

PRÁCTICA 7

INTRODUCCIÓN WINHUGS



Universidad de Huelva

7.1. REPASO

Entrada y Salida

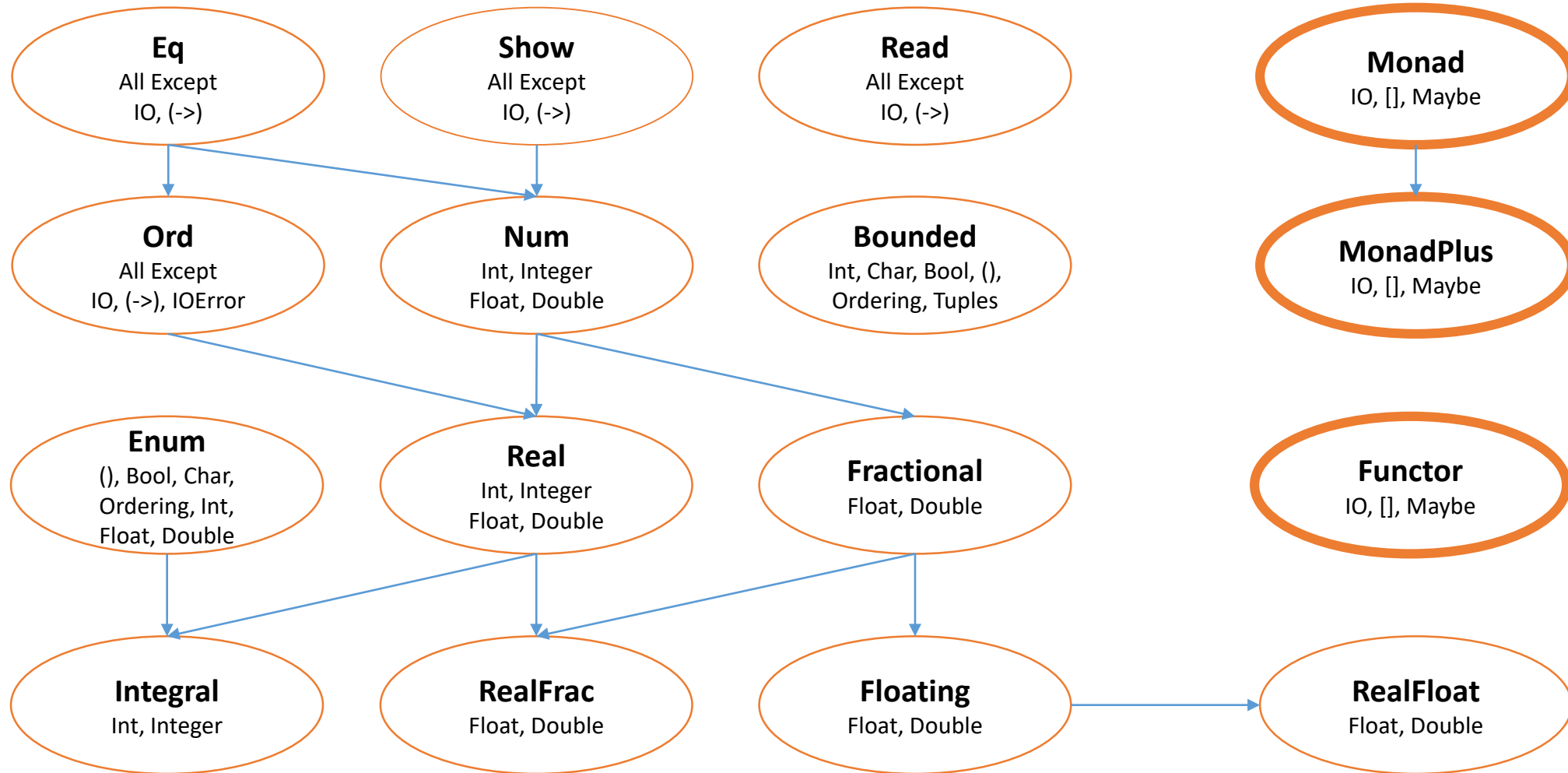
Main

Bloque Do

Acciones

Ejercicios

7.2. FUNCIONES SOBRE ACCIONES



Mondadas

Es una estructura que representa cálculos definidos como una secuencia de pasos, anidando funciones del mismo tipo.

Una mónada consiste de un constructor de tipo M y dos operaciones, unir y retorno.

La operación retorno toma un valor de un tipo plano y lo pone en un contenedor mónadico usando el constructor, creando así un valor mónadico.

La operación unir realiza el proceso inverso, extrayendo el valor original del contenedor y pasándolo a la siguiente función asociada en la tubería, posiblemente con chequeos adicionales y transformaciones

Mondadas: Maybe

`data Maybe t = Just t | Nothing`

Considérese al opción de tipo `Maybe a` (“tal vez un”), representando un valor que es o un valor del tipo `a`, o ninguno.

Para distinguirlos, se tienen dos constructores de tipo de dato algebraico: `Just t`, conteniendo el valor `t`, o `Nothing`, sin contenido.

Veamos un ejemplo

Mondadas: Maybe

```
zipTree1 :: (a -> b -> c) -> Tree a -> Tree b -> Maybe (Tree c)
zipTree1 _ (Node _ _ _) Empty = Nothing
zipTree1 _ Empty (Node _ _ _) = Nothing
zipTree1 _ Empty Empty = Just Empty
zipTree1 f (Node l1 x r1) (Node l2 y r2) =
  case zipTree1 f l1 l2 of
    Nothing -> Nothing
    Just l   -> case zipTree1 f r1 r2 of
      Nothing -> Nothing
      Just r   -> Just $ Node l (f x y) r
```


Mondadas: Maybe

Utilizando una función auxiliar, podemos tener un código más elegante

```
bindMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
bindMaybe mx f = case mx of
    Nothing -> Nothing
    Just x   -> f x

zipTree2 :: (a -> b -> c) -> Tree a -> Tree b -> Maybe (Tree c)
zipTree2 _ (Node _ _ _) Empty = Nothing
zipTree2 _ Empty (Node _ _ _) = Nothing
zipTree2 _ Empty Empty       = Just Empty
zipTree2 f (Node l1 x r1) (Node l2 y r2) =
    bindMaybe (zipTree2 f l1 l2) $ \l ->
        bindMaybe (zipTree2 f r1 r2) $ \r ->
            Just (Node l (f x y) r)
```

Funciones que operan sobre acciones

El operador `$` realiza el paralelismo entre las dos acciones siguientes:

`f x` (aplicar `f` al valor puro `x`), y

`f <$> x` (aplicar `f` al resultado del cálculo `x`).

```
lines :: String -> [String]
```

Gracias al operador `<$>`, se puede usar de la forma siguiente:

```
lineasDe :: FilePath -> IO [String]
```

```
lineasDe file = lines <$> readFile file
```

Funciones que operan sobre acciones

when :: Bool -> IO () -> IO ()

Está incluida en el módulo Control.Monad

El primer argumento es una condición.

Si la condición es cierta se devuelve el segundo argumento,

Si no se ejecuta “return ()”.

```
import Control.Monad

when_con_if = do
  c <- getChar
  if (c == ' ')
    then return()
    else
      do
        putChar c
        when_con_if

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

Funciones que operan sobre acciones

unless :: Bool -> IO () -> IO ()

Es semejante a when, pero verifica que la condición sea falsa.

```
unless_con_if = do
  c <- getChar
  if (c /= ' ')
    then return()
    else
      do
        putChar c
        unless_con_if

unless_con_unless = do
  c <- getChar
  unless (c /= ' ') $ do
    putChar c
    unless_con_unless
```

Funciones que operan sobre acciones

`sequence :: [IO a] -> IO [a]`

Toma una **lista de acciones** y devuelve una **acción** que realiza todas esas acciones y **obtiene como resultado una lista de los resultados de estas acciones**. Suele utilizarse junto a `map`.

```
Main> sequence (map print [1,2,3])  
1  
2  
3  
:: IO [()]
```

¿Qué ocurre si no usamos `sequence`?

```
Hugs> map print [1,2,3]  
ERROR - Cannot find "show" function for:  
*** Expression : map print [1,2,3]  
*** Of type    : [IO ()]
```

Funciones que operan sobre acciones

`mapM :: (a -> IO b) -> [a] -> IO [b]`

Mapea una acción sobre una lista.

Es equivalente a secuenciar el resultado de mapear una acción sobre una lista.

```
Main> mapM print [1,2,3]
```

```
1
```

```
2
```

```
3
```

```
:: IO [()]
```

Funciones que operan sobre acciones

`mapM_ :: (a -> IO b) -> [a] -> IO [b]`

Es equivalente a `mapM` pero desechando el resultado

```
Main> mapM_ print [1,2,3]
```

```
1
```

```
2
```

```
3
```

```
:: IO [()]
```

¿Qué diferencia hay entre la dos?

Funciones que operan sobre acciones

`mapM :: (a -> IO b) -> [a] -> IO [b]` vs `mapM_ :: (a -> IO b) -> [a] -> IO [b]`

Ambos toman funciones de la forma $(a \rightarrow m\ b)$ y producen un resultado final envuelto en m . La diferencia está en cuál es el resultado final: `mapM` nos da una $m\ [a]$ y `mapM_` nos da una $m\ ()$.

Esto significa que con `mapM` obtenemos una lista como salida, así como cualquier efecto que tenga m , mientras que `mapM_` solo nos da el efecto pero descarta la lista.

Funciones que operan sobre acciones

`mapM :: (a -> IO b) -> [a] -> IO [b]` vs `mapM_ :: (a -> IO b) -> [a] -> IO [b]`

Al devolver un `m ()` directamente desde `mapM_`, podemos evitar crear la lista en primer lugar, lo que puede ahorrar mucho trabajo y asignación de memoria para algunos casos de uso.

Si en última instancia no nos importan los elementos de la lista de todos modos, ¿por qué crear una lista?

Funciones que operan sobre acciones

forever :: IO a -> IO b

Definida en el módulo Control.Monad.

Toma una acción y la repite indefinidamente, es decir, genera una secuencia infinita repitiendo la acción.

Funciones que operan sobre acciones

`interact :: (String -> String) -> IO ()`

Toma una función que transforma cadenas y la aplica a todo el contenido de la entrada estándar, volcando el resultado sobre la salida estándar.

```
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in result
```

Funciones que operan sobre acciones

`interact :: (String -> String) -> IO ()`

Toma una función que transforma cadenas y la aplica a todo el contenido de la entrada estándar, volcando el resultado sobre la salida estándar.

```
import Data.Char(toUpper)

main :: IO ()
main = interact test

test :: String -> String
test [] = show ""
test a = show a

--main = interact (map toUpper . (++)) "Convertir caracter a mayuscula:\n\n"
--main = interact ((++) "Convertir caracter a mayuscula:\n\n" . map toUpper)
```

7.3. FICHEROS

Ficheros

Las funciones que hemos visto hasta ahora realizan acciones sobre la entrada y salida estándar.

Las acciones sobre ficheros son similares, pero necesitan un manejador de fichero (***handle***) para indicar el direccionamiento de la acción.

`openFile :: FilePath -> IO Mode -> IO Handle`

Acción que abre un fichero. Está definida en el módulo `System.IO`. El primer argumento es la ruta de acceso al fichero (***FilePath es un sinónimo de String***). El segundo argumento es el modo de apertura del fichero, que puede ser ***ReadMode***, ***WriteMode***, ***AppendMode*** o ***ReadWriteMode***. El resultado es el manejador del fichero.

Ficheros

hGetContents :: Handle -> IO String

Obtiene el contenido completo de un fichero. Está definida en el módulo System.IO. *Hay que tener en cuenta que la evaluación perezosa provoca que el contenido no se lea realmente hasta que no sea necesario.*

hClose :: Handle -> IO ()

Cierra un fichero. Está definida en el módulo System.IO. *Es responsabilidad del programador el cierre de los ficheros que se abren mediante openFile.*

Ficheros

```
import System.IO
main = do
    handle <- openFile "C:\\Users\\LUKE\\Dropbox\\_DOCENCIA\\2122\\HASKELL\\Sesion07\\file1.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

Main> main

esto es una linea

y esto es otra linea :: IO ()

Ficheros

withFile :: FilePath -> IOMode -> (Handle -> IO r) -> IO r

Encadena las acciones de abrir fichero, tratarlo y cerrarlo.

El primer argumento es la ruta del fichero.

El segundo argumento es el modo de apertura.

El tercer argumento es la acción a realizar sobre el fichero.

Ficheros

hGetChar :: Handle -> IO Char

Lee un carácter del fichero.

```
import IO
main = do hdl <- openFile "C:\\Users\\LUKE\\Dropbox\\_DOCENCIA\\2122\\HASKELL\\Sesion07\\file1.txt" ReadMode
        lee hdl
lee hdl = do t <- hIsEOF hdl
          if t then return()
              else do x <- hGetChar hdl
                    putChar x
                    lee hdl
```

Ficheros

hGetLine :: Handle -> IO String

Lee una línea del fichero.

¿Cómo leemos todas las líneas?

```
import IO
main = do x <- openFile "C:\\Users\\LUKE\\Dropbox\\_DOCENCIA\\2122\\HASKELL\\Sesion07\\file1.txt" ReadMode
        y <- hGetLine x
        putStr y
```

```
import IO
main = do hdl <- openFile "C:\\Users\\LUKE\\Dropbox\\_DOCENCIA\\2122\\HASKELL\\Sesion07\\file1.txt" ReadMode
        lee hdl
lee hdl = do t <- hIsEOF hdl
        if t then return()
            else do y <- hGetLine hdl
                  putStrLn y
                  lee hdl
```

Ficheros

hPutChar :: Handle -> Char -> IO ()

Escribe un carácter en un fichero.

```
import IO
import Char

main = do hdl <- openFile "C:\\Users\\LUKE\\Dropbox\\_DOCENCIA\\2122\\HASKELL\\Sesion07\\file2.txt" WriteMode
         hPutChar hdl (chr 66)
         hPutChar hdl ("A" !! 0)
         hPutChar hdl ("B" !! 0)
         hClose hdl
         hdl <- openFile "C:\\Users\\LUKE\\Dropbox\\_DOCENCIA\\2122\\HASKELL\\Sesion07\\file2.txt" ReadMode
         x <- hGetContents hdl
         putStr x
```

Ficheros

hPutStr :: Handle -> String -> IO ()

Escribe una línea en un fichero.

```
import IO
import Char

main = do hdl <- openFile "C:\\Users\\LUKE\\Dropbox\\_DOCENCIA\\2122\\HASKELL\\Sesion07\\file2.txt" WriteMode
        hPutStr hdl "cadena 1"
        hPutStrLn hdl "cadena 2"
        hPutStr hdl "cadena 3"
        hClose hdl
        hdl <- openFile "C:\\Users\\LUKE\\Dropbox\\_DOCENCIA\\2122\\HASKELL\\Sesion07\\file2.txt" ReadMode
        x <- hGetContents hdl
        putStr x
```

Ficheros

readFile :: FilePath -> IO String

Abre un fichero y obtiene una acción que lee su contenido.

Podemos obtener este contenido con la instrucción <-.

Cuando se utiliza esta función no se obtiene el manejador así que el cierre del fichero es responsabilidad de Haskell.

```
import IO
main = do x <- readFile "C:\\Users\\LUKE\\Dropbox\\_DOCENCIA\\2122\\HASKELL\\Sesion07\\file3.txt"
        y <- rList x
        print (sum y)

rList :: String -> IO [Int]
rList = readIO
```

Ficheros

writeFile :: FilePath -> String -> IO ()

Abre un fichero en modo escritura y genera una acción de escribir el segundo argumento en el fichero.

```
main = writeFile "C:\\Users\\LUKE\\Dropbox\\_DOCENCIA\\2122\\HASKELL\\Sesion07\\file4.txt" cadena
cadena = "Esta es la cadena a escribir"
```

appendFile :: FilePath -> String -> IO ()

Abre un fichero en modo añadir y genera una acción de escribir el segundo argumento en el fichero.

```
main = appendFile "C:\\Users\\LUKE\\Dropbox\\_DOCENCIA\\2122\\HASKELL\\Sesion07\\file4.txt" cadena
cadena = "\\nCadena2" ++ "Cadena3"
```

7.4. PRACTICAR

Ejercicios

- Realizar un ejemplo que lea de un fichero varias líneas y las muestre por pantalla. Si es posible, mostrar la posición de la línea al principio de la misma.
- Realizar un ejemplo que realice una acción con interact de forma indefinida.
- Realizar un ejemplo que copie un fichero A en el fichero B