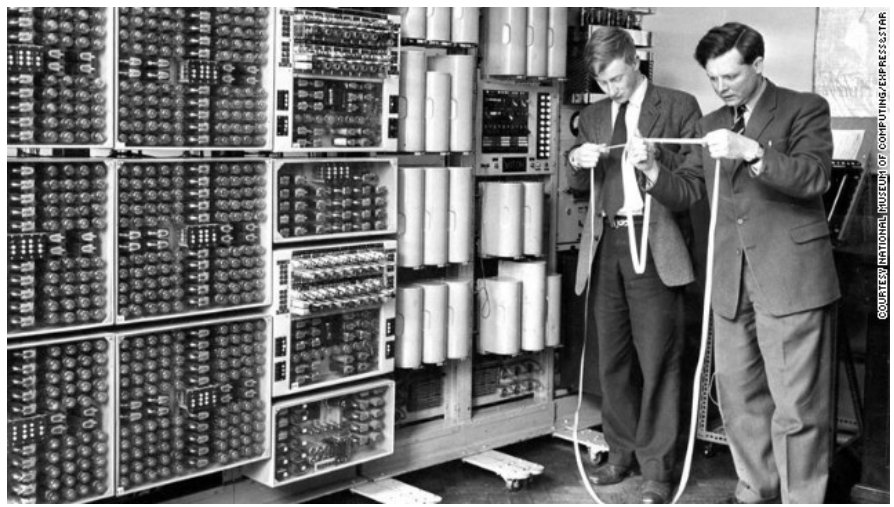
Entrada/Salida

Mauro Jaskelioff

Viernes 20 de Octubre, 2017



Entrada/Salida

- Los programas escritos hasta ahora no tienen efectos como entrada/salida.
- Esto nos permite razonar en forma ecuacional.
  - Siempre podemos reemplazar iguales por iguales

$$x + y = y + x$$

- ¿Cómo razonamos ecuacionalmente si tenemos E/S?
- Considere

$$x = \textit{print}\ \texttt{"hola"};\ \textit{return}\ 2$$
$$y = \textit{print}\ \texttt{"mundo"};\ \textit{return}\ 3$$

¿Sigue siendo válida la ecuación?

Computaciones con efectos

- Para diferenciar los valores *puros* de los que pueden producir algún *efecto lateral* utilizaremos el sistema de tipos.
- Por ejemplo, considere una función

$$f : Int \to ES\ String$$

- El sistema de tipos nos avisa que f:
  - toma un entero;
  - y devuelve una cadena
  - posiblemente luego de realizar ciertas operaciones de entrada/salida.
- El sistema de tipos asegura, por ejemplo, que una función $f :: Int \to String$ no puede acceder o modificar archivos, por ejemplo.
- Para poder componer computaciones pedimos que sean una **mónada**.

Repaso de Mónadas

- La clase de mónadas

$$\begin{aligned}&\textbf{class } \mathit{Monad}\ m \textbf{ where}\\ &\quad \mathit{return} :: a \to m\ a\\ &\quad (\gg\!=)\ :: m\ a \to (a \to m\ b) \to m\ b\end{aligned}$$

  más leyes de mónadas

- Secuenciación

$$\begin{aligned}&(\gg)\ \ :: \mathit{Monad}\ m \Rightarrow m\ a \to m\ b \to m\ b\\ &t \gg u = t \gg\!= \backslash\_ \to u\end{aligned}$$

Lenguaje para entrada salida

- Anteriormente, definimos el lenguaje:

$$\begin{aligned}\textbf{data}\ \mathit{ES}\ a = {} & \mathit{Read}\ (\mathit{Char} \to \mathit{ES}\ a) \\ \mid {} & \mathit{Write}\ \mathit{Char}\ (\mathit{ES}\ a) \\ \mid {} & \mathit{Var}_{\mathit{ES}}\ a\end{aligned}$$

- Vimos que es una instancia de mónada.

$$\begin{aligned}&\textbf{instance}\ \mathit{Monad}\ \mathit{ES}\ \textbf{where} \\ &\quad \mathit{return} = \mathit{Var}_{\mathit{ES}} \\ &\quad \mathit{Read}\ k \gg\!= v = \mathit{Read}\ (\lambda c \to k\ c \gg\!= v) \\ &\quad \mathit{Write}\ c\ t \gg\!= v = \mathit{Write}\ c\ (t \gg\!= v) \\ &\quad \mathit{Var}_{\mathit{ES}}\ a \gg\!= v = v\ a\end{aligned}$$

Razonando ecuacionalmente

- Representamos los *efectos computacionales* con una mónada.
- Podemos seguir razonando ecuacionalmente:
  Los programas anteriores tienen tipo:

$$x :: \mathit{ES}\ \mathit{Int} \qquad\qquad y :: \mathit{ES}\ \mathit{Int}$$
$$x = \mathit{print}\ \texttt{"hola"}; \mathit{return}\ 2 \qquad y = \mathit{print}\ \texttt{"mundo"}; \mathit{return}\ 3$$

- La ecuación $x + y = y + x$ no tiene más sentido, ya que $+$ toma enteros (puros), y no computaciones sobre enteros.
- La "suma" para *ES Int* es ahora:

$$\begin{aligned} &\mathit{suma} :: \mathit{ES}\ \mathit{Int} \to \mathit{ES}\ \mathit{Int} \to \mathit{ES}\ \mathit{Int} \\ &\mathit{suma}\ \mathit{ex}\ \mathit{ey} = \mathit{ex} \gg\!= \lambda x \to \\ &\qquad\qquad\qquad\quad \mathit{ey} \gg\!= \lambda y \to \\ &\qquad\qquad\qquad\quad \mathit{return}\ (x + y) \end{aligned}$$

- El orden de los efectos es explícito, por lo que en general $\mathit{suma}\ \mathit{ex}\ \mathit{ey} \not\equiv \mathit{suma}\ \mathit{ey}\ \mathit{ex}$.

Términos como tipo abstracto

- Todos los programas de entrada/salida se pueden escribir usando solamente:
  - La interfaz mónadica: *return* y $\gg=$.
  - las operaciones *writeChar* y *readChar*
- El constructor de tipos *ES* puede ser abstracto (solo las operaciones arriba mencionadas son expuestas al programador).
- Las operaciones *return*, $\gg=$, *writeChar*, y *readChar* son las únicas que tienen acceso a la representación interna del constructor de tipos *ES*.

Entrada/Salida en un lenguaje funcional puro

- El tipo *ES* provee la **sintaxis** de operaciones de entrada/salida, pero las operaciones no son ejecutadas.
- Si tuvieramos una función $ejecutar :: ES\ a \rightarrow a$ que ejecute las operaciones de entrada salida, estaríamos en problemas
- '!No podríamos razonar ecuacionalmente!

$$
\begin{aligned}
&x, y :: Int \\
&x = ejecutar\ (writeChar\ \text{'1'} \gg return\ 1) \\
&y = ejecutar\ (writeChar\ \text{'2'} \gg return\ 2) \\
&x + y \equiv y + x\ ?
\end{aligned}
$$

La Mónada IO

- Haskell provee la mónada *IO*.
- *IO* es un tipo abstracto, que es una mónada (provee *return* y $\gg=$), mas ciertas operaciones básicas de entrada/salida (entre otras operaciones).
- Haskell ejecuta las operaciones de la mónada IO (a diferencia de *ES* que es sólo sintaxis).
  - Un programa compilado debe tener una función *main* :: *IO* ()
- *IO* no puede ser definido **en** Haskell, sino que es una primitiva del lenguaje.

Algunas Operaciones de IO

- Algunas operaciones de la mónada *IO* son:
  - $getChar :: IO\ Char$
  - $putChar :: Char \to IO\ ()$
  - $getLine :: IO\ String$
  - $putStr :: String \to IO\ ()$
  - $putStrLn :: String \to IO\ ()$
- Hay más operaciones para acceso a archivos, soporte POSIX, acceso a variables de entorno, llamadas a procedimientos en otros lenguajes, etc.
- Para detalles, ver, por ej., bibliotecas *System* y *Foreign* en `http://www.haskell.org/ghc/docs/latest/html/libraries/index.html`

Observaciones

- Notar la imposibilidad[1] de una función con tipo

$$IO\ a \to a$$

- La mónada *IO* es una muy buena solución pero no es la última palabra en modelado de efectos computacionales. Sería deseable:
  - Mayor granularidad (*IO Char* puede leer del teclado o borrar el disco rígido)
  - Especificación que permita razonar sobre la interacción de efectos computacionales.

[1] Prohibido preguntar acerca de *unsafeperformIO*

Mejorando la notación

- Un programa monádico típico tiene la siguiente estructura:

$$
\begin{aligned}
& m_1 \gg\!= \lambda x_1 \rightarrow \\
& m_2 \gg\!= \lambda x_2 \rightarrow \\
& \quad \vdots \\
& m_n \gg\!= \lambda x_n \rightarrow \\
& \mathit{return}\ (f\ x_1\ x_2\ \ldots\ x_n)
\end{aligned}
$$

- m_1 ejecuta algunas operaciones y produce un valor x_1, en base al cual se ejecuta m_2 ..., finalmente se devuelve un valor calculado por una función f de los valores obtenidos anteriormente.

Notación **do**

- Haskell provee una notación especial para este tipo de programas:

$$
\begin{aligned}
&\textbf{do}\ x_1 \leftarrow m_1 \\
&\quad\ x_2 \leftarrow m_2 \\
&\qquad \vdots \\
&\quad\ x_n \leftarrow m_n \\
&\quad\ \mathit{return}\ (f\ x_1\ x_2\ \ldots\ x_n)
\end{aligned}
$$

- Las expresiones $x_i \leftarrow m_i$ son llamadas *generadores*.
- Se requiere que todos los generadores empiecen en la misma columna (o usar llaves y ;).

Ejemplo: Ahorcado

Considere una versión del juego del ahorcado con las siguientes reglas:

- Un jugador entra una palabra secreta
- El otro jugador trata de adivinarla en una secuencia de intentos.
- Para cada intento, la computadora indica cual de las letras en la palabra secreta están en la palabra ingresada.
- El juego termina cuando se adivina la palabra secreta.

basado en transparencias de G. Hutton

Ahorcado (cont.)

Empezamos por la función principal:

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

Ahorcado (cont.)

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


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

Ahorcado (cont.)

La función *adivina* es el bucle principal, que lee y procesa los intentos haste que el juego termina.

```
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
```

Ahorcado (cont.)

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

$$
\begin{aligned}
&\mathit{diff} && :: \mathit{String} \to \mathit{String} \to \mathit{String} \\
&\mathit{diff}\ \mathit{xs}\ \mathit{ys} &&= [\textbf{if}\ \mathit{elem}\ x\ \mathit{ys}\ \textbf{then}\ x\ \textbf{else}\ \texttt{'-'} \mid x \leftarrow \mathit{xs}]
\end{aligned}
$$

Por ejemplo,

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

Resumen de Mónadas

- Modelamos los efectos computacionales con una mónada.
- Preservamos la capacidad de razonar ecuacionalmente.
- La mónada *IO* representa las computaciones que interaccionan con el mundo.
- Usando la notación **do**, podemos escribir programas interactivos en forma similar a los lenguajes imperativos.

Referencias

- Monads for Functional Programming. Philip Wadler (1995)
- Introduction to Functional Programming. Richard Bird (1998)
- The Craft of Functional Programming (2nd ed). Simon Thompson (1999)