Sustitución, funciones de orden superior e introducción a mónadas
Lógica Computacional 2017-2

Lourdes del Carmen González Huesca
Roberto Monroy Argumedo
Fernando A. Galicia Mendoza

Facultad de ciencias,UNAM

Miércoles,8 de marzo del 2016

Sustitución

En el primer tercio de la clase nos dedicaremos a implementar la sustitución en la lógica de primer orden.

La función de orden superior mas intutiva

```
map :: (a → b) → [a] → [b]
map _ [] = []
map f (x:xs) = (f x:map f xs)
```

Las funciones `fold`

Problema:

Crear una función f que reciba un operador binario $\circ$, un elemento x de tipo `a` y una lista ℓ de tipo `a`, tal que:

$$pliegue(\circ, x, \ell) = (\ldots((x \circ x_1) \circ x_2) \circ \cdots \circ x_n)$$

Las funciones fold

Problema:

Crear una función f que reciba un operador binario $\circ$, un elemento x de tipo a y una lista ℓ de tipo a, tal que:

$$pliegue(\circ, x, \ell) = (\ldots((x \circ x_1) \circ x_2) \circ \cdots \circ x_n)$$

Solución:

```
--plIzq = Pliegue Izquierdo
plIzq :: (b → a → b) → b → [a] → b
plIzq f z [] = z
plIzq f z (x:xs) = plIzq f (f z x) xs
```

Las funciones `fold`

Problema:

Crear una función f que reciba un operador binario $\circ$, un elemento x de tipo a y una lista ℓ de tipo a, tal que:

$$pliegue(\circ, x, \ell) = (\ldots((x \circ x_1) \circ x_2) \circ \cdots \circ x_n)$$

Solución:

```
--plIzq = Pliegue Izquierdo
plIzq :: (b → a → b) → b → [a] → b
plIzq f z [] = z
plIzq f z (x:xs) = plIzq f (f z x) xs
```

Problema 2: Definir el pliegue derecho.

Problemas de funciones de orden superior:

1. Define una función que devuelva el máximo elemento, según indique la clase `Ord`, de una lista.
2. Utilizando la función `foldl` da una construcción alternativa a `length`.
3. Utilizando la función `foldl` da una construcción alternativa a `map`.
4. Utilizando la función `foldl` da una construcción alternativa a `concat`.

Seamos monjes sabios

En cualquier lenguaje de programación imperativo (`Java`,`Python`,`C`,etc.) define un programa que reciba un archivo de texto y convierta todas las letras minúsculas a mayúsculas.

Seamos monjes sabios

En cualquier lenguaje de programación imperativo (Java,Python,C,etc.) define un programa que reciba un archivo de texto y convierta todas las letras minúsculas a mayúsculas.

En Haskell el programa que hace eso es el siguiente:

```
import Data.Char
main = do
     inpStr ←readFile "input.txt"
     writeFile "output.txt" (map toUpper inpStr)
```

Simple pero veamos el fondo de este programa.

Notación do

La notación do esta implementada de la siguiente forma:

```
do {v} = v
do {x ← m;p} = m »= λx → do {p}
do {m1;m2} = m1  »  m2 do {m2}
do {let {x1=y1;...;xn=yn};p} = let {x1=y1;...;xn=yn} in do p
```

Analicemos esta notación brindada por el lenguaje.

Un error usual…¿O excepción usual?

Es usual que cuando se inicia a estudiar errores en un lenguaje imperativo, se de el ejemplo de división entre cero. En `Haskell` la función que divide dos números enteros es la siguiente:

```
divide :: Int → Int → Float
divide _ 0 = error "Division entre cero."
divide n m = n/m
```

Recordando que la expresión `error` termina el programa, para evitar esto podemos definir la función encapsulando el resultado correcto:

Un error usual. . . ¿O excepción usual?

Es usual que cuando se inicia a estudiar errores en un lenguaje imperativo, se de el ejemplo de división entre cero. En `Haskell` la función que divide dos números enteros es la siguiente:

```
divide :: Int → Int → Float
divide _ 0 = error "Division entre cero."
divide n m = n/m
```

Recordando que la expresión `error` termina el programa, para evitar esto podemos definir la función encapsulando el resultado correcto:

```
divide :: Int → Int → Maybe Float
divide _ 0 = Nothing
divide n m = Just (n/m)
```

Maybe la monada introductoria ideal

Maybe está implementada de la siguiente forma:

```
instance Monad Maybe where
return = Just
x >>= f = case x of
              Nothing → Nothing
              Just x → f x
```

Tratemos de inferir las firmas y especificaciones de los métodos `return` y >>=.

Maybe la monada introductoria ideal

Maybe está implementada de la siguiente forma:

```
instance Monad Maybe where
return = Just
x >>= f = case x of
                Nothing → Nothing
                Just x → f x
```

Tratemos de inferir las firmas y especificaciones de los métodos `return` y >>=. Observamos que `Maybe` es instancia de la clase `Monad`, cuya implementación de esta última es la siguiente:

```
class Monad m where
   return :: a → m a
   (>>=) :: m a → (a → m b) → m b
```

Un par de hermosos monstruos

Observemos que `return` la moneria que hace es encapsular un tipo dado en una mónada y bind dado un tipo `a` encapsulado y una función f que va del tipo `a` al tipo `b` encapsulado, devuelve la aplicación de f a cada elemento del tipo `a` encapsulado.
Esto claramente recuerda a ...

Un par de hermosos monstruos

Observemos que `return` la moneria que hace es encapsular un tipo dado en una mónada y bind dado un tipo `a` encapsulado y una función f que va del tipo `a` al tipo `b` encapsulado, devuelve la aplicación de f a cada elemento del tipo `a` encapsulado.
Esto claramente recuerda a . . . la estructura de listas. La cual resulta también ser una monada, veamos su implementación.

```
instance Monad List where
    return x = Cons x Nil
    xs >>= k = join (map k xs)

join :: List (List a) → List a
join Nil = Nil
join (Cons xs xss) =  cat xs (join xss)

cat :: List a → List a → List a
cat Nil ys = ys
cat (Cons x xs) ys = Cons x (cat xs ys)
```

Expliquen al ayudante que hacen las funciones `join` y `cat`.

¿Qué necesito para que mi estructura sea una monada?

Como hemos visto las clases son abstracciones cuyos métodos deben de cumplir los tipos que las instancien y a parte cada clase cuenta con cierta axiomatización para analizar sus funciones.
Por parte de las mónadas se deben cumplir los siguiente axiomas, propuestos por Kleisli:

```
x >>= return = x
(return x) >>= f = f x
(x >>= f) >>= g = λz → (f z >>= g)
```

Para mayor información de esta axiomatización revisar el trabajo *Mónadas en la programación funcional: Una prueba formal de su equivalencia con las ternas de Kleisli* de C. Moisés Vázquez Reyes.

Exepciones

Veamos por último la simulación de excepciones utilizando `Maybe`.

```
divide :: Int → Int → Maybe Float
divide _ 0 = Nothing
divide n m = Just (fromIntegral n / fromIntegral m)

res :: Maybe Float → String
res Nothing = "Execpcion: Division entre cero."
res (Just x) = "Respuesta "++show x

main = do
  putStrLn "Ingresa un numero: ";
  x ← readLn;
  putStrLn "Ingresa otro numero: ";
  y ← readLn;
  putStrLn (res (divide x y));
  --Aqui no se detuvo el computo
  putStrLn "Sigo vivo :D"
```