CI4251 - Programación Funcional Avanzada
Tarea 2

Ernesto Hernández-Novich
86-17791
<emhn@usb.ve>

Mayo 12, 2016

Yo dawg! I heard you liked functors...

Declaración de instancias de `Functor`, `Applicative` y `Monad` para el tipo `Functor f`.

Para `Functor` es necesario implementar `fmap`. Para el constructor `Bonus` es necesario hacer `fmap` sobre el functor interno y luego `fmap` sobre el `Bonus` interno.

```
instance Functor f => Functor (Bonus f) where
  fmap g (Malus a)  = Malus (g a)
  fmap g (Bonus bs) = Bonus (fmap (fmap g) bs)
```

El `Applicative` pide la implentación de `pure` y del operador `<*>`. Si el primer elemento es un Malus simplemente mappeamos su función sobre el segundo elemento. Si el segundo elemento es un Malus mappeamos la evaluacion de su elemento sobre todo el primer elemento. Finalmente si ambos elementos son Bonus, mappeamos `b1 <*>` sobre b2.

```
instance Functor f => Applicative (Bonus f) where
  pure = Malus
  (Malus f)    <*>     x       = fmap f x
  bs@(Bonus _) <*> (Malus a)  = fmap (flip ($) a) bs
  bs1          <*> (Bonus f)  = Bonus (fmap (bs1 <*>) f)
```

Para el `Monad` es necesario `return` y el `bind`. Si el primer elemento del bind es un `Malus` simplemente sacamos su valor y lo pasamos a la función. Si es un `Bonus`, mappeamos el bind dentro de su functor.

```
instance Functor f => Monad (Bonus f) where
  return = Malus
  (Malus x)  >>= f = f x
  (Bonus bs) >>= f = Bonus (fmap (flip (>>=) f) bs)
```

Can I has pizza?

```
data Want a = Want ((a -> Pizza) -> Pizza)

data Pizza = Eat (IO Pizza)
           | Combo Pizza Pizza
           | Happy
```

```
instance Show Pizza where
   show (Eat x)     = " :-P "
   show (Combo x y) = " combo(" ++ show x
                                ++ ","
                                ++ show y ++ ") "
   show Happy       = " :-D "
```

Al `Want` hay que pasarle una función que devuelva una Pizza. El único constructor que no necesita elementos es `Happy`.

```
want :: Want a -> Pizza
want (Want f) = f (\_ -> Happy)
```

`happy` simplemente devuelve un `Want` que solo devuelve un `Happy`.

```
happy :: Want a
happy = Want (\_ -> Happy)
```

`nomnom` recibe una acción de `IO`. Se devuelve un `Want` que cuando reciba su función `(a ->Pizza)` liftea esa función a la acción monádica para que pase de ser `IO a` a `IO Pizza` y pasarlo al constructor `Eat`.

```
nomnom :: IO a -> Want a
nomnom ioAct = Want (\f -> Eat (liftM f ioAct))
```

`combo` recibe un `Want` y devuelve otro que espera una función de tipo `() ->Pizza` y cuando la recibe devuelve un `Combo`.

```
combo :: Want a -> Want ()
combo w = Want (\f -> Combo (want w) (f ()))
```

`pana` combina dos `Want`s en uno solo. Para ello encierra las funciones de los dos `Want` evaluadas con la funcion que recibirá el resultado y los junta con un `Combo`.

```
pana :: Want a -> Want a -> Want a
pana (Want f1) (Want f2) = Want (\x -> Combo (f1 x) (f2 x))
```

`pizzeria` funciona con recursion de cola sobre la lista que recibe. Por cada Pizza que lee revisa su constructor. Si es un `Eat`, ejecuta su acción monádica y el resultado (que es una `Pizza`) lo pone al final de la lista. Si es un `Combo` simplemente agrega sus dos `Pizzas` al final de la cola. Si es un `Happy` no hace nada.

```
pizzeria :: [Pizza] -> IO ()
pizzeria []     = return ()
pizzeria (p:ps) = do case p of
                       Eat f -> do piz <- f
                                   pizzeria (ps ++ [piz])
                       Combo p1 p2 -> pizzeria (ps ++ [p1,p2])
                       Happy -> return ()
```

El `bind` del `Monad` devuelve un Want que incluye el anterior. Al `Want` inicial le pasa una función que evalúa el `a` que recibe con la función `g`. El resultado de esto es otro `Want`, al cual le extraemos su función interna y esa la evaluaríamos con el argumento que espera el `Want` final.

```
instance Monad Want where
  return x       = Want (\f -> f x)
  (Want w) >>= g = Want (\b -> w (\a -> (exctract (g a)) b))
      where exctract (Want a) = a
```