vacio (\x y->x `mod` 5 < y `mod` 5))
```