Programación Funcional

Curso 2019-20

ENTRADA / SALIDA

Entrada/salida

- La entrada/salida es un problema en el paradigma funcional puro
- En Haskell: entrada/salida *monádica*
- Mónadas: Abstracción adecuada para integrar en el sistema de tipos cómputos efectuados en secuencia que acarrean efectos laterales (no necesariamente I/O)
- Propuestas por Phil Wadler (adaptando ideas previas de otros)

¿Por qué es un problema la E/S para Haskell?

¿Qué pasaría si tuviésemos, por ejemplo, `getInt::Int`?

- Por el principio de que toda expresión e de tipo T tiene un valor $[\![e]\!]$ del tipo T, $[\![\texttt{getInt}]\!]$ sería un número entero.
- Por la transparencia referencial $[\![\texttt{getInt}]\!]$ debería ser único, independientemente de dónde y cuándo se use `getInt`.
- Podemos formar una expresión como `getInt - getInt`.
- Hasta la fecha se tenía: $[\![e - e']\!] = [\![e]\!] - [\![e']\!]$
- Debería entonces ser:
  $[\![\texttt{getInt - getInt}]\!] = [\![\texttt{getInt}]\!]$ - $[\![\texttt{getInt}]\!] = 0$
- ¡Absurdo! Significaría que al evaluar `getInt - getInt` leemos dos veces el mismo entero
- Similar: `getInt + getInt` debería ser equivalente a `2*getInt`
- Más cosas: hasta la fecha, para evaluar $e - e'$ podemos, si queremos, evaluar e y e' en paralelo. Ahora ya no!

Entrada/Salida en Haskell

El tipo IO a

- Procesos de entrada/salida: expresiones del tipo `IO a`
- ¿Qué es un valor del tipo `IO a`?: *acción* de I/O que, **si se efectúa**, produce un efecto de I/O con un resultado asociado de tipo `a`.
- `getInt::IO Int`: acción de leer un entero
- `getChar::IO Char`: acción de leer un carácter
- `putChar::Char -> IO ()`: acción de escribir un carácter

  `()` $\rightsquigarrow$ tipo con un solo valor, que es también `()`

  Las acciones de tipo `IO ()` no generan valor asociado
- La acción representada por una e`::IO a` se efectúa si e es evaluada al evaluar la expresión escrita en la consola o la expresión `main` de un módulo.

Algunas acciones I/O básicas y predefinidas

- `getChar:: IO Char`
- `putChar:: Char -> IO ()`
- `return:: a -> IO a`
  `return x` $\rightsquigarrow$ acción sin efecto lateral con valor asociado `x`
  Hace falta para expresar acciones complejas
- `getLine:: IO String`
- `putStr:: String -> IO ()`
- `print :: Show a => a -> IO ()`
  `print = putStr . show`

Secuenciación de acciones de I/O: operador >>= (*'then'*)

```
infixl 1 >>=
(>>=):: IO a -> (a -> IO b) -> IO b
```

- `IO a` primera acción; tendrá un valor asociado `x::a`
- `(a -> IO b)` `x::a` determina la segunda acción
- `IO b` el resultado es esta segunda acción

Leer una línea y repetirla dos veces

```
eco2:: IO ()
eco2 = getLine >>= \xs -> putStr (xs++"\n"++xs++"\n")
```

putStr,getLine definidas mediante putChar,getChar

```
putStr:: String -> IO ()
putStr []     = return ()
putStr(c:cs) = putChar c >>= \_ -> putStr cs

getLine::IO String
getLine = getChar >>= \c ->
          if c=='\n' then return []
                     else getLine >>= \cs ->
                          return (c:cs)
```

Secuenciación de acciones: notación `do`

```
do  x1 <- e1
    e2
    ...
    xn <- en
    ...
    em
```

```
e1::IO a1, x1::a1
e2::IO a2
...
en::IO an, x::an
...
em::IO am ⇝ valor del do
```

`do` es un azúcar sintáctico

```
do  x <- e
    e'
```

≡

```
e >>= \x -> e'
```

```
do  e
    e'
```

≡

```
e >>= \_ -> e'
```

A `do` se le aplica la regla de indentación

Leer una línea y repetirla dos veces

```
eco2:: IO ()
eco2 = do xs <- getLine
          putStr (xs++"\n"++xs++"\n")
```

putStr,getLine definidas mediante putChar,getChar

```
putStr:: String -> IO ()
putStr []     = return ()
putStr(c:cs) = do putChar c
                  putStr cs

getLine::IO String
getLine = do c <- getChar
             if c=='\n' then return []
                        else do cs <- getLine
                                return (c:cs)
```

Leer un entero

```
getInt:: IO Int
getInt = do line <- getLine
            return (read line::Int)
```

¿por qué hace falta `::Int` ?

`getInt` no es predefinida del Prelude

Lee enteros y escribe sus cubos hasta que el entero sea 0

```
cubos:: IO ()
cubos =
 do putStrLn "Escribe un entero (0 para salir)"
    n <- getInt
    if n==0 then return ()
            else do putStr (show n++"^3 = ")
                    print (n^3)
                    cubos
```

La transparencia referencial, a salvo

- `getInt - getInt` sin sentido, mal tipado
-
```
do  x <- getInt
    y <- getInt
    print (x-y)
```
no tiene por qué ser $= 0$
-
```
do  x <- getInt
    print (x+x)
```
$\simeq$
```
do  x <- getInt
    print (2*x)
```

La clase `Monad`

- `IO` es una instancia de la clase `Monad`
- `return` , `>>=` son métodos de `Monad`
- La notación `do` es aplicable para expresar secuenciación de cómputos en tipos de `Monad`
- `Monad` es una **clase de constructoras de tipo**
  - `IO` no es un tipo, sino una constructora de tipos
  - Es la constructora de tipos `IO` quien pertenece a `Monad` , no los tipos `IO Int` , `IO String` , . . .
  - Otras constructoras de tipo de `Monad` son, por ejemplo, `Maybe` o `[]`