Pràctiques de Llenguatges, Tecnologies i Paradigmes de programació

Part II Programació Funcional

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Práctica 5: Mòduls i Polimorfisme en Haskell

Índex

1 Objectiu de la Pràctica

En aquesta pràctica es presenta l'ús dels mòduls en Haskell i s'introdueixen alguns conceptes bàsics del polimorfisme en Haskell. S'han previst 2 sessions per a resoldre els exercicis plantejats.

2 Mòduls

Un programa en Haskell consisteix bàsicament en una col·lecció de mòduls. Un mòdul de Haskell pot contenir definicions de funcions, de tipus de dades i de classes de tipus.

Com s'ha vist prèviament, des d'un mòdul és possible importar altres mòduls, per al que s'utilitza la sintaxi:

```
import ModuleName
```

que s'ha d'escriure abans de definir qualsevol funció, per la qual cosa usualment es posa al principi del propi mòdul.

Com es recordarà, el nom dels mòduls és alfanumèric i ha de començar amb majúscula. Addicionalment, el contingut d'un mòdul comença, a més, amb la paraula reservada `module`.

2.1 Importació de mòduls

Per poder importar un mòdul, cal que el seu nom coincideixca amb el nom del fitxer que el continga quan el mòdul importat i el que realitza la importació es troben en el mateix directori.

Si el mòdul que es vol importar no es troba en el mateix directori que aquell des del qual es realitze la importació, llavors cal nomenar el mòdul a ser importat prefixant el seu nom amb la seqüència (*path*) de directoris per arribar fins al mateix.

Per exemple, si es vol importar cert mòdul, que es trobe en el fitxer de nom `EjemImport.hs` situat al directori `A/B/C`, relatiu al mòdul en què es vullga fer la importació, llavors el mòdul a ser importat ha de tenir necessàriament per nom: `A.B.C.EjemImport`.

2.2 Llista d'exportació

Al costat del nom del mòdul, pot aparèixer una llista dels elements del mateix que es vulguin exportar perquè puguin ser utilitzats per altres mòduls, seguint la sintaxi:

```
module Nom ( llista del que s'exporta ) where
```

Si s'omet la llista d'exportació, llavors s'exporta tot el definit (tal i com s'ha fet fins al moment). Òbviament, és molt útil poder seleccionar el que s'exporta per poder oferir a l'exterior, únicament una interfície, ocultant detalls interns no rellevants. Per exemple, si s'escriu el següent mòdul al fitxer `Geometry2D.hs`:

```
module Geometry2D
( areaSquare
, perimeterSquare
) where

areaRectangle :: Float -> Float -> Float
areaRectangle base height = base * height

perimeterRectangle :: Float -> Float -> Float
perimeterRectangle base height = 2*(base+height)

areaSquare :: Float -> Float
areaSquare side = areaRectangle side side

perimeterSquare :: Float -> Float
perimeterSquare side = perimeterRectangle side side
```

I si a continuació proveu a executar el programa següent (escrit en un fitxer `test.hs`):

```
import Geometry2D
main = putStrLn ("The area is " ++ show (areaRectangle 2 3))
```

Text traduït S'observa que un programa defineix una funció `main`. Per a executar aquest programa, en lloc d'utilitzar l'intèrpret, GHCi, s'ha d'escriure el següent en la línia d'ordres:

```
bash$ runghc test.hs
```

com s'observarà, es mostra l'error:

```
test.hs:2:55: Not in scope: ‘areaRectangle’
```

Si a continuació es modifica, la definició de la funció `main` per la següent

```
main = putStrLn ("The area is " ++ show (areaSquare 2))
```

i torneu a provar l'execució amb la comanda RunGHC, ja no hi haurà cap error.

Com s'ha observat en aquest exemple, la funció `putStrLn` mostra una cadena per la eixida estàndar. Existeix un altra funció anomenada `putStr` pareguda a l'anterior amb la diferència de que no afegeix un canvi de linia. A més de compilar i executar un programa mitjançant `runghc` és posible simplement compilar-lo emprant `ghc` de la manera següent:

```
bash$ ghc --make test.hs
```

que genera un fitxer executable anomenat `test` es pot executar directament:

```
bash$ ./test
The area is 4.0
```

Quan es vol utilitzar diverses instruccions d'eixida en una mateixa funció es poden agrupar amb la notació `do` de la manera següent:

```
import Geometry2D
main = do
  putStrLn ("The area is " ++ show (areaSquare 2))
  let other = (areaSquare 5)
  putStrLn ("Another area is " ++ show other)
```

on la definició de variables dins del bloc `do` es fa emprant `let`.

2.3 Importacions qualificades

Què passa si dos mòduls tenen definicions amb els mateixos identificadors?. Vegem-ho amb un exemple: suposant que es té el mòdul:

```
module NormalizeSpaces where
  normalize :: String -> String
  normalize = unwords . words
```

que utilitza la funció `words` per fraccionar una cadena en una llista de paraules (ignorant espais, tabuladors i `enter` extras) i la funció `unwords` per formar de nou la cadena a partir de la llista. Suposant ara que hi ha un altre mòdul `NormalizeCase` amb una funció amb el mateix nom:

```
module NormalizeCase where
  import Data.Char (toLower) -- import only function toLower
  normalize :: String -> String
  normalize = map toLower
```

Importar-los simultàniament provocaria una col·lisió de noms. Per resoldre aquest problema, Haskell permet importar mòduls usant la paraula reservada *qualified* que fa que els identificadors definits per aquest mòdul tingan com a prefix el nom del seu mòdul:

```
module NormalizeAll where
  import qualified NormalizeSpaces
  import qualified NormalizeCase
  normalizeAll :: String -> String
  normalizeAll = NormalizeSpaces.normalize . NormalizeCase.normalize
```

Exercici 1 *Escriure un mòdul* `Circle.hs` *amb una funció* `area` *i un altre mòdul* `Triangle.hs` *amb una funció* `area`. *Després escriure un programa senzill que importe de manera qualificada la funció* `area` *de cada mòdul i que mostre per pantalla l'àrea d'un cercle de radi 2 i l'àrea d'un triangle de base 4 i alçària 5.*

3 Polimorfisme en Haskell

3.1 Polimorfisme paramètric

En pràctiques anteriors hem utilitzat les llistes, el tipus de les quals és `[a]` que són un tipus algebraic (amb els constructor `[]` i `:`) que a més és polimòrfic, ja que en l'expressió `[a]` apareix una variable de tipus: `a`.

Una funció és genèrica si el seu tipus conté variables de tipus. Per exemple, la funció que calcula la longitud d'una llista, que ja coneixeu, té una implementació que la defineix per a tots els possibles tipus de `a` (aquesta classe de polimorfisme es coneix com a polimorfisme paramètric):

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

D'altra banda, a voltes la definició universal per a tots els tipus `a` és massa àmplia. Per exemple, la funció de comparació de llistes `==` requerix que, al seu torn, els valors continguts en elles també es puguen comparar (a causa de l'expressió `x==y` de la següent definició que, com vorem, requerix afegir una restricció):

```
(==) :: [a] -> [a] -> Bool
[]     == []     = True
[]     == (x:xs) = False
(x:xs) == []     = False
(x:xs) == (y:ys) = x==y && xs==ys
```

Per a poder indicar la restricció de que el tipus `a` deu admetre la comparació, Haskell utilitza les *classes de tipus*.

Nota: En cara que s'usa la paraula *classe* en aquesta denominació, no s'ha de confondre les classes de tipus de Haskell amb el concepte de classe de la programació orientada a objectes. Els tipus no són objectes. Les classes de tipus agrupen un conjunt de tipus d'operació, de manera que si un tipus és una instància d'una classe de tipus, tenim la garantia de que té definides eixes operacions. Això és més paregut a les interficies de Java que a les seues classes. Un tipus pot ser instància de més d'una classe de tipus.

El sistema de classes de tipus de Haskell permet utilitzar la parametrització per a definir funcions sobrecarregades, imposant pertànyer a una classe als tipus sobre els quals s'aplica la funció. Per exemple, la classe de tipus `Eq` representa els tipus que tenen definides les funcions `==` i `/=`. El fet que `a` siga de la classe de tipus `Eq` es denota `Eq a` i es posa com una restricció "`(Eq a) =>`" a l'hora de definir el tipus de la funció `(==)` com seguix:

```
(==) :: (Eq a) => [a] -> [a] -> Bool
```

La restricció "`(Eq a) =>`" en la definició anterior fa que aquesta es llegisca: "per a tot tipus `a` que siga una instància de la classe de tipus `Eq`, la funció `(==)` té un tipus `[a] -> [a] -> Bool`". És a dir, que un tipus siga una instància d'una classe de tipus, és una garantia de que especifica les operacions que la classe de tipus indica.

Al llarg d'aquesta pràctica vorem exemples i realitzarem exercicis amb tipus algebraics i amb classes de tipus. Vorem tant exemples de polimorfisme paramètric com de polimorfisme *ad hoc* (també conegut com *sobrecàrrega*). Per a més informació pots consultar diversos materials, entre ells la següent pàgina:

`http://www.haskell.org/tutorial/classes.html`

En la definició de les classes de tipus de Haskell no existix la distinció de control d'accés als mètodes que apareixen en Java (`public`, `private`, etc.). En lloc d'açò, s'utilitza el sistema de mòduls que, com vas vore en la secció anterior, permet definir llistes d'elements a exportar i pot servir per a ocultar els detalls d'implementació.

El següent exemple mostra un mòdul on es definix una estructura de dades de tipus pila o `Stack` amb una sèrie de funcions per a crear una pila buida (`empty`), afegir i eliminar elements de la pila (`push` i `pop`), consultar el tope de la pila (`top`) i determinar si la pila està buida (`isEmpty`):

```
module Stack (Stack, empty, push, pop, top, isEmpty) where
    data Stack a       = EmptyStack | Stk a (Stack a)
    push x s           = Stk x s
    top (Stk x s)      = x
    pop (Stk _ s)      = s
    empty              = EmptyStack
    isEmpty EmptyStack = True
    isEmpty (Stk x s)  = False
```

Observe's que els mòduls que importen `Stack` no poden utilitzar els constructors dels valors del tipus `Stack` (ens referim a `EmptyStack` i a `Stk`), ja que no són visibles (no han sigut exportats). En el seu lloc, hem de crear piles mitjançant les funcions `empty`, `push` i `pop`. Cal comprovar què passa en intentar utilitzar un dels constructors. Per a açò, cal escriure el fitxer `testStak.hs` com segueix:

```
import Stack
main = putStrLn show(isEmpty (EmptyStack))
```

i intentar compilar-ho, amb el consegüent error de l'ús del constructor `EmptyStack`:

```
bash$ ghc --make testStack.hs
[1 of 2] Compiling Stack            ( Stack.hs, Stack.o )
[2 of 2] Compiling Main             ( testStack.hs, testStack.o )
testStack.hs:3:30: Not in scope: data constructor ‘EmptyStack’
```

Aquest altre exemple:

```
import Stack
main = putStrLn (show (top (push 5 empty)))
```

funciona sense problemes:

```
bash$ runghc testStack2.hs
5
```

És a dir, podem ocultar els detalls de l'estructura de dades i la definició de les funcions. Açò ens pot permetre canviar aquesta implementació sense afectar als que facen ús del `Stack`. Per exemple, podem redefinir la pila utilitzant una llista:

```
module Stack (Stack, empty, push, pop, top, isEmpty) where
    data Stack a          = Stk [a]
    empty                 = Stk []
    push   x (Stk xs)     = Stk (x:xs)
    pop      (Stk (x:xs)) = Stk xs
    top      (Stk (x:xs)) = x
    isEmpty  (Stk xs)     = null xs
```

Els mòduls que utilitzen la pila seguirien funcionant igual. En aquest cas, el tipus de dades algebraic utilitzat té un sol constructor: quan volem crear un tipus que és bàsicament igual que un altre (una llista) però no és un sinònim (ja que les funcions no les definim per a una llista), és preferible utilitzar `newtype` en lloc de `data`, però no anem a aprofundir en l'ús de `newtype` en aquesta pràctica.

Imaginem que ens interessa mostrar una pila (és a dir, mostrar-la mitjançant una cadena). Per a açò, la forma estàndard en Haskell consistix a fer que la pila siga una instància de la classe de tipus `Show`, la qual cosa garantiria que hi ha una funció de tipus:

```
show :: (Stack a) -> String
```

encara que, llevat que ens limitem a mostrar una cadena de tipus `"una pila"`, voldrem mostrar el contingut de la pila i, per a açò, seria necessari que el tipus `a` fóra també de la classe de tipus `Show`:

```
show :: (Show a) => (Stack a) -> String
```

Fer que `Stack` siga de la classe de tipus `Show` pot aconseguir-se de manera molt senzilla: n'hi ha prou amb afegir `deriving Show` en la declaració del tipus `Stack` (tornem a utilitzar la implementació inicial):

```
module Stack (Stack, empty, push, pop, top, isEmpty) where
    data Stack a = EmptyStack | Stk a (Stack a) deriving Show
    ...
```

El us de `deriving` està limitat a un conjunt limitat de classes de tipus estàndard (`Eq`, `Show`, `Ord`, `Enum`, `Bounded` y `Read`) i proporciona un comportament per defecte per a les funcions associades. En el cas de tipus algebraics, seria com mostra aquest exemple (fitxer `testStack3.hs`):

```
import Stack
main = putStrLn (show (push 7 (push 5 empty)))
```

que dóna aquest resultat (funciona perquè `Int` és de la classe de tipus `Show`):

```
bash$ runghc testStack3.hs
Stk 7 (Stk 5 EmptyStack)
```

Vegem la forma més general d'indicar que un tipus és una instància d'una classe de tipus i, al mateix temps, vegem com definir la funció `show` per al tipus `Stack`, per a fer açò cal afegir el següent al final del mòdul `Stack`:

```
instance (Show a) => Show (Stack a) where
   show EmptyStack = "|"
   show (Stk x y) = (show x) ++ " <- " ++ (show y)
```

Nota: Observe's que en la definició de la funció apareixen dues cridades a `show` però fixat que la primera empra la definició de `show` del tipus `a` mentres que la segona és una cridada *recursiva*.

Observe's també que el caràcter "`|`" indica el fons de la pila com mostra el següent exemple (fitxer `testStack4.hs`):

```
import Stack
main = do
  putStrLn (show (pop (push 1 empty)))
  putStrLn (show (push 10 (push 5 empty)))
```

que genera la sortida següent:

```
|
10 <- 5 <- |
```

Exercici 2 *Definix la funció operador* `==` *per al tipus* `Stack a` *que funcione per a tipus* `a` *que siguen de la classe de tipus* `Eq`*. La idea és que es podria realitzar fàcilment utilitzant* `deriving Eq` *però ho has de resoldre utilitzant* `instance`*.*

Exercici 3 *Defineix les funcions* `fromList` *i* `toList` *que convertixen un valor del tipus* `Stack a` *en una llista de tipus* `[a]` *amb els elements de la pila i viceversa. Per a açò, has d'importar el mòdul* `Stack` *i utilitzar les funcions que aquest exporta (sense recórrer als constructors dels valors de tipus).*

3.2 Polimorfisme ad hoc o sobrecàrrega

Per a definir una funció, el comportament de la qual depenga del tipus de valor rebut, no fa falta necessàriament recórrer a classes de tipus. El següent exemple mostra com es pot definir un tipus de figura geomètrica `Shape` que definix dos tipus de figura, de manera que el càlcul de l'àrea es definix segons el tipus:

```
type Height = Float
type Width  = Float
type Radius = Float
data Shape  = Rectangle Height Width |
              Circle Radius
              deriving (Eq, Show)
area :: Shape -> Float
area (Rectangle h w) = h * w
area (Circle    r)   = pi * r**2
```

Un problema d'aquesta forma de treballar és que no resulta possible afegir dinàmicament més constructors per al tipus `Shape`.

La forma de solucionar-ho és definir una *classe de tipus* `Shape` i després tantes instàncies d'ella com a figures concretes vulguem crear, per exemple `Rectangle` i `Circle`. Observe's que després podrem definir termes dels tipus de dades `Rectangle` i `Circle`, per la qual cosa tindrem una classe de tipus, dues instàncies i successius termes d'aquestes instàncies. La definició usant classes de tipus és la següent:

```
type Height = Float
type Width  = Float
type Radius = Float
data Rectangle = Rectangle Height Width
data Circle = Circle Radius

class Shape a where
  area :: a -> Float

instance Shape Rectangle where
  area (Rectangle w h) = w * h

instance Shape Circle where
  area (Circle r) = pi * r**2

type Volume = Float
volumePrism :: (Shape a) => a -> Height -> Volume
volumePrism base height = (area base) * height
```

La funció `volumePrism` del final és capaç d'utilitzar un element de la classe de tipus `Shape a`, en concret termes dels tipus `Rectangle` i `Circle` (instàncies de `Shape a`) i invocar a la funció `area`, que executarà una de les dues funcions `area` depenent del tipus. Direm que la funció `area` té polimorfisme adhoc.

Exercici 4 *Cal fer que la classe de tipus* `Shape` *tinga també una funció* `perimeter` *que retorne el perímetre d'una figura. Per això, modifica adequadament la instanciació de* `Rectangle` *i de* `Circle`*.*

Exercici 5 *Afegeix la mateixa funció* `perimeter` *de l'exercici anterior a la definició de figures basada en tipus algebraics vista una mica més amunt. És a dir, a la definició que inclou*

```
data Shape  = Rectangle Height Width |
              Circle Radius
              deriving (Eq, Show)
```

Exercici 6 *Definix una funció* `surfacePrism` *que calcule la superfície d'un prisma.*

Exercici 7 *Modifica la definició basada en classes de tipus perquè siga possible mostrar i comparar mitjançant la igualtat els valors de la classe de tipus* `Shape` *instanciant les classes de tipus* `Show` *i* `Eq`*. La idea consisteix bàsicament en canviar la linia:*

```
class Shape a where
```

per

```
class (Eq a, Show a) => Shape a where
```

i després incloure el codi necessari per tal que compile i funcione correctament.

4 Avaluació

L'assistència a les sessions de pràctiques és obligatòria per a aprovar l'assignatura. A banda, més endavant es farà un examen de pràctiques individual en el laboratori.