Tarea práctica de programación funcional

Universidad Simón Bolívar — CI3661

Abril–Julio 2015

Integrantes del grupo:

- 11-10428 gustavogut1993@gmail.com
- 11-10743 josepas979@gmail.com

Introducción

Este documento soporta la tarea práctica de programación funcional para CI3661 (el Laboratorio de Lenguajes de Programación 1) en Abril–Julio de 2015.

Puede cargar la fuente de este archivo en GHCi directamente con el comando:

```
ghci t2-11-10428_11-10743.lhs
```

Declaraciones preliminares

En esta sección puede agregar todas las directivas necesarias para importar símbolos de módulos adicionales a `Prelude` (que se importa implícitamente) y para dar opciones al compilador.

```
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE TypeSynonymInstances #-}

import Control.Applicative (pure)
import Control.DeepSeq     (NFData, ($!!))
import Control.Monad       (void)
import Data.Map            (Map, empty, singleton, fromList, foldWithKey)
import GHC.Generics        (Generic)
import System.Environment  (getArgs, getProgName)
import System.IO           (hPutStrLn, stderr)
```

Expresiónes aritméticas

Considere la siguiente declaración de un tipo algebráico para representar expresiones aritméticas con números de punto flotante:

```
data Expresión
  = Suma           Expresión Expresión
  | Resta          Expresión Expresión
  | Multiplicación Expresión Expresión
  | División       Expresión Expresión
  | Negativo       Expresión
  | Literal        Integer
  deriving (Eq, Read, Show)
```

Ejercicio 1 (0.2 puntos cada una; 0.6 puntos en total): Complete las siguientes definiciones para que `t1`, `t2` y `t3` representen a las respectivas expresiones aritméticas:

```
t1 = 42
t2 = 27 + t1
t3 = (t2 * (t2 * 1)) + (- ((t1 + 0) / 3))
```

```
t1, t2, t3 :: Expresión
t1 = Literal 42
t2 = Suma (Literal 27) t1
t3 = Suma (Multiplicación t2 (Multiplicación t2 (Literal 1)))
          (Negativo (División (Suma t1 (Literal 0))(Literal 3)))
```

Catamorfismos

Los valores de cada tipo algebraico en *Haskell* son datos que representan la estructura abstracta de una operación, sin especificar cuál es la operación particular que debe hacerse. Un valor de un tipo algebraico puede analizarse para convertirlo en una operación particular: por ejemplo, el valor `t2` representa la estructura abstracta de la operación «la suma de 42 y 27», sin decir qué significa «la suma».

Un valor de esta forma puede convertirse en una operación particular de varias maneras. Si bien la interpretación más obvia de esa operación abstracta corresponde con la suma aritmética de los números 42 y 27, que produce un número, también podrían realizarse *otras* operaciones siguiendo la misma estructura. Por ejemplo, puede buscarse cuál es el máximo operando de todas las operaciones, en cuyo caso la operación especificada como «suma» sería tomar el máximo entre dos números, y en el caso de `t2` produciría como resultado el número 42.

En el contexto de *Haskell*, un **catamorfismo** es cualquier transformación de un tipo algebraico a una operación en otro tipo que se haga de esta manera: según la estructura del valor del tipo algebraico.

Ejercicio 2 (0.5 puntos): Complete la siguiente definición que calcule el resultado de evaluar una expresión aritmética.

```
evaluar :: Expresión -> Double
evaluar e
  = case e of
      Suma           e1 e2 -> evaluar e1 + evaluar e2
      Resta          e1 e2 -> evaluar e1 - evaluar e2
      Multiplicación e1 e2 -> evaluar e1 * evaluar e2
      División       e1 e2 -> evaluar e1 / evaluar e2
      Negativo       e     -> evaluar e * (-1)
      Literal        n     -> fromInteger n
```

En particular,

```
evaluar t1 == 42.0
evaluar t2 == 69.0
evaluar t3 == 4747.0
```

Ejercicio 3 (0.25 puntos): Complete la siguiente definición que calcule la cantidad de operaciones aritméticas especificadas en una expresión aritmética. Se considera que la expresión correspondiente a un simple literal especifica cero operaciones.

```
operaciones :: Expresión -> Integer
operaciones e
  = case e of
      Literal        n     -> 0
```

```
      Negativo       e     -> 1 + operaciones e
      Suma           e1 e2 -> 1 + operaciones e1 + operaciones e2
      Resta          e1 e2 -> 1 + operaciones e1 + operaciones e2
      Multiplicación e1 e2 -> 1 + operaciones e1 + operaciones e2
      División       e1 e2 -> 1 + operaciones e1 + operaciones e2
```

En particular,

```
operaciones t1 == 0
operaciones t2 == 1
operaciones t3 == 8
```

Ejercicio 4 (0.25 puntos): Complete la siguiente definición que calcule la suma de todos los literales presentes en una expresión aritmética.

```
sumaLiterales :: Expresión -> Integer
sumaLiterales e
  = case e of
      Literal        n     -> n
      Negativo       e     -> sumaLiterales e
      Suma           e1 e2 -> sumaLiterales e1 + sumaLiterales e2
      Resta          e1 e2 -> sumaLiterales e1 + sumaLiterales e2
      Multiplicación e1 e2 -> sumaLiterales e1 + sumaLiterales e2
      División       e1 e2 -> sumaLiterales e1 + sumaLiterales e2
```

En particular,

```
sumaLiterales t1 == 42
sumaLiterales t2 == 69
sumaLiterales t3 == 184
```

Ejercicio 5 (0.25 puntos): Complete la siguiente definición que calcule la lista de todos los literales presentes en una expresión aritmética.

```
literales :: Expresión -> [Integer]
literales e
  = case e of
      Literal        n     -> n:[]
      Negativo       e     -> literales e
```

```
      Suma           e1 e2 -> literales e1 ++ literales e2
      Resta          e1 e2 -> literales e1 ++ literales e2
      Multiplicación e1 e2 -> literales e1 ++ literales e2
      División       e1 e2 -> literales e1 ++ literales e2
```

En particular,

```
literales t1 == [42]
literales t2 == [27, 42]
literales t3 == [27, 42, 27, 42, 1, 42, 0, 3]
```

Ejercicio 6 (0.25 puntos): Complete la siguiente definición que calcule la altura de una expresión aritmética. Se considera que un literal es una expresión aritmética de altura cero, y que todas las demás operaciones agregan uno a la altura.

```
altura :: Expresión -> Integer
altura e
  = case e of
      Literal        n     -> 0
      Negativo       e     -> 1 + altura e
      Suma           e1 e2 -> 1 + max (altura e1) (altura e2)
      Resta          e1 e2 -> 1 + max (altura e1) (altura e2)
      Multiplicación e1 e2 -> 1 + max (altura e1) (altura e2)
      División       e1 e2 -> 1 + max (altura e1) (altura e2)
```

En particular,

```
altura t1 == 0
altura t2 == 1
altura t3 == 4
```

Catamorfismo generalizado

Los catamorfismos de las preguntas anteriores deben seguir un patrón común: cada constructor del tipo `Expresión` se hace corresponder con una operación en el tipo del resultado del catamorfismo. Esas operaciones se realizan con los valores almacenados en cada constructor, y en el caso de las ocurrencias

anidadas de valores del tipo `Expresión`, éstas se transforman a valores del tipo del resultado con una invocación recursiva al catamorfismo.

En efecto, todo catamorfismo se construye de la misma forma para un tipo algebraico dado — solo es necesario especificar de qué manera se combinan a un valor del tipo resultante los datos obtenidos de cada constructor.

Ejercicio 7 (0.5 puntos): Complete la siguiente definición para el catamorfismo generalizado del tipo `Expresión`.

```
cataExpresión
  :: (a -> a -> a)
  -> (a -> a -> a)
  -> (a -> a -> a)
  -> (a -> a -> a)
  -> (a -> a)
  -> (Integer -> a)
  -> Expresión -> a

cataExpresión
  suma
  resta
  multiplicacion
  division
  negativo
  literal
  e
  = case e of
      Literal        n     -> literal n
      Negativo       e     -> negativo (currificada e)
      Suma           e1 e2 -> suma (currificada e1) (currificada e2)
      Resta          e1 e2 -> resta (currificada e1) (currificada e2)
      Multiplicación e1 e2 -> multiplicacion (currificada e1) (currificada e2)
      División       e1 e2 -> division (currificada e1) (currificada e2)
   where currificada = cataExpresión suma resta multiplicacion division negativo literal
```

Ejercicio 8 (0.2 puntos cada una; 1 punto en total): Complete las siguientes definiciones para los catamorfismos que definió en las preguntas anteriores, esta vez en términos de `cataExpresión`.

```haskell
evaluar' :: Expresión -> Double
evaluar' = cataExpresión (+) (-) (*) (/) (negate) (fromInteger)

operaciones' :: Expresión -> Integer
operaciones' = cataExpresión (aumentar) (aumentar) (aumentar) (aumentar) (+1) (*0)
                  where aumentar x y = 1 + x + y

sumaLiterales' :: Expresión -> Integer
sumaLiterales' = cataExpresión (+) (+) (+) (+) (id) (id)

literales' :: Expresión -> [Integer]
literales' = cataExpresión (++) (++) (++) (++) (id) (:[])

altura' :: Expresión -> Integer
altura' = cataExpresión (aumentar) (aumentar) (aumentar) (aumentar) (1+) (*0)
              where aumentar x y = 1 + max x y
```

Lenguajes de marcado

Los lenguajes de marcado descendientes de SGML se utilizan para especificar documentos jerárquicos, donde el texto del documento se incluye en *elementos* que pueden anidarse y se clasifican según el *nombre de etiqueta* de cada uno. Además, los elementos pueden asociarse con un diccionario de *atributos* textuales identificados por un nombre de atributo.

Puede representarse una versión simplificada de esta idea con los siguientes tipos de datos algebraicos de *Haskell*:

```haskell
type Atributos
  = Map String String

newtype Documento
  = Documento Elemento
  deriving Show

data Elemento
  = Elemento String Atributos [Elemento]
  | Texto String
  deriving Show
```

Los documentos completos están formados por un elemento como raíz. Los elementos pueden tener un nombre de etiqueta, un diccionario de atributos

(usando el tipo `Map` definido en el módulo `Data.Map`), y una lista de elementos anidados — salvo en el caso de los elementos que sencillamente contienen texto.

Combinadores

Ejercicio 9 (0.15 puntos cada una; 0.6 puntos en total): Complete las siguientes definiciones para combinadores que produzcan representaciones de los elementos de XHTML `html`, `head`, `body` y `div` a partir de una lista de elementos anidados dentro de ellos. Los elementos resultantes de aplicar estos combinadores deben tener diccionarios de atributos vacíos, salvo en el caso del elemento `html`, que debe incluir exactamente un atributo que debe llamarse `xmlns` y asociarse al valor `http://www.w3.org/1999/xhtml`.[1]

```
htmlE, headE, bodyE, divE :: [Elemento] -> Elemento
htmlE  = Elemento "html" (singleton "xmlns" "http://www.w3.org/1999/xhtml")
headE  = Elemento "head" empty
bodyE  = Elemento "body" empty
divE   = Elemento "div"  empty
```

Ejercicio 10 (0.15 puntos cada una; 0.6 puntos en total): Complete las siguientes definiciones para combinadores que produzcan representaciones de los elementos de XHTML `title`, `style`, `h1` y `p` a partir de un `String` con el texto que debe incluirse dentro de ellos. Los elementos resultantes de aplicar estos combinadores deben tener diccionarios de atributos vacíos, salvo el elemento `style` que debe tener el atributo `type` asociado al texto `text/css`.

```
styleE, titleE, h1E :: String -> Elemento
styleE s = Elemento "style" (singleton "type" "text/css") [Texto s]
titleE s = Elemento "title" empty [Texto s]
h1E    s = Elemento "h1"    empty [Texto s]
pE     s = Elemento "p"     empty [Texto s]
```

Ejercicio 11 (0.2 puntos): Complete la siguiente definición para un combinador que produzca una representación del elemento de XHTML `p` a partir de un valor de cualquier tipo `a` que pertenezca a la clase de tipos `Show`; el elemento

[1] Se utiliza el sufijo `E` para evitar conflictos con los nombres `head` y `div` importados implícitamente desde el módulo `Prelude` de *Haskell*.

`p` resultante de aplicar este combinador debe contener únicamente un nodo de texto cuyo `String` sea el resultante de aplicar la función `show` al valor pasado como parámetro, y debe tener su diccionario de atributos vacío.

```
showP :: Show a => a -> Elemento
showP = pE . show
```

Generación de XHTML

Considere la siguiente clase de tipos:

```
class RenderXHTML a where
  render :: a -> String
```

Se desea usar el método `render` para generar texto XHTML a partir de un valor de cualquier tipo que sea instancia de esta clase de tipos.[2] Para convertir un `Documento` a su código XHTML, se declara que `Documento` es una instancia de `RenderXHTML`:

```
instance RenderXHTML Documento where
  render (Documento raíz)
    = encabezado ++ render raíz
    where
      encabezado
        = unlines
          [ "<?xml version='1.0' encoding='UTF-8'?>"
          , "<!DOCTYPE html"
          , "     PUBLIC '-//W3C//DTD XHTML 1.0 Strict//EN'"
          , "     'http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd'>"
          ]
```

Ejercicio 12 (1.25 puntos): Escriba una instancia de la clase `RenderXHTML` para el tipo `Atributos`. Puede suponer que las claves de los diccionarios de atributos únicamente contienen nombres de atributos válidos para XHTML, y que los valores de atributos no contienen entidades ilegales en XHTML ni comillas — es decir, no es necesario que se preocupe por escapar el texto obtenido del diccionario de atributos.

[2] Note que esta clase es estructuralmente idéntica a la clase `Show` de *Haskell*. Sin embargo, recuerde que el propósito de la clase `Show` es construir representaciones textuales de valores de *Haskell* para asistir al programador a estudiar un tipo de datos y visualizar sus valores — la clase `RenderXHTML`, en cambio, existe para generar código XHTML a partir de algunos tipos de datos que puedan convertirse de esa manera.

```haskell
instance RenderXHTML Atributos where
  render = foldWithKey f ""
           where f k v acc = acc ++ " " ++ k ++ "='" ++ v ++ "'"
```

Ejercicio 13 (1.25 puntos): Escriba una instancia de la clase `RenderXHTML` para el tipo `Elemento`. Puede suponer que los nombres de etiquetas de los elementos siempre son válidos para XHTML, y que los nodos de texto no contienen entidades ilegales ni caracteres reservados por XHTML — es decir, no es necesario que se preocupe por escapar el texto obtenido del elemento.

El texto generado debe corresponder a una etiqueta XHTML para el elemento dado. Por ejemplo, para un elemento con el nombre de etiqueta `welp`, y un hijo que sea un nodo textual con el texto `trolololololo`, debe generar el texto

```
<welp foo='bar baz' quux='meh' wtf='wow://such.example.com/amaze/'>trolololololo</welp>
```

o cualquier texto con el mismo significado en XHTML — el espacio en blanco, por ejemplo, es irrelevante.

```haskell
instance RenderXHTML Elemento where
  render e
    = case e of
          Texto s -> s
          Elemento s a xs -> "<" ++ s ++ render a ++ ">"  ++
              (concat (map render xs))  ++ "</" ++ s ++ ">"
```

XHTML para expresiones

Ejercicio 14 (1.25 puntos): Complete la siguiente definición que convierta un valor dado del tipo `Expresión` en un valor del tipo `Elemento` que represente a la estructura de la expresión aritmética con un árbol de elementos XHTML.

Escriba su definición en términos de `cataExpresión` y utilice los combinadores para elementos de XHTML que definió previamente.

```haskell
expresionXHTML :: Expresión -> Elemento
expresionXHTML e
  = case e of
      Literal        n     -> showP n
      Negativo       e     -> divE [pE "-", expresionXHTML e]
```

```
    Suma           e1 e2 -> divE [expresionXHTML e1, pE "+", expresionXHTML e2]
    Resta          e1 e2 -> divE [expresionXHTML e1, pE "-", expresionXHTML e2]
    Multiplicación e1 e2 -> divE [expresionXHTML e1, pE "*", expresionXHTML e2]
    División       e1 e2 -> divE [expresionXHTML e1, pE "/", expresionXHTML e2]
```

Ejercicio 15 (1.25 puntos): Complete la siguiente definición que convierta un valor dado del tipo `Expresión` en un valor del tipo `Documento` que muestre información sobre la expresión aritmética dada en un documento XHTML. Debe usar los combinadores definidos en ejercicios previos para implantar esta función.

```
expresiónDocumento :: Expresión -> Documento
expresiónDocumento e = Documento (htmlE [headE [titleE "Expresión", styleE estilo],
                                         bodyE [h1E "Expresión original", showP e,
                                                h1E "Estructura", expresionXHTML e,
                                                h1E "Valor", showP (evaluar' e),
                                                h1E "Altura", showP (altura' e),
                                                h1E "Número de operaciones", showP
                                                    (operaciones e),
                                                h1E "Literales", showP (literales' e)]
                                    ])
```

La siguiente definición contiene el texto necesario a incluir en el elemento `style` del documento a generar. El elemento `style` con este contenido debe ser incluido en el elemento `head` del documento generado.

```
estilo :: String
estilo
  = unlines
    [ "div, p {"
    , "  border: 1px solid black;"
    , "  float: left;"
    , "  margin: 1em;"
    , "}"
    , "h1 {"
    , "  clear: both;"
    , "}"
    ]
```

Programa principal

Se define un programa principal para poder compilar y ejecutar este archivo:

```
deriving instance Generic Expresión
instance NFData Expresión

main :: IO ()
main = do
  args <- getArgs
  case args of
    (nombreArchivo : expresiónTexto : _) -> do
      expresión <- pure $!! read expresiónTexto
      writeFile nombreArchivo . render $ expresiónDocumento expresión

    _ -> do
      progName <- getProgName
      hPutStrLn stderr $ "Uso: " ++ progName ++ " ARCHIVO.xhtml EXPRESIÓN"
```

Puede compilar la fuente de este archivo con el comando:

```
ghc t2-11-10428_11-10743.lhs -o expresión
```

y ejecuntarlo, por ejemplo, con:

```
./expresión expresión.xhtml 'Suma (Literal 42) (Literal 27)'
```