Trabajo Práctico 1
Programación Funcional

Exploradores

Paradigmas de Lenguajes de Programación
2do cuatrimestre, 2016

Fecha de entrega: martes 6 de septiembre

Introducción

En este TP analizaremos distintas maneras de explorar las estructuras ya conocidas (números naturales, listas, árboles binarios y RoseTrees). Para esto, definimos el tipo `Explorador` de la siguiente manera.

```
type Explorador a b = a -> [b]
```

Conceptualmente, un `Explorador` es una función que analiza una estructura y arroja resultados (pudiendo arrojar uno, varios, ninguno o incluso infinitos).

Por ejemplo, los siguientes exploradores recorren listas y extraen información de ellas:

- `map even` ⇜ Para cada elemento, un resultado: `True` si es par y `False` si es impar.
- `\xs->[sum xs, length xs]` ⇜ Devuelve dos resultados: la suma y la longitud de la lista.
- `concat` ⇜ Extrae los contenidos de cada elemento de una lista de listas.

Contamos ya con la definición de las distintas estructuras, así como también las funciones para compararlas por igual y mostrarlas en pantalla.

Implementación

A continuación se detallan los problemas a resolver.

Exploradores básicos

En esta sección definiremos algunos exploradores sencillos para ir entrando en tema.

Ejercicio 1

Implementar los siguientes exploradores:

1. `expNulo :: Explorador a b`, que acepta cualquier estructura y no devuelve resultados.
2. `expId :: Explorador a a`, que devuelve como único resultado la estructura explorada.
3. `expHijosRT :: Explorador (RoseTree a) (RoseTree a)`, que dado un RoseTree devuelve como resultados todos sus hijos, es decir los RoseTrees que salen directamente de la raíz.
4. `expHijosAB :: Explorador(AB a) (AB a)`, similar a la anterior pero para árboles binarios (debe devolver 2 resultados si el árbol tiene raíz, y ninguno si es `Nil`).
5. `expTail :: Explorador [a] a`, que devuelve todos los elementos de la lista menos el primero, si lo hay (si no lo hay, no devuelve resultados, es decir, devuelve una lista vacía).

Esquemas de recursión

Para poder recorrer las distintas estructuras adecuadamente, vamos a necesitar sus esquemas de recursión estructural. Para el caso de las listas ya contamos con `foldr` y `foldl`. Para las otras estructuras vamos a necesitar esquemas similares.

Ejercicio 2

Implementar las funciones `foldNat`, `foldAB` y `foldRT` para poder recorrer números naturales, árboles binarios y RoseTrees respectivamente. Indicar el tipo de cada función.

Para los naturales, pueden utilizar el tipo `Integer`. Es aconsejable devolver un error declarativo en caso de que la función se aplique a un número negativo.

Para los RoseTrees existen distintos esquemas de recursión más o menos estructural (ver clase de Fidel). Pueden usar el que les resulte más cómodo para resolver los ejercicios, pero no definan esquemas más complejos que los vistos en clase.

Nota: efectivamente, pueden resolver este ejercicio con funciones vistas en clase. Esto es solo un paso preliminar para poder resolver las otras partes del TP. Pueden (y deben) utilizar recursión explícita en esta parte, ya que justamente se están definiendo esquemas de recursión.

Más exploradores

Ejercicio 3

Implementar los siguientes exploradores de listas.

1. `singletons :: Explorador [a] [a]`, que devuelve, para cada elemento de la lista, una lista con ese elemento.

   Por ejemplo: `singletons [1,2,3,4,5,6]` $\rightsquigarrow$ `[[1],[2],[3],[4],[5],[6]]`

2. `sufijos :: Explorador [a] [a]`, que devuelve todos los sufijos de la lista explorada.

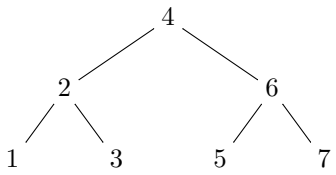
   Por ejemplo: `sufijos ‘‘abc’’` $\rightsquigarrow$ `[‘‘abc’’,‘‘bc’’,‘‘c’’,‘‘’’]`

Ejercicio 4

Definir el explorador de números naturales `listasQueSuman :: Explorador Integer ?` (poner el tipo que corresponde en lugar del `?`), que devuelve la lista de todas las listas de enteros positivos cuya suma sea igual al número explorado. Sí, es la misma función de la práctica, pero vista como explorador. Se puede usar recursión explícita. Explicar por qué el esquema `foldNat` no es adecuado para resolver este problema.

Ejercicio 5

Definir los exploradores clásicos para recorrer árboles binarios: `preorder`, `inorder` y `postorder` (e indicar qué tipo de exploradores son). Por ejemplo, sea el siguiente árbol:



los resultados de sus respectivos recorridos son:

- `preorder: [4,2,1,3,6,5,7]`
- `inorder: [1,2,3,4,5,6,7]`
- `postorder: [1,3,2,5,7,6,4]`

Ejercicio 6

Definir los siguientes exploradores de RoseTrees:

1. `dfsRT :: Explorador (RoseTree a) a`, que devuelve el recorrido `DFS` del `RoseTree` explorado (similar a `preorder`, pero para RoseTrees).
2. `hojasRT :: Explorador (RoseTree a) a`, que devuelve las hojas del RoseTree explorado (es decir, los nodos sin hijos).
3. `ramasRT :: Explorador (RoseTree a) [a]`, que devuelve la lista de caminos desde la raíz hasta las hojas.

Operaciones sobre exploradores

Implementar las siguientes funciones:

Ejercicio 7

`ifExp :: (a->Bool) -> Explorador a b -> Explorador a b -> Explorador a b`,
que combina condicionalmente dos exploradores de acuerdo a una condición. Si la estructura explorada satisface la condición, entonces le aplica el primer explorador. De lo contrario, le aplica el segundo.

Por ejemplo: `ifExp even expId expNulo` es un explorador de enteros que, al aplicarse a un entero `x`, devuelve `[x]` si `x` es par, y `[]` si `x` es impar.

Ejercicio 8

`(<++>) :: Explorador a b -> Explorador a b -> Explorador a b`,
que aplica dos exploradores a una misma estructura y concatena sus resultados.

Sea `ab` el árbol binario del ejemplo del ejercicio 5:
`(preorder <++> inorder) ab` $\rightsquigarrow$ `[4,2,1,3,6,5,7,1,2,3,4,5,6,7]`

Ejercicio 9

`(<.>) :: Explorador b c -> Explorador a b -> Explorador a c`,
que compone dos exploradores de la siguiente manera: dada una estructura de tipo `a`, le aplica el segundo explorador obteniendo una lista de resultados de tipo `b`, y a cada uno de estos resultados se le aplica el primer explorador, para luego unir los resultados finales en una lista de elementos de tipo `c`.

Por ejemplo `((\x->[1..x]) <.> (map (+1))) [2,4]` ⇝ `[1,2,3,1,2,3,4,5]`

Ejercicio 10

`(<^>) :: Explorador a a -> Integer -> Explorador a a`,
que aplica un mismo explorador tantas veces como lo indique el número recibido (que debe ser natural), primero a la estructura original, luego a cada uno de los resultados y así sucesivamente.

Por ejemplo, sea `ab` el árbol de los ejemplos anteriores, `(expHijosAB <^>2) ab` devuelve una lista de árboles cuyas raíces son 1, 3, 5 y 7 y cuyos hijos son todos `Nil`.

Las difíciles

Ejercicio 11

Las siguientes funciones generan listas infinitas. Para aprobar este TP deberán implementar **al menos una** de estas funciones.

1. `listasDeLongitud :: Explorador Integer [Integer]`, que dado un número natural l devuelve todas las listas de enteros positivos (mayores o iguales que 1) cuya longitud sea l.

   Ejemplo: `take 6 $ listasDeLongitud 2` ⇝ `[[1,1],[1,2],[2,1],[1,3],[2,2],[3,1]]`

   Debe poder generar en menos de 5 minutos cualquier lista de longitud menor o igual que 3 con números de 1 o 2 dígitos. Por ejemplo, `elem [99,99,99] $ listasDeLongitud 3` debe devolver `True` en menos de 5 minutos.

   Para esto se recomienda usar una función auxiliar que devuelva una función cuyo parámetro vaya acotando los contenidos de las listas. Notar que [] es la única lista de longitud 0.

2. `(<*>) :: Explorador a a -> Explorador a [a]`,

   que compone un explorador consigo mismo tantas veces como sea posible, siempre que quede – en la lista generada por el paso anterior – algún resultado por explorar. La lista resultante puede ser o no infinita. Por ejemplo, `expId <*> 1` es la lista que contiene infinitas veces a la lista `[1]`, pero `expHijosAB <*> ab` – con `ab` el árbol de los ejemplos anteriores – devuelve una lista que contiene 4 listas de árboles binarios: una con el propio `ab`, una con sus hijos, otra con los hijos de sus hijos, y la última con 8 veces el árbol `Nil`.

   Sugerencia: usar las funciones `iterate` y `takeWhile`.

Pautas de Entrega

Se debe entregar el código impreso con la implementación de las funciones pedidas. Cada función asociada a los ejercicios debe contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el código fuente en Haskell a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-PF]` seguido inmediatamente del nombre del grupo.
- El código Haskell debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto (puede adjuntarse un `.zip` o `.tar.gz`).
- El código entregado **debe** incluir tests que permitan probar las funciones definidas.

El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté **adecuadamente** comentado (son comentarios adecuados los que ayudan a entender lo que no es evidente o explican decisiones tomadas; no son adecuadas las traducciones al castellano del código). Los objetivos a evaluar son:

- Corrección.
- Declaratividad.
- Prolijidad: evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso adecuado de funciones de alto orden, currificación y esquemas de recursión: Es necesario para los ejercicios que usen las funciones que vimos en clase y aprovecharlas, por ejemplo, usar zip, map, filter, take, takeWhile, dropWhile, foldr, foldl, listas por comprensión, etc, cuando sea necesario y no volver a implementarlas.

Salvo donde se indique lo contrario, **no se permite utilizar recursión explícita**, dado que la idea del TP es aprender a aprovechar las características enumeradas en el ítem anterior. Se permite utilizar listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `List`, `Maybe`, `Data.Char`, `Data.Function`, `Data.List`, `Data.Maybe`, `Data.Ord` y `Data.Tuple`. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas. Pueden escribirse todas las funciones auxiliares que se requieran, pero estas no pueden usar recursión explícita (ni mutua, ni simulada con `fix`).

Importante: se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

Tests: se recomienda la codificación de tests. Tanto HUnit https://hackage.haskell.org/package/HUnit como HSpec https://hackage.haskell.org/package/hspec permiten hacerlo con facilidad.

Para instalar HUnit usar: `> cabal install hunit`

Para instalar cabal ver: https://wiki.haskell.org/Cabal-Install

Referencias del lenguaje Haskell

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la última versión del lenguaje Haskell a la fecha, disponible online en http://www.haskell.org/onlinereport/haskell2010.
- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en http://learnyouahaskell.com/chapters.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en http://book.realworldhaskell.org/read.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos *parciales*, online en http://www.haskell.org/hoogle.
- **Hayoo!**: buscador de módulos no estándar (i.e. aquéllos no necesariamente incluídos con la plataforma Haskell, sino a través de **Hackage**), online en http://holumbus.fh-wedel.de/hayoo/hayoo.html.