Programación Declarativa 2020-2
Mónadas 103 `IO`

Javier Enríquez Mendoza Favio E. Miranda Perea

Facultad de Ciencias UNAM

28 de mayo de 2020

Programas interactivos

Un programa interactivo es aquel que requiere un intercambio de información de un usuario para su funcionamiento

En Haskell se modela como una función que toma el estado del *mundo real* como argumento y regresa un estado con modificaciones

Supongamos que existe el tipo `RealWorld` para modelar el estado del mundo real, el tipo `IO` se define como sigue:

```
type IO = RealWorld -> RealWorld
```

Pero el programa también puede devolver un valor, aparte del estado, entonces hay que agregarlo al tipo

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

en donde `a` es el tipo del valor de regreso

Acciones

Las expresiones de tipo `IO` se conocen como acciones

Si `a :: IO Int` entonces `a` es una acción que regresa un entero

Cuando una acción no va a devolver un valor su tipo es `IO ()`, es decir regresa la tupla vacía

La biblioteca `System.IO` tiene muchas acciones definidas con diferentes funcionamientos que van desde leer o escribir un carácter en la pantalla, hasta leer o escribir archivos en la memoria


IO IN HASKELL
SO SIMPLE
makeameme.org

Ejemplos de Acciones

Algunas acciones definidas son:

Entre muchas otras

Ejemplos de Acciones

Algunas acciones definidas son:

- `getChar ::IO Char` lee un carácter

Entre muchas otras

Ejemplos de Acciones

Algunas acciones definidas son:

- `getChar ::IO Char` lee un carácter
- `putChar ::Char ->IO ()` escribe un carácter, no regresa ningún valor

Entre muchas otras

Ejemplos de Acciones

Algunas acciones definidas son:

- `getChar ::IO Char` lee un carácter
- `putChar ::Char ->IO ()` escribe un carácter, no regresa ningún valor
- `getLine ::IO String` Lee una cadena de texto

Entre muchas otras

Ejemplos de Acciones

Algunas acciones definidas son:

- `getChar ::IO Char` lee un carácter
- `putChar ::Char ->IO ()` escribe un carácter, no regresa ningún valor
- `getLine ::IO String` Lee una cadena de texto
- `putStr ::String ->IO ()` escribe una cadena de texto

Entre muchas otras

Ejemplos de Acciones

Algunas acciones definidas son:

- `getChar ::IO Char` lee un carácter
- `putChar ::Char ->IO ()` escribe un carácter, no regresa ningún valor
- `getLine ::IO String` Lee una cadena de texto
- `putStr ::String ->IO ()` escribe una cadena de texto
- `putStrLn ::String ->IO ()` igual lee una cadena de texto pero le pone un salto de linea al final

Entre muchas otras

Ejemplos de Acciones

Algunas acciones definidas son:

- `getChar ::IO Char` lee un carácter
- `putChar ::Char ->IO ()` escribe un carácter, no regresa ningún valor
- `getLine ::IO String` Lee una cadena de texto
- `putStr ::String ->IO ()` escribe una cadena de texto
- `putStrLn ::String ->IO ()` igual lee una cadena de texto pero le pone un salto de linea al final
- `readFile ::FilePath ->IO String` Lee un archivo de la memoria y regresa el contenido como una cadena

Entre muchas otras

Ejemplos de Acciones

Algunas acciones definidas son:

- `getChar ::IO Char` lee un carácter
- `putChar ::Char ->IO ()` escribe un carácter, no regresa ningún valor
- `getLine ::IO String` Lee una cadena de texto
- `putStr ::String ->IO ()` escribe una cadena de texto
- `putStrLn ::String ->IO ()` igual lee una cadena de texto pero le pone un salto de linea al final
- `readFile ::FilePath ->IO String` Lee un archivo de la memoria y regresa el contenido como una cadena
- `writeFile ::FilePath ->String ->IO ()` escribe el texto en el archivo

Entre muchas otras

action

Definamos ahora una función encargada de tomar un elemento y convertirlo en una acción

```
action :: a -> IO a
accion v = \w -> (v,w)
```

Vista de otra forma la función `action` solo se encarga de devolver el valor `v` sin modificar el estado del mundo real

combine

Ahora definamos una función encargada de combinar acciones

```
combine :: IO a -> (a -> IO b) -> IO b
combine f g = \w -> case f w of
                      (v,w') -> (g v w')
```

Es decir ejecuta la acción `f` con el estado del mundo real `w` lo que regresa un valor `v` y un nuevo estado modificado `w'` con los cuales ejecuta la acción `g`

Eso suena como secuencia

¿Acaso IO es una mónada?

Es claro que la interacción con el *mundo real* se trata de una acción que puede tener *side effects*, es decir una acción imperativa

Y acabamos de definir una función que ejecuta acciones de forma secuencial

Nadie lo vea venir pero si, `IO` es también una mónada

La instancia de la clase `Monad` es la siguiente:

```
instance Monad IO where
  return = action
  (>>=) = combine
```


SAY MONAD
ONE MORE TIME
memegenerator.net

¿Y las mónadas son realmente necesarias?

Esta es una pregunta bastante frecuente, pues lenguajes como Python, Scala o Scheme no parecen necesitar mónadas para hacer todo esto

Entonces, ¿las mónadas son necesarias? ¿o son otra forma de hacer mas obscurantista el código en Haskell?

La respuesta es SI, las mónadas son necesarias

Pero ¿por qué en Haskell son necesarias y en otros lenguajes no?

Respuesta corta: Porque Haskell es ~~cool~~ diferente

Respuesta larga(Python)

Definamos en Python un programa que recibe dos cadenas del usuario y regresa la concatenación de ellas.

```
def tomaDos():
    x = input()
    y = input()
    return (x + y)
```

Ahora escribamos otro programa que recibe una única cadena del usuario y regresa la concatenación de esa cadena con ella misma.

```
def tomaUna():
    x = input()
    return (x + x)
```

Respuesta larga(Haskell)

Intentemos traducir estos programas a Haskell sin usar mónadas.

```
tomaDos () = res
  where
    x   = input ()
    y   = input ()
    res = x + y


tomaUna () = res
  where
    x   = input()
    res = x ++ x
```

Respuesta larga

Parece que esos programas hacen lo que queremos, y no se necesitan mónadas

Pero no es así, los programas en Haskell no van a funcionar de la forma en la que esperaríamos

Pero ¿Por qué?

Porque Haskell es ~~cool~~ diferente

Veamos que es lo que hace a Haskell diferente

Orden de Ejecución Python

En Python tenemos un orden de ejecución explicito, es decir, ejecuta el código de forma lineal. Ejecuta la primera linea, una vez que termina la segunda y así continua.

Esto quiere decir que la función `tomaDos` en Python se ejecuta en este orden:

Orden de Ejecución Python

En Python tenemos un orden de ejecución explicito, es decir, ejecuta el código de forma lineal. Ejecuta la primera linea, una vez que termina la segunda y así continua.

Esto quiere decir que la función `tomaDos` en Python se ejecuta en este orden:

Orden de Ejecución Python

En Python tenemos un orden de ejecución explicito, es decir, ejecuta el código de forma lineal. Ejecuta la primera linea, una vez que termina la segunda y así continua.

Esto quiere decir que la función `tomaDos` en Python se ejecuta en este orden:

1. `x = input()`

Orden de Ejecución Python

En Python tenemos un orden de ejecución explicito, es decir, ejecuta el código de forma lineal. Ejecuta la primera linea, una vez que termina la segunda y así continua.

Esto quiere decir que la función `tomaDos` en Python se ejecuta en este orden:

1. `x = input()`
2. `y = input()`

Orden de Ejecución Python

En Python tenemos un orden de ejecución explicito, es decir, ejecuta el código de forma lineal. Ejecuta la primera linea, una vez que termina la segunda y así continua.

Esto quiere decir que la función `tomaDos` en Python se ejecuta en este orden:

1. `x = input()`
2. `y = input()`
3. `return (x + y)`

Orden de Ejecución Haskell

Pero Haskell ejecuta los enunciados por dependencias funcionales, es decir, por necesidad

Si se ejecuta la función `tomaDos` en Haskell lo hace de la siguiente forma:

Pero, ¡¡es lo mismo!!

En este caso si, pero que pasa al ejecutar la función `tomaUna`

Orden de Ejecución Haskell

Pero Haskell ejecuta los enunciados por dependencias funcionales, es decir, por necesidad

Si se ejecuta la función `tomaDos` en Haskell lo hace de la siguiente forma:

Pero, ¡¡es lo mismo!!

En este caso si, pero que pasa al ejecutar la función `tomaUna`

Orden de Ejecución Haskell

Pero Haskell ejecuta los enunciados por dependencias funcionales, es decir, por necesidad

Si se ejecuta la función `tomaDos` en Haskell lo hace de la siguiente forma:

1. Primero ve que tiene que regresar `res` y busca su valor `x ++ y`

Pero, ¡¡es lo mismo!!

En este caso si, pero que pasa al ejecutar la función `tomaUna`

Orden de Ejecución Haskell

Pero Haskell ejecuta los enunciados por dependencias funcionales, es decir, por necesidad

Si se ejecuta la función `tomaDos` en Haskell lo hace de la siguiente forma:

1. Primero ve que tiene que regresar `res` y busca su valor `x ++ y`
2. Busca el valor de `x` en `x = input ()` lo sustituye resultado `input () ++ y`

Pero, ¡¡es lo mismo!!

En este caso si, pero que pasa al ejecutar la función `tomaUna`

Orden de Ejecución Haskell

Pero Haskell ejecuta los enunciados por dependencias funcionales, es decir, por necesidad

Si se ejecuta la función `tomaDos` en Haskell lo hace de la siguiente forma:

1. Primero ve que tiene que regresar `res` y busca su valor `x ++ y`
2. Busca el valor de `x` en `x = input ()` lo sustituye resultado `input () ++ y`
3. Busca el valor de `y` en `y = input ()`y lo sustituye obteniendo `input () ++ input ()`

Pero, ¡¡es lo mismo!!

En este caso si, pero que pasa al ejecutar la función `tomaUna`

Orden de Ejecución Haskell

Pero Haskell ejecuta los enunciados por dependencias funcionales, es decir, por necesidad

Si se ejecuta la función `tomaDos` en Haskell lo hace de la siguiente forma:

1. Primero ve que tiene que regresar `res` y busca su valor `x ++ y`
2. Busca el valor de `x` en `x = input ()` lo sustituye resultado `input () ++ y`
3. Busca el valor de `y` en `y = input ()`y lo sustituye obteniendo `input () ++ input ()`
4. Entonces pide ambas entradas al usuario para calcular el resultado.

Pero, ¡¡es lo mismo!!

En este caso si, pero que pasa al ejecutar la función `tomaUna`

Orden de Ejecución Haskell

Pero Haskell ejecuta los enunciados por dependencias funcionales, es decir, por necesidad

Si se ejecuta la función `tomaDos` en Haskell lo hace de la siguiente forma:

1. Primero ve que tiene que regresar `res` y busca su valor `x ++ y`
2. Busca el valor de `x` en `x = input ()` lo sustituye resultado `input () ++ y`
3. Busca el valor de `y` en `y = input ()`y lo sustituye obteniendo `input () ++ input ()`
4. Entonces pide ambas entradas al usuario para calcular el resultado.
5. Pide al usuario 2 entradas para terminar la ejecución

Pero, ¡¡es lo mismo!!

En este caso si, pero que pasa al ejecutar la función `tomaUna`

Orden de Ejecución

El orden de ejecución de la función `tomaUna` en Haskell es el siguiente

Pero, solo necesitábamos una entrada

Desde el punto de vista de Haskell ambas funciones son iguales

Pero entonces ¿cómo vamos a escribir programas de este estilo en Haskell?

Con Mónadas

Orden de Ejecución

El orden de ejecución de la función `tomaUna` en Haskell es el siguiente

1. Primero ve que tiene que regresar `res` y busca su valor `x ++ x`

Pero, solo necesitábamos una entrada

Desde el punto de vista de Haskell ambas funciones son iguales

Pero entonces ¿cómo vamos a escribir programas de este estilo en Haskell?

Con Mónadas

Orden de Ejecución

El orden de ejecución de la función `tomaUna` en Haskell es el siguiente

1. Primero ve que tiene que regresar `res` y busca su valor `x ++ x`
2. Busca el valor de `x` en `x = input ()` y lo sustituye resultado `input () ++ input ()`

Pero, solo necesitábamos una entrada

Desde el punto de vista de Haskell ambas funciones son iguales

Pero entonces ¿cómo vamos a escribir programas de este estilo en Haskell?

Con Mónadas

Orden de Ejecución

El orden de ejecución de la función `tomaUna` en Haskell es el siguiente

1. Primero ve que tiene que regresar `res` y busca su valor `x ++ x`
2. Busca el valor de `x` en `x = input ()` y lo sustituye resultado `input () ++ input ()`
3. Pide al usuario 2 entradas para terminar la ejecución

Pero, solo necesitábamos una entrada

Desde el punto de vista de Haskell ambas funciones son iguales

Pero entonces ¿cómo vamos a escribir programas de este estilo en Haskell?

Con Mónadas

una vez más todo está bajo control gracias a

las Mónadas

Ejemplo

Hagamos un programa que reciba 2 palabras del usuario y escriba una frase sencilla en pantalla

```
phrase :: IO ()
phrase = putStrLn "Escribe un nombre:" >>
         getLine >>= \name ->
         putStrLn "Escribe un adjetivo:" >>
         getLine >>= \adj ->
         putStrLn (name ++ "es muy " ++ adj)
```

Recordemos que el operador (>>) ejecuta los cómputos de forma secuencial sin almacenar el resultado

do

Reescribamos la función anterior usando la notación do

```
phrase1 :: IO ()
phrase1 = do
    putStrLn ("Escribe un nombre:")
    name <- getLine
    putStrLn "Escribe un adjetivo:"
    adj <- getLine
    putStrLn (name ++ " es muy " ++ adj)
```

Ejecución

La ejecución se vería así

```
Escribe un nombre:
> Haskell
Escribe un adjetivo:
> cool
Haskell es muy cool
```

Main

Cuando se compila código en Haskell, el compilador busca la definición de la función `main` la cual definirá el comportamiento del programa

Una vez creado el ejecutable, al correrlo se ejecutara la función `main`

Esta función tiene tipo `IO`, usualmente se trata de `IO ()`

Por ejemplo el siguiente programa

```
main :: IO ()
main = putStrLn "Hello World"
```


I KNOW
HASKELL
memegenerator.net