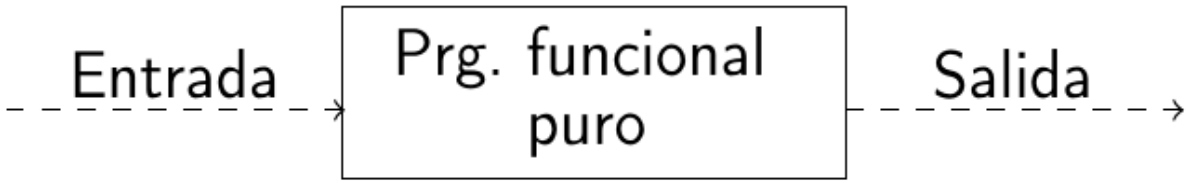
Mónada IO

Cecilia Manzino

19/11/2024

Transparencia referencial

- Hasta ahora definimos funciones puras en Haskell.

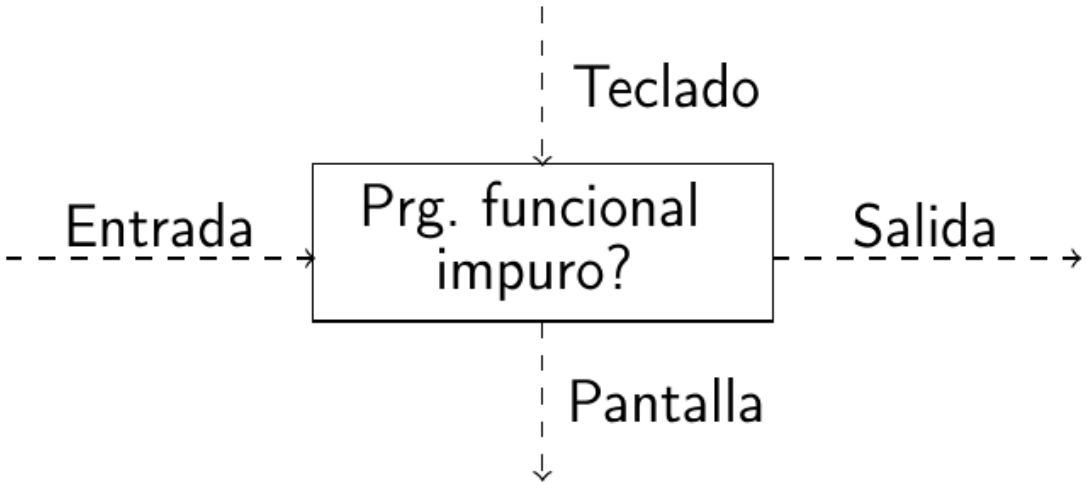


- Escribimos programas como un conjunto de ecuaciones, donde el flujo de datos era explícito. Ésto nos permite por ejemplo reemplazar x+y por y+x.

- Transparencia referencial: "En un programa, una expresión E puede sustituirse por otra de igual valor V, sin modificar la semántica del programa".

Programas interactivos

- El esquema para un programa interactivo es el siguiente:



- ¿Podrá un programa interactivo ser modelado como una función pura?
  ¿Vale x+y = y+x?

```
x = print "hola"; return 2
y = print "mundo"; return 3
```

E/S

- Los lenguajes puros nos permiten razonar sobre los programas y tienen el beneficio de la evaluación lazy, mientras que los lenguajes impuros ofrecen el beneficio de la E/S.

- Haskell utiliza el sistema de tipos para diferenciar los cómputos puros de los impuros, definiendo el constructor de tipos `IO` para representar cómputos que realizan E/S:

```
type IO a = World -> (a,World)
```

Cuando se ejecuta un valor de tipo `IO a`, se realiaza alguna operación de E/S y se devuelve un valor de tipo a.

Mónada IO

- El sistema de tipos nos asegura que una función `f :: Int -> String`, no puede realizar operaciones de E/S.
- `IO` es una mónada de un sólo camino, no se puede definnir una funci ón de tipo `IO a -> a`.
- La definición de ésta mónada es específica de la plataforma, no se exportan sus constructores.
- Aunque no se pueda salir de ésta mónada, en la práctica la mayor parte de funciones en Haskell son puras.
  Haskell es considerado un lenguaje puramente funcional (si ignoramos `unsafePerformIO`).

Operaciones de E/S básicas

- `getChar :: IO Char`
  Lee un caracter del teclado, lo imprime en la pantalla y lo devuelve como valor.

- `putChar :: Char -> IO ()`
  Muestra un caracter en la pantalla.

- `putStr :: String -> IO ()`
- `putStrLn :: String -> IO ()`
- `getLine :: IO String`
- `readFile :: FilePath -> IO String`
  Lee el contenido de un archivo y lo devuelve.

Secuenciación

```
(>>) :: Monad m => m a -> m b -> m b
t >> u = t >>= \_ -> u
```

Mientras que en las computaciones puras el orden es irrelevante, en las no puras no.

```
g :: IO (Char, Char)
g  =  do  c <- getChar
          getChar
          d <- getChar
          return (c, d)
```

Ejercicio

1. Dar una definición de `getLine` usando `getChar`, donde `getLine` lee caracteres del teclado hasta que lee `'\n'` y devuelve la cadena leída.

Solución:

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

Ejemplo: Ahorcado

Basado en transparencias de G. Hutton y modificado por Mauro Jaskelioff.

```
ahorcado :: IO ()
ahorcado =
      do putStrLn "Piense en una palabra"
         palabra <- sgetLine
         putStrLn "Intente adivinarla:"
         adivina palabra
```

Ejemplo: Ahorcado

`sgetLine` lee una línea del teclado, mostrando un guión por cada caracter ingresado.

```
sgetLine :: IO String
sgetLine = do hSetEcho stdin False
              palabra <- sgetLine'
              hSetEcho stdin True
              return palabra
```

Ejemplo: Ahorcado

Lee una cadena del teclado e imprime un '-' por cada caracter leído.

```
sgetLine' :: IO String
sgetLine' = do x <- getChar
               if x ==  '\n'
               then do putChar x
                       return []
               else do putChar '-'
                       xs <- sgetLine'
                       return (x:xs)
```

Ejemplo: Ahorcado

Lee los intentos hasta que el juego termina cuando se adivina la palabra.

```
adivina :: String -> IO ()
adivina palabra =
   do putStr "> "
      xs <- getLine
      if xs == palabra
        then putStrLn "Esa es la palabra!"
        else do putStrLn (diff palabra xs)
                adivina palabra
```

Ejemplo: Ahorcado

La función diff indica qué caracteres de una cadena están en la otra cadena.

```
diff :: String -> String -> String
diff xs ys = [if elem x ys then x else '-'↩
    | x <- xs]
```

Por ejemplo,

```
> diff "computacion" "compilacion"
"comp--acion"
```