PRÁCTICA 8

INTRODUCCIÓN WINHUGS

Universidad de Huelva

8.1. REPASO

8.1. Repaso

Entrada y Salida

Monadas

Operador $

when | unless | sequence | mapM | mapM_

forever | interact

openFile | hGetContets | hClose | withFile

hGetChar | hGetLine | hPutChar | hPutStr |

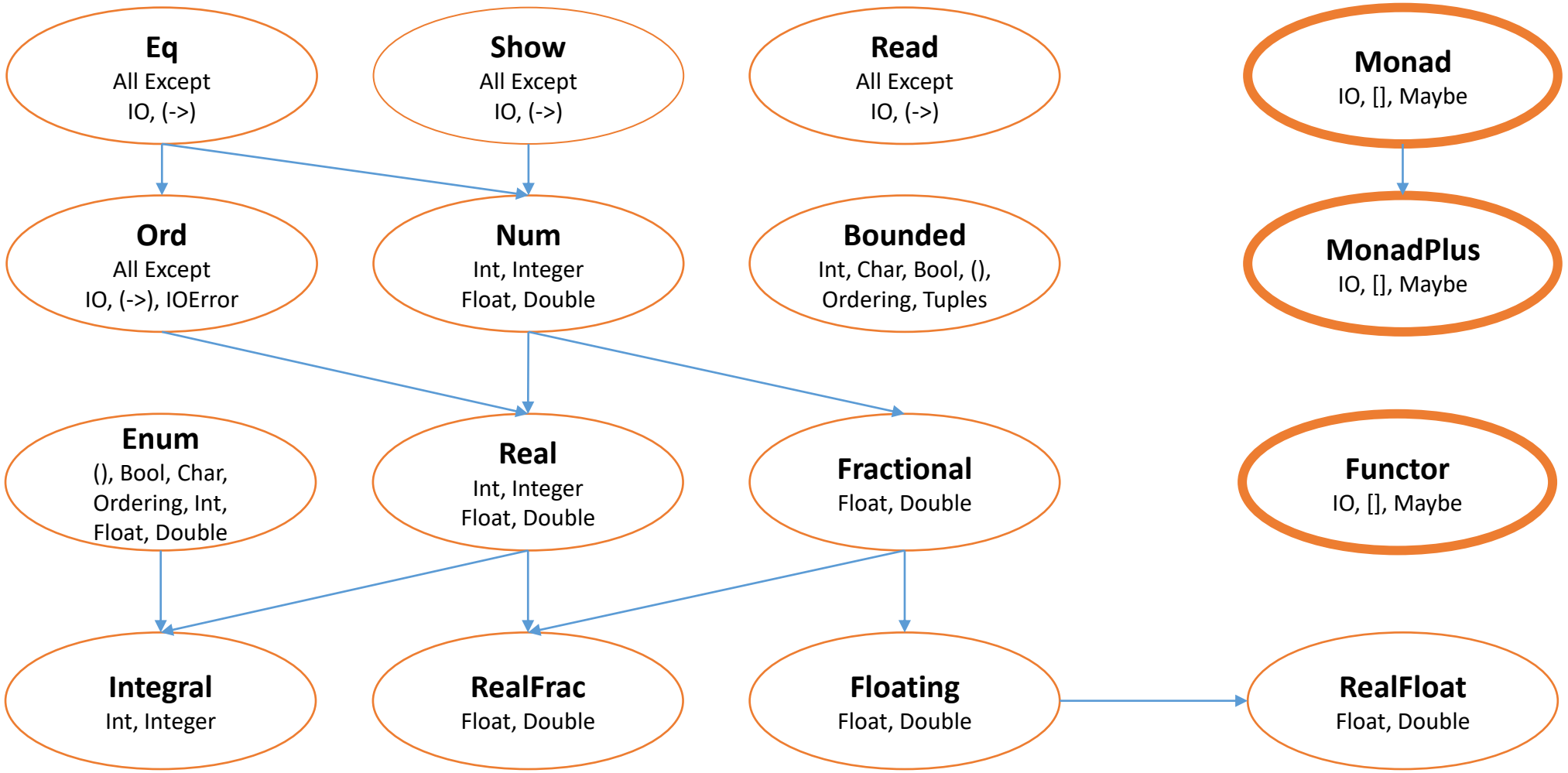
readFile | writeFile | appendFile

Ejercicios

```haskell
name <- getLine
putStrLn $ "Encantado de conocerte, " ++ name ++ "!"
```

```haskell
when_con_when = do
    c <- getChar
    when (c /= ' ') $ do
        putChar c
        when_con_when
```

7.2.1. Funciones sobre acciones



8.1. Repaso

Functors

Es algún tipo de **contenedor** y una **función asociada fmap** que le permite alterar lo que esté contenido, dada una función que transforma el contenido.

Las listas son este tipo de contenedor,

```
Hugs> map (+1) [1,2,3,4]
[2,3,4,5] :: [Integer]
```

Maybe también se puede convertir en un functor

```
Hugs> fmap  (+5) (Just 5)
Just 10 :: Maybe Integer
```

El tipo general de fmap muestra claramente lo que está sucediendo

8.1. Repaso

Functors

Veamos qué está sucediendo:

```
Hugs> :type fmap
fmap :: Functor a => (b -> c) -> a b -> a c
```

Tipos lista

```
Hugs> fmap :: (a -> b) -> [a] -> [b]
```

Tipos Maybe

```
Hugs> fmap :: (a -> b) -> Maybe a -> Maybe b
```

8.1. Repaso

Functors

fmap es por tanto, una **generalización** de map:

```
map (subtract 1) [2,4,8,16] = [1,3,7,15]
--     Int->Int      [Int]         [Int]

fmap (* 10)  [2,4,8,16] = [20,40,80,160]
--   Int->Int    [Int]         [Int]
```

Los functors también funcionan con IO, lo cual map no haría y daría error

```
Hugs> fmap ("Hello, " ++)  getLine  >>= putStrLn
Antonio
Hello, Antonio
 :: IO ()
```

8.2. DEFINICION DE TIPOS

8.2.1. Definición de tipos

Declaración type

La instrucción *type* permite asignar un alias a una declaración de tipo

La sintaxis es la siguiente:

typeDecl ::= type *simpleType* = *expresion_de_tipo*

El símbolo ***simpleType*** define el identificador del tipo y puede incluir variables de tipo.

simpleType ::= Identificador $tvar_1$ $tvar_2$ ... $tvar_n$

Las variables incluidas en *simpleType* deben corresponder a las variables de tipo incluidas en la expresión.

8.2.1. Definición de tipos

Declaraciones de tipos como sinónimos

Se puede definir un nuevo nombre para un tipo existente mediante una declaración de tipo.

Ejemplo: Las cadenas son listas de caracteres.

El nombre del tipo tiene que empezar por mayúscula

```
type String = [Char]
words :: String -> [String]
```

8.2.1. Definición de tipos

Declaraciones de tipos como sinónimos

```haskell
libretaDirecciones :: [(String,String)]
libretaDirecciones =
    [("Antonio","657123456"), ("Jesus","657123457"), ("Albaro","657123458")]

--Sinonimos
type Registro = [(Nombre,Telefono)]
type Telefono = String
type Nombre = String

existeRegistro :: Nombre -> Telefono -> Registro -> Bool
existeRegistro nombre telefono libreta = (nombre,telefono) `elem` libreta
```

```
Main> existeRegistro "Antonio" "657123456" libretaDirecciones
True :: Bool
```

8.2.1. Definición de tipos

Declaraciones de tipos nuevos

Las declaraciones de tipos pueden usarse para facilitar la lectura de tipos.

Las posiciones son pares de enteros:

```
type Pos = (Int,Int)
```

origen es la posición (0,0)

```
origen :: Pos
origen = (0,0)
```

(izquierda p) es la posición a la izquierda de la posición p.

```
izquierda :: Pos -> Pos
izquierda (x,y) = (x-1,y)
```

8.2.1. Definición de tipos

Declaraciones de data

La forma convencional de definir un nuevo tipo de dato (lo que se conoce como datos algebraicos) es por medio de la instrucción data.

```
dataDecl   ::= data [context =>] simpleType [= constrs] [deriving]
simpleType  ::=   Identificador   tvar1  tvar2 ... tvarn
```

El contexto es opcional y permite indicar las clases a las que pertenecen los variables de tipo.

8.2.1. Definición de tipos

Declaraciones de data

La declaración de tipos de datos algebraicos se basa en definir constructores del tipo.

Por ejemplo, el tipo de dato *Bool* está basado en dos constructores: *True* y *False*. Los constructores de tipo utilizan identificadores que comienzan en mayúscula.

```
data Bool = False
          | True
```

Los constructores sin argumentos permiten definir fácilmente enumeraciones:

```
data Dias = Lunes | Martes | Miercoles | Jueves | Viernes | Sabado | Domingo
```

8.2.1. Definición de tipos

Declaraciones de data

```
data  Dias =  Lunes | Martes | Miercoles | Jueves | Viernes | Sabado | Domingo
deriving (Show)
```

```
Main> :reload
Main> Miercoles
Miercoles :: Dias
```

8.2.1. Definición de tipos

Declaraciones de data

```
data Dias = Lunes | Martes | Miercoles | Jueves | Viernes | Sabado | Domingo
deriving (Eq,Show)
```

```
Main> :reload
Main> Miercoles == Miercoles
ERROR - Cannot infer instance
*** Instance   : Eq Dias
*** Expression : Miercoles == Miercoles

Main> :reload
Main> Miercoles == Miercoles
True :: Bool
```

8.2.1. Definición de tipos

Declaraciones de data

```
data Dias = Lunes | Martes | Miercoles | Jueves | Viernes | Sabado | Domingo
```

```
Main> Miercoles
ERROR - Cannot find "show" function for:
*** Expression : Miercoles
*** Of type    : Dias
```

¿Cómo lo arreglamos?

8.2.1. Definición de tipos

Declaraciones de data

Los constructores pueden tener argumentos. No es necesario que todos los constructores de un tipo tengan los mismos argumentos.

Por ejemplo: (Shape) que pueda tomar dos formas: un círculo o un rectángulo.

El círculo necesita la posición del centro y el radio

El rectángulo necesita la posición de una esquina y su contraria

```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
```

Los constructores de tipo son funciones que toman los parámetros como entrada y generan un valor del tipo descrito

8.2.1. Definición de tipos

Declaraciones de data

Podemos utilizar los constructores para definir procesos de emparejamiento (pattern matching).

```
data  Shape  =  Circle  Float  Float  Float |  Rectangle  Float  Float  Float  Float

surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs (x2 - x1)) * (abs (y2 - y1))
```

```
Main> surface (Circle 5 3 2)
12.56637 :: Float
Main> surface (Rectangle 5 6 4 5)
1.0 :: Float
```

8.2.1. Definición de tipos

Declaraciones de data

El constructor de un tipo puede tener el mismo identificador que el tipo . Por ejemplo, podemos definir un punto como un tipo *Point* y utilizarlo en las definiciones de figuras.

```
surface2 :: Shape -> Float
surface2 (Circle _ r) = pi * r ^ 2
surface2 (Rectangle (Point x1 y1) (Point x2 y2)) =  (abs (x2 - x1)) * (abs (y2 - y1))
```

```
data  Point  =  Point  Float  Float
```

```
Main> surface2 (Circle (Point 5 2) 2)
12.56637 :: Float
Main> surface2 (Rectangle (Point 5 6) (Point 4 5))
1.0 :: Float
```

8.2.1. Definición de tipos

Declaraciones de data

Los tipos de datos algebraicos permiten definir estructuras de tipo registro. Por ejemplo, podríamos definir el tipo Persona que almacenara el nombre, apellidos y edad. Para acceder a estos campos podríamos definir funciones basadas en **patrones**.

```
Main> nombre (usuario1)
"Antonio" :: [Char]
Main> apellidos (usuario1)
"Palanco Salguero" :: [Char]
Main> edad (usuario1)
43 :: Int
```

```
data   Persona = Persona String  String  Int
nombre:: Persona -> String
nombre  (Persona n  _  _ ) =  n
apellidos :: Persona -> String
apellidos (Persona _  a  _) =  a
edad :: Persona -> Int
edad (Persona  _  _  e) = e

usuario1 = Persona "Antonio" "Palanco Salguero" 43
```

8.2.1. Definición de tipos

Declaraciones de data

Existe una sintaxis especial que permite definir las estructuras de tipo registro de una forma mucho más compacta. Este formato define directamente las funciones asociadas a los campos y permite indicar el valor de cada campo al construir el dato. El orden de los campos puede incluso ser diferente al de la definición del tipo.

```
data Persona = Persona { nombre :: String, apellidos :: String, edad :: Int }
usuario2 = Persona { nombre = "Juan" , apellidos = "Nadie" , edad = 50 }
```

```
Main> nombre(usuario2)
"Juan" :: [Char]
```

8.2.1. Definición de tipos

Declaraciones de data

Las definiciones de tipos pueden ser recursivas, es decir, las expresiones de tipos de los constructores pueden utilizar el mismo tipo de dato que estamos definiendo.

Por ejemplo, podemos definir un tipo de dato árbol con dos tipos de nodos: un nodo hoja que almacene un valor entero y un nodo interno que almacene dos ramas.

```haskell
data Arbol  =  Hoja  Int
            |  Nudo  Arbol  Arbol
           deriving (Eq, Show)

arbol = Nudo (Nudo (Hoja 1) (Hoja 2)) (Hoja 3)
```

```
Main> arbol
Nudo (Nudo (Hoja 1) (Hoja 2)) (Hoja 3) :: Arbol
```

8.2.1. Definición de tipos

Ejemplos

Crear el tipo RGB que define un color: ([0..255], [0..255],[0..255])

Definir colores: red, green y blue.

Crear colores mezclados:

```
data RGBColor = RGBColor Int Int Int deriving (Show)

red     = RGBColor 255 0 0
green   = RGBColor 0 255 0
blue    = RGBColor 0 0 255

mixColor (RGBColor r1 g1 b1) (RGBColor r2 g2 b2) = RGBColor (r1+r2) (g1+g2) (b1+b2)
```

8.3. PRACTICAR

8.3.1. Ejercicios

Ejercicios

- Realizar un ejemplo de definición de tipos (como por ejemplo los datos de un préstamo de materiales: id, nombre, fk_usuario, fecha_salida, fecha_entrada, observaciones). Crear las instancias de al menos 3 prestamos y mostrar por pantalla uno de ellos.

- Definir una lista de personas (al menos 5) con los siguientes campos: (Nombre,Apellidos,Edad). La lista debe definirse de forma ordenada por edad, de menor a mayor. Una ve hecho, realizar las siguientes funciones:
  - Obtener el segmento más largo de la lista con las personas de edad inferior a la dada
  - Buscar una persona por nombre.

8.3.1. Ejercicios

Ejercicios

```haskell
data Persona = Persona { nombre :: String, apellidos :: String, edad :: Int } deriving (Show,Eq)
usuario1 = Persona { nombre = "n1" , apellidos = "a1" , edad = 39 }
usuario2 = Persona { nombre = "n2" , apellidos = "a2" , edad = 40 }
usuario3 = Persona { nombre = "n3" , apellidos = "a3" , edad = 55 }
usuario4 = Persona { nombre = "n4" , apellidos = "a4" , edad = 60 }
usuario5 = Persona { nombre = "n5" , apellidos = "a5" , edad = 65 }

personas = [usuario1, usuario2,usuario3,usuario4,usuario5]
```

8.3.1. Ejercicios

Ejercicios

```haskell
takePersonas :: [Persona] -> Int ->[Persona]
takePersonas [] f = []
takePersonas (p:ps) f | esMenor p f = p : (takePersonas ps f)
                      | otherwise = []

esMenor :: Persona -> Int -> Bool
esMenor (Persona n a e) e1 =  e < e1
```

8.3.1. Ejercicios

Ejercicios

```haskell
existe :: [Persona] -> String -> [Persona]
existe [] p1 = []
existe (p:ps) p1  = do
                    {
                        if esta p p1 then [p]
                        else existe ps p1
                    }

esta :: Persona -> String -> Bool
esta (Persona n a e) e1 = n == e1
```