# Estructurando código con Mónadas

Mauro Jaskelioff

Jueves 2 de Noviembre, 2017

# Mónadas

- Las mónadas proveen una forma de modelar entrada/salida en lenguajes puros ($IO$).
- También vimos que proveen una generalización de substitución en mónadas de términos.
- En esta clase veremos que son una forma efectiva y versátil de estructurar el código.
- En particular al estructurar el código con mónadas ganamos:
  - Modularidad
  - Reuso
  - Claridad conceptual

# Un evaluador simple

- El AST es:

$$\begin{aligned}\textbf{data}\ \mathit{Exp} = {} & \mathit{Const}\ \mathit{Int} \\ & \mid \mathit{Plus}\ \mathit{Exp}\ \mathit{Exp} \\ & \mid \mathit{Div}\ \ \mathit{Exp}\ \mathit{Exp}\end{aligned}$$

- El evaluador es:

$$\begin{aligned}&\mathit{eval}_1 && :: \mathit{Exp} \to \mathit{Int} \\ &\mathit{eval}_1\ (\mathit{Const}\ n) && = n \\ &\mathit{eval}_1\ (\mathit{Plus}\ t\ u) && = \mathit{eval}_1\ t + \mathit{eval}_1\ u \\ &\mathit{eval}_1\ (\mathit{Div}\ t\ u) && = \mathit{eval}_1\ t\ `\mathit{div}`\ \mathit{eval}_1\ u\end{aligned}$$

- La división puede ocasionar un error en run-time que no manejamos.

# Un evaluador con manejo de error

- Modificamos el evaluador para que maneje el error:

$$
\begin{aligned}
&\mathit{eval}_2 :: \mathit{Exp} \to \mathit{Maybe\ Int}\\
&\mathit{eval}_2\ (\mathit{Const}\ n) = \mathit{Just}\ n\\
&\mathit{eval}_2\ (\mathit{Plus}\ t\ u) = \textbf{case}\ \mathit{eval}_2\ t\ \textbf{of}\\
&\quad \mathit{Nothing} \to \mathit{Nothing}\\
&\quad \mathit{Just}\ m \to \textbf{case}\ \mathit{eval}_2\ u\ \textbf{of}\\
&\qquad\qquad \mathit{Nothing} \to \mathit{Nothing}\\
&\qquad\qquad \mathit{Just}\ n \to \mathit{Just}\ (m+n)\\
&\mathit{eval}_2\ (\mathit{Div}\ t\ u) = \textbf{case}\ \mathit{eval}_2\ t\ \textbf{of}\\
&\quad \mathit{Nothing} \to \mathit{Nothing}\\
&\quad \mathit{Just}\ m \to \textbf{case}\ \mathit{eval}_2\ u\ \textbf{of}\\
&\qquad\qquad \mathit{Nothing} \to \mathit{Nothing}\\
&\qquad\qquad \mathit{Just}\ n \to \textbf{if}\ n \equiv 0\ \textbf{then}\ \mathit{Nothing}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{Just}\ (m\ \text{`}\mathit{div}\text{`}\ n)
\end{aligned}
$$

- El evaluador ya no es tan simple.

# Contando operaciones

- Modificamos el evaluador original para contar divisiones.

$$
\begin{aligned}
&\mathit{eval}_3 && :: \mathit{Exp} \to (\mathit{Int}, \mathit{Int}) \\
&\mathit{eval}_3\ (\mathit{Const}\ n) && = (n, 0) \\
&\mathit{eval}_3\ (\mathit{Plus}\ t\ u) && = \textbf{let}\ (m, cm) = \mathit{eval}_3\ t \\
&&&\qquad\ \ (n, cn)\ \ = \mathit{eval}_3\ u \\
&&&\quad \textbf{in}\ (n + m, cm + cn) \\
&\mathit{eval}_3\ (\mathit{Div}\ t\ u) && = \textbf{let}\ (m, cm) = \mathit{eval}_3\ t \\
&&&\qquad\ \ (n, cn)\ \ = \mathit{eval}_3\ u \\
&&&\quad \textbf{in}\ (n\ `\mathit{div}`\ m, cm + cn + 1)
\end{aligned}
$$

# Agregando variables

- Modificamos el AST para poder tener variables.

$$
\begin{aligned}
&\textbf{type } \mathit{Variable} = \mathit{String}\\
&\textbf{data } \mathit{Exp}_v = \mathit{Const}\ \mathit{Int}\\
&\qquad\qquad\quad \mid \mathit{Var}\ \ \mathit{Variable}\\
&\qquad\qquad\quad \mid \mathit{Plus}\ \mathit{Exp}_v\ \mathit{Exp}_v\\
&\qquad\qquad\quad \mid \mathit{Div}\ \ \mathit{Exp}_v\ \mathit{Exp}_v
\end{aligned}
$$

- El evaluador recibe un entorno:

$$
\begin{aligned}
&\textbf{type } \mathit{Env} = \mathit{Variable} \to \mathit{Int}\\
&\mathit{eval}_4 :: \mathit{Exp}_v \to \mathit{Env} \to \mathit{Int}\\
&\mathit{eval}_4\ (\mathit{Const}\ n)\ \rho = n\\
&\mathit{eval}_4\ (\mathit{Var}\ v)\ \ \rho = \rho\ v\\
&\mathit{eval}_4\ (\mathit{Plus}\ t\ u)\ \rho = \mathit{eval}_4\ t\ \rho + \mathit{eval}_4\ u\ \rho\\
&\mathit{eval}_4\ (\mathit{Div}\ t\ u)\ \rho = \mathit{eval}_4\ t\ \rho\ \text{`}\mathit{div}\text{`}\ \mathit{eval}_4\ u\ \rho
\end{aligned}
$$

# Combinando todo

$$
\begin{aligned}
&\mathit{eval}_5 :: \mathit{Exp}_v \to \mathit{Env} \to \mathit{Maybe}\ (\mathit{Int}, \mathit{Int})\\
&\mathit{eval}_5\ (\mathit{Const}\ n)\ \rho = \mathit{Just}\ (n, 0)\\
&\mathit{eval}_5\ (\mathit{Var}\ v)\ \ \rho = \mathit{Just}\ (\rho\ v, 0)\\
&\mathit{eval}_5\ (\mathit{Plus}\ t\ u)\ \rho = \textbf{case}\ \mathit{eval}_5\ t\ \rho\ \textbf{of}\\
&\quad \mathit{Nothing} \to \mathit{Nothing}\\
&\quad \mathit{Just}\ (m, cm) \to \textbf{case}\ \mathit{eval}_5\ u\ \rho\ \textbf{of}\\
&\qquad \mathit{Nothing} \to \mathit{Nothing}\\
&\qquad \mathit{Just}\ (n, cn) \to \mathit{Just}\ (m + n, cm + cn)\\
&\mathit{eval}_5\ (\mathit{Div}\ t\ u)\ \ \rho = \textbf{case}\ \mathit{eval}_5\ t\ \rho\ \textbf{of}\\
&\quad \mathit{Nothing} \to \mathit{Nothing}\\
&\quad \mathit{Just}\ (m, cm) \to \textbf{case}\ \mathit{eval}_5\ u\ \rho\ \textbf{of}\\
&\qquad \mathit{Nothing} \to \mathit{Nothing}\\
&\qquad \mathit{Just}\ (n, cn) \to \textbf{if}\ n \equiv 0\ \textbf{then}\ \mathit{Nothing}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{Just}\ (m\ \text{`}\mathit{div}\text{`}\ n, cm + cn + 1)
\end{aligned}
$$

# Evaluación de lo hecho

- Para cada modificación hubo que reescribir gran parte del código.
- Incluso cuando las modificaciones sólo afectaban a una pequeña parte:
  - Sólo *Div* puede ocasionar un error;
  - Sólo *Div* agrega algo al contador;
  - Sólo *Var* hace algo con el entorno.
- Conclusión: esta forma de implementar es muy poco modular, dificulta el reuso, y se pierde la idea del programa en detalles.
- Incluso en un lenguaje tan simple el evaluador combinado es muy complejo.

# ¡Mónadas al rescate!

- Los diferentes evaluadores son esencialmente iguales:
  - Realizan una computación que devuelve un entero.
  - Esto lo podemos representar mediante un tipo $m\ Int$ donde $m$ es un constructor de tipos que modela la computación.
- Necesitamos una forma de devolver el valor entero computado (*return*)
- Necesitamos una forma de componer computaciones ($\gg=$).
- Como tipo de retorno usaremos una mónada $m$ que modela el tipo de computación que queremos realizar.

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

# Mónada Identidad

- Modela computaciones puras.

$$
\begin{aligned}
&\textbf{newtype } \mathit{Id}\ a = \mathit{Id}\ a \\
&\mathit{runId} \quad :: \mathit{Id}\ a \to a \\
&\mathit{runId}\ (\mathit{Id}\ x) = x \\
&\textbf{instance } \mathit{Monad}\ \mathit{Id} \textbf{ where} \\
&\quad \mathit{return}\ x \quad = \mathit{Id}\ x \\
&\quad \mathit{Id}\ x \gg\!= f = f\ x
\end{aligned}
$$

- La mónada Identidad no agrega nada, pero funciona como tipo abstracto de datos, que expone sólo la interfaz de la mónada.

# Evaluador monádico

- Reimplementamos $eval_1$ con la mónada identidad

$$
\begin{aligned}
&evalM_1 && :: Exp \to Id\ Int \\
&evalM_1\ (Const\ n) && = return\ n \\
&evalM_1\ (Plus\ t\ u) && = \mathbf{do}\ m \leftarrow evalM_1\ t \\
&&& \qquad\quad n \leftarrow evalM_1\ u \\
&&& \qquad\quad return\ (m + n) \\
&evalM_1\ (Div\ t\ u) && = \mathbf{do}\ m \leftarrow evalM_1\ t \\
&&& \qquad\quad n \leftarrow evalM_1\ u \\
&&& \qquad\quad return\ (m\ `div`\ n)
\end{aligned}
$$

- Notar que $evalM_1$ sólo usa las operaciones *return* y $(\gg=)$.
- Es más, el tipo más general del evaluador es:

$$evalM_1 :: Monad\ m \Rightarrow Exp \to m\ Int$$

- Ejercicio: Probar que $eval_1 = runId \circ evalM_1$.

# Mónada *Maybe*

- Modela computaciones que pueden fallar

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

$$\begin{aligned}&\textbf{instance}\ \mathit{Monad}\ \mathit{Maybe}\ \textbf{where}\\ &\quad \mathit{return}\ a \qquad\quad = \mathit{Just}\ a\\ &\quad \mathit{Nothing} \gg\!= f = \mathit{Nothing}\\ &\quad \mathit{Just}\ x \gg\!= f \quad = f\ x\end{aligned}$$

- Posee una operación con acceso a la representación interna que permite señalar errores.

$$\begin{aligned}&\mathit{throw} :: \mathit{Maybe}\ a\\ &\mathit{throw} = \mathit{Nothing}\end{aligned}$$

# Evaluador Monádico con manejo de error

- El evaluador monádico con manejo de errores es:

$$
\begin{aligned}
&\mathit{evalM}_2 && :: \mathit{Exp} \to \mathit{Maybe}\ \mathit{Int}\\
&\mathit{evalM}_2\ (\mathit{Const}\ n) && = \mathit{return}\ n\\
&\mathit{evalM}_2\ (\mathit{Plus}\ t\ u) && = \mathbf{do}\ m \leftarrow \mathit{evalM}_2\ t\\
&&&\qquad\ n \leftarrow \mathit{evalM}_2\ u\\
&&&\qquad\ \mathit{return}\ (m + n)\\
&\mathit{evalM}_2\ (\mathit{Div}\ t\ u) && = \mathbf{do}\ m \leftarrow \mathit{evalM}_2\ t\\
&&&\qquad\ n \leftarrow \mathit{evalM}_2\ u\\
&&&\qquad\ \mathbf{if}\ n \equiv 0\ \mathbf{then}\ \mathit{throw}\\
&&&\qquad\qquad\ \mathbf{else}\ \mathit{return}\ (m\ `\mathit{div}`\ n)
\end{aligned}
$$

- El evaluador sólo utiliza las operaciones *return*, $(\gg=)$ y *throw*.
- Con respecto a $\mathit{evalM}_1$ sólo modificamos el chequeo de división por 0.
- Ejercicio: Probar que $\mathit{eval}_2 = \mathit{evalM}_2$.

# Mónada de acumulación

- Modela computaciones que llevan un acumulador

$$
\begin{aligned}
&\textbf{newtype } \mathit{Acum}\ a = \mathit{Ac}\ (a, \mathit{Int}) \\
&\mathit{runAcum} \qquad\quad :: \mathit{Acum}\ a \to (a, \mathit{Int}) \\
&\mathit{runAcum}\ (\mathit{Ac}\ p) = p \\
&\textbf{instance } \mathit{Monad}\ \mathit{Acum} \textbf{ where} \\
&\quad \mathit{return}\ x \qquad\quad = \mathit{Ac}\ (x, 0) \\
&\quad \mathit{Ac}\ (x, n) \gg\!= f = \textbf{let } \mathit{Ac}\ (x', n') = f\ x \\
&\qquad\qquad\qquad\qquad\quad \textbf{in } \mathit{Ac}\ (x', n + n')
\end{aligned}
$$

- Posee una operación para sumar 1 al acumulador.

$$
\begin{aligned}
&\mathit{tick} :: \mathit{Acum}\ () \\
&\mathit{tick} = \mathit{Ac}\ ((), 1)
\end{aligned}
$$

# Evaluador monádico con contador

- El evaluador monádico con contador es:

$$
\begin{aligned}
&evalM_3 && :: Exp \to Acum\ Int \\
&evalM_3\ (Const\ n) && = return\ n \\
&evalM_3\ (Plus\ t\ u) && = \mathbf{do}\ m \leftarrow evalM_3\ t \\
&&&\qquad\ n \leftarrow evalM_3\ u \\
&&&\qquad\ return\ (m + n) \\
&evalM_3\ (Div\ t\ u) && = \mathbf{do}\ m \leftarrow evalM_3\ t \\
&&&\qquad\ n \leftarrow evalM_3\ u \\
&&&\qquad\ tick \\
&&&\qquad\ return\ (m\ `div`\ n)
\end{aligned}
$$

- El evaluador sólo utiliza las operaciones *return*, $(\gg=)$ y *tick*.
- Con respecto a $evalM_1$ sólo agregamos un *tick* para contar la operación de división
- Ejercicio: Probar que $eval_3 = runAcum \circ evalM_3$.

# Mónada de entorno

- Modela computaciones que llevan un entorno

$$\textbf{newtype}\ \mathit{Reader}\ a = \mathit{Reader}\ (\mathit{Env} \to a)$$

$$\mathit{runReader} :: \mathit{Reader}\ a \to \mathit{Env} \to a$$
$$\mathit{runReader}\ (\mathit{Reader}\ h) = h$$

$$\textbf{instance}\ \mathit{Monad}\ \mathit{Reader}\ \textbf{where}$$
$$\quad \mathit{return}\ x \qquad\quad = \mathit{Reader}\ (\lambda\_ \to x)$$
$$\quad \mathit{Reader}\ h \gg\!= f = \mathit{Reader}\ (\lambda\rho \to \mathit{runReader}\ (f\ (h\ \rho))\ \rho)$$

- Posee una operación para obtener el entorno

$$\mathit{ask} :: \mathit{Reader}\ \mathit{Env}$$
$$\mathit{ask} = \mathit{Reader}\ (\lambda\rho \to \rho)$$

# Evaluador monádico con variables

- El evaluador monádico con variables es:

$$
\begin{aligned}
&\mathit{evalM4} && :: \mathit{Exp}_v \to \mathit{Reader\ Int}\\
&\mathit{evalM4}\ (\mathit{Const}\ n) && = \mathit{return}\ n\\
&\mathit{evalM4}\ (\mathit{Var}\ v) && = \mathbf{do}\ \rho \leftarrow \mathit{ask}\\
&&& \quad\ \mathit{return}\ (\rho\ v)\\
&\mathit{evalM4}\ (\mathit{Plus}\ t\ u) && = \mathbf{do}\ m \leftarrow \mathit{evalM4}\ t\\
&&& \quad\ n \leftarrow \mathit{evalM4}\ u\\
&&& \quad\ \mathit{return}\ (m + n)\\
&\mathit{evalM4}\ (\mathit{Div}\ t\ u) && = \mathbf{do}\ m \leftarrow \mathit{evalM4}\ t\\
&&& \quad\ n \leftarrow \mathit{evalM4}\ u\\
&&& \quad\ \mathit{return}\ (m\ `\mathit{div}`\ n)
\end{aligned}
$$

- El evaluador sólo utiliza las operaciones *return*, $(\gg\!=)$ y *ask*.
- Con respecto a $\mathit{evalM}_1$ sólo agregamos el pattern *Var*. El resto de las líneas es exactamente igual.
- Ejercicio: Probar que para todo $t$, $\mathit{eval}_4\ t = \mathit{runReader}\ (\mathit{evalM4}\ t)$.

# Evaluador monádico combinado

- El evaluador combinado queda:

$$
\begin{aligned}
&\mathit{evalM5} && :: \mathit{Exp}_v \to M\ \mathit{Int}\\
&\mathit{evalM5}\ (\mathit{Const}\ n) && = \mathit{return}\ n\\
&\mathit{evalM5}\ (\mathit{Var}\ v) && = \mathbf{do}\ \rho \leftarrow \mathit{ask}\\
&&&\quad \mathit{return}\ (\rho\ v)\\
&\mathit{evalM5}\ (\mathit{Plus}\ t\ u) && = \mathbf{do}\ m \leftarrow \mathit{evalM5}\ t\\
&&&\quad n \leftarrow \mathit{evalM5}\ u\\
&&&\quad \mathit{return}\ (m + n)\\
&\mathit{evalM5}\ (\mathit{Div}\ t\ u) && = \mathbf{do}\ m \leftarrow \mathit{evalM5}\ t\\
&&&\quad n \leftarrow \mathit{evalM5}\ u\\
&&&\quad \mathit{tick}\\
&&&\quad \mathbf{if}\ n \equiv 0\ \mathbf{then}\ \mathit{throw}\\
&&&\qquad\qquad \mathbf{else}\ \mathit{return}\ (m\ `\mathit{div}`\ n)
\end{aligned}
$$

- Sólo hace falta una mónada $M$ con *throw*, *tick*, y *ask*.

# Mónada para el evaluador combinado

$$\textbf{newtype}\ M\ a = M\ (Env \to Maybe\ (a, Int))$$

$$runM \quad :: M\ a \to Env \to Maybe\ (a, Int)$$
$$runM\ (M\ x) = x$$

$$\textbf{instance}\ Monad\ M\ \textbf{where}$$
$$\begin{aligned}
&return\ x \quad = M\ (\backslash_ \to Just\ (x, 0)) \\
&M\ h \gg\!= f = M\ (\lambda\rho \to \textbf{case}\ h\ \rho\ \textbf{of} \\
&\qquad\qquad Nothing \quad \to Nothing \\
&\qquad\qquad Just\ (a, m) \to \textbf{case}\ runM\ (f\ a)\ \rho\ \textbf{of} \\
&\qquad\qquad\qquad\qquad Nothing \to Nothing \\
&\qquad\qquad\qquad\qquad Just\ (b, n) \to Just\ (b, m + n))
\end{aligned}$$

$$throw :: M\ a$$
$$throw = M\ (\backslash\_ \to Nothing)$$

$$tick :: M\ ()$$
$$tick = M\ (\backslash\_ \to Just\ ((), 1))$$

$$ask \quad :: M\ Env$$
$$ask \quad = M\ (\lambda\rho \to Just\ (\rho, 0))$$

# Observaciones

- Las mónadas capturan una gran cantidad de efectos, hasta acá vimos, errores, entornos y acumulación.
  - En la práctica veremos algunos más.
- Los evaluadores monádicos toman argumentos puros, pero devuelven un valor en una mónada.
- Las funciones que *usan* la mónada sólo usan la interfaz, es decir: *return*, $(\gg=)$ y operaciones propias de la mónada.
- Podemos hacer los evaluadores más generales definiendo clases de mónada que soportan determinadas operaciones.

$$\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadThrow}\ m\ \textbf{where}$$
$$\quad \mathit{throw} :: m\ a$$

Luego el evaluador con manejo de errores queda definido para cualquier mónada que implemente *throw*:

$$\mathit{evalM}_2 :: \mathit{MonadThrow}\ m \Rightarrow \mathit{Exp} \rightarrow m\ \mathit{Int}$$

# Más Observaciones

- En versiones recientes de GHC, se requiere que toda instancia de mónada sea de la clase *Applicative*, y que todo *Applicative* sea *Functor*.
- Para dejar tranquilo al compilador podemos agregar a toda instancia de mónada $M$:

```
instance Functor M where
  fmap = liftM
instance Applicative M where
  pure  = return
  (<*>) = ap
```

posiblemente con los siguiente imports

```
import Control.Applicative (Applicative (..))
import Control.Monad (liftM, ap)
```

# Resumen

- Las mónadas son una abstracción muy útil para estructurar código.
- Permiten la programación de código más modular y reusable.
- Su uso no se limita a intérpretes. (aunque casi cualquier cosa puede ser vista como un intérprete)
- Su uso no se limita a Haskell (aunque en general se usa en lenguajes con alto orden).
- Las mónadas proveen un ; programable
- No son las únicas estructuras que se utilizan (ver por ej. functores aplicativos y arrows)

# 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)