Estructuras Algebraicas

Programación Funcional, Universidad Nacional de Quilmes

24 de junio de 2018

Aclaraciones:

- *Los ejercicios fueron pensados para ser resueltos en el orden en que son presentados. No se saltee ejercicios sin consultar antes a un docente.*
- *Recuerde que puede aprovechar en todo momento las funciones que ha definido, tanto las de esta misma práctica como las de prácticas anteriores.*
- *Pruebe todas sus implementaciones, al menos en una consola interactiva.*
- *Es sumamente aconsejable resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en clase, dado que los exámenes de la materia evaluación principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*

1. Monoides

La clase monoide está definida como:

```
class Semigroup a where
   (<>) :: a -> a -> a

class Semigroup a => Monoid a where
   mempty :: a
   mappend :: a -> a -> a
   mconcat :: [a] -> a
   mconcat = foldr mappend mempty
```

Donde se cumple que `mempty` es el neutro de `mappend`, y además `mappend` es asociativa (técnicamente `<>` y `mappend` son sinónimos).

1. Definir instancias de `Monoid` para los siguientes tipos de datos:

   ```
   newtype Sum = Sum Int
   newtype Prod = Prod Int
   newtype All = All Bool
   newtype Any = Any Bool
   newtype Endo a = Endo (a -> a)
   [a]
   ```

2. Resolver las siguientes funciones utilizando esta typeclass:

   a) `sum, product :: [Int] -> Int`

   b) `and, or :: [Bool] -> Bool`

   c) `concat :: [[a]] -> [a]`

   d) `id :: a -> a`

   e) `twice :: (a -> a) -> a -> a`

f) `concatMap :: (a -> [b]) -> [a] -> [b]`

g) `applyN :: Int -> (a -> a) -> a -> a`

h) `ntimes :: Monoid a => Int -> a -> a`

i) `factorial :: Int -> Int`

j) `sumatoria :: Int -> Int`

k) `tconcat :: Monoid a => Tree a -> a`

l) `sumT :: Tree Int -> Int`

m) `anyT :: (a -> Bool) -> Tree a -> Bool`

n) `allT :: (a -> Bool) -> Tree a -> Bool`

$\tilde{n}$) `concatT :: Tree [a] -> [a]`

2. Functors

2.1. Ejemplos

Indicar el resultado de las siguientes expresiones:

1. fmap (replicate 3) [1,2,3,4]
2. fmap (replicate 3) (Just 4)
3. fmap (replicate 3) (Right "hola")
4. fmap (replicate 3) Nothing
5. fmap (replicate 3) (Left "foo")

3. Maybe Monad

Dadas las siguientes definiciones

```
tailM :: [a] -> Maybe [a]
initM :: [a] -> Maybe [a]

f :: Maybe [a]
f = case tailM [1,2,3,4,5] of
    Nothing -> Nothing
    Just xs1 -> case initM xs1 of
        Nothing -> Nothing
        Just xs2 -> case tailM xs2 of
            Nothing -> Nothing
            Just xs3 -> initM xs3
```

dar una definición monádica para `f`

4. List Monad

La definición de `Monad` de listas es:

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
```

1. Indicar el resultado de las siguientes expresiones:
   - `[3,4,5] >>= \x -> [x,-x]`
   - `[] >>= \x -> ["bad","mad","rad"]`
   - `[1,2,3] >>= \x -> []`
   - `[1,2,3] >>= (\x -> [x,x]) >>= (\y -> return [y,y*y])`
2. Definir las siguientes funciones utilizando la mónada de listas
   - `productoCartesiano :: [a] -> [b] -> [(a, b)]`
   - ```
     data Resultado = Cara | Seca
     tiradas :: Int -> [[Resultado]]
     ```
   - (Desafío) `powerset :: [a] -> [[a]]`
     Devuelve todas las sublistas (pensar en subconjuntos) posibles.
     Se recomienda usar `filterM` (ver Hoogle)

5. Reader Monad

5.1. Functor

Dada la siguiente instancia de `Functor`

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
    -- Equivalente a fmap = (.)
```

Indicar el resultado de las siguientes expresiones:

- fmap (*3) (+100) 1
- fmap (show . (*3)) (*100) 1
- fmap (replicate 3) (const 3) 3
- fmap (replicate 3) (+3) 3
- fmap (const 3) (const 3) 3

5.2. Expresiones Aritméticas con variables

Dada la siguiente representación de expresiones aritméticas

```
data Env = [(String, Int)]

data Exp = Sum Exp Exp | Var String | Const Int

getValue :: String -> Env -> Int
getValue s e =
   case lookup s e of
        Nothing -> 0
        Just v  -> v
```

completar la definición de `eval :: Exp -> Reader Env Int`
```

5.3. Propiedades

Demuestre que para la instancia `Functor` de `(->) r` las siguientes propiedades

- fmap id = id
- fmap (f . g) = fmap f . fmap g

6. Writer

1. Definir la siguiente función que dado un número se queda con los elementos que son mayores a éste, informando qué elementos son agregados al resultado:

   ```
   mayoresA :: Int -> [Int] -> Writer [String] [Int]
   ```

2. Dada la siguiente definición para obtener el Máximo Común Dividor

   ```
   gcd' :: Int -> Int -> Int
   gcd' a b = if b == 0 then a else gcd' b (a `mod` b)
   ```

   transformarla a una versión monádica que indique el valor de los parámetros en cada momento

7. State Monad

7.1. Stack

Dada la siguiente definición de una `Stack`

```
type Stack = [Int]

pop :: Stack -> (Int,Stack)
pop (x:xs) = (x,xs)

push :: Int -> Stack -> Stack
push a xs = a:xs
```

1. Transformar dicha interfaz a una versión monádica con estado
2. implementar con esa interfaz las siguientes operaciones
   - `dropN :: Int -> State Stack ()`
   - `takeN :: Int -> State Stack Stack`

7.2. Gobstones

1. Dar una implementación de las operaciones de `Gobstones` donde el estado sea un tablero y los comandos tengan tipo `State Tablero ()`.

   ```
   data Color = Azul | Negro
   data Dir = Izq | Der
   data Celda = C Int Int

   data Tablero = T [(Int,Celda)] Int
   --               fila          cabezal
   ```

- `tableroVacio :: Int -> Int -> Tablero`
- `poner :: Color -> State Tablero ()`
- `mover :: Dir -> State Tablero ()`
- `nroBolitas :: Color -> Tablero -> Int`
- `puedeMover :: Dir -> Tablero -> Bool`

2. Escribir el siguiente programa con dicha implementación, que se ejecute sobre un tablero inicial vacío.

```
program {
    PonerN(10, Negro)
    PonerN(5, Azul)
    MoverN(3, Der)
    if (puedeMover(Der)) { Mover(Der) }
}
```

8. Funciones Genéricas

Definir las siguientes funciones estándar sobre mónadas en general (ver ejemplos en Hoogle):

- `void :: Monad m => m a -> m ()`
- `when :: Monad f => Bool -> f () -> f ()`
- `liftM :: Monad m => (a -> b) -> m a -> m b`
- `liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c`
- `appM :: (Monad m) => m (a -> b) -> m a -> m b`
- `join :: (Monad m) => m (m a) -> m a`
- `sequence :: Monad m => [m a] -> m [a]`
- `sequence_ :: Monad m => [m a] -> m ()`
- `mapM :: Monad m => (a -> m b) -> [a] -> m [b]`
- `mapM_ :: Monad m => (a -> m b) -> [a] -> m ()`
- `filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]`
- `forM :: (Monad m) => [a] -> (a -> m b) -> m [b]`
- `(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c`
- `forever :: Monad m => m a -> m b`
- `zipWithM :: Applicative m => (a -> b -> m c) -> [a] -> [b] -> m [c]`
- `foldM :: Monad m => (b -> a -> m b) -> b -> [a] -> m b`
- `foldM_ :: (Monad m) => (b -> a -> m b) -> b -> [a] -> m ()`
- `replicateM :: Monad m => Int -> m a -> m [a]`
- `replicateM_ :: Monad m => Int -> m a -> m ()`