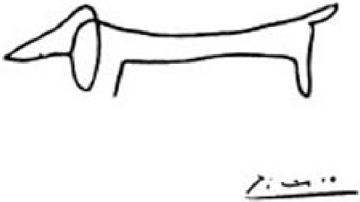
Abstracciones: Functores y Mónadas

Mauro Jaskelioff

Viernes 13 de Octubre, 2017



¿Por qué abstraer?

- Al programar debemos expresar **ideas complejas**.
- Manejamos esta complejidad es mediante **abstracciones**.
- Mediante abstracciones logramos:
  - Simplificar la escritura de programas
  - Hacer programas más entendibles
  - Facilitar el reuso y la verificación
- Las abstracciones nos proveen un

  lenguaje para comunicar ideas.

Abstracciones como lenguaje

- Usar abstracciones ad-hoc dificulta la comunicación.
- Para que la comunicación sea efectiva la abstracción debe ser conocida tanto por la persona que la escribe, como por la que lo interpreta.
- Algunas abstracciones que ya pertenecen a este inconsciente colectivo de los programadores:
  - Aritmética
  - Funciones
  - Tuplas
  - Grafos
- ¿Qué requerimientos debe cumplir una abstracción para ser considera **buena**?

Buenas abstracciones

Consideramos buenas abstracciones aquellas con:

- **Definición**. Debe ser precisa.
- **Teoría**. Resultados. Teoremas y propiedades generales.
- **Generalidad**. el concepto debe ser útil en diversas situaciones.

Recientemente, la teoría de categorías nos ha provisto de buenas abstracciones.

- Hoy veremos dos ejemplos: funtores y mónadas.

Generalizando *map*

- Generalizando $map :: (a \to b) \to [a] \to [b]$
  - $\mathbf{data}\ Bin\ a = Leaf\ \ a \mid Node\ (Bin\ a)\ (Bin\ a)$
    $binmap :: (a \to b) \to Bin\ a \to Bin\ b$
  - $\mathbf{data}\ Tree\ a = Empty \mid Branch\ a\ (Tree\ a)\ (Tree\ a)$
    $treemap :: (a \to b) \to Tree\ a \to Tree\ b$
  - $\mathbf{data}\ GenTree\ a = Gen\ a\ [GenTree\ a]$
    $gentreemap :: (a \to b) \to GenTree\ a \to GenTree\ b$
- *map* no se siempre se puede generalizar.

$$\mathbf{data}\ Func\ a = Func\ (a \to a)$$

$$funcmap :: (a \to b) \to Func\ a \to Func\ b$$
$$funcmap\ g\ (Func\ h) = ???$$

Generalizando *map* (cont.)

- Intuitivamente, podremos generalizar *map* a un *constructor de tipos* f, si podemos proveer una función

$$fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$$

  que aplique el argumento a todos los elementos de a almacenados en $f\ a$.

- No toda función con este tipo es una generalización de *map*.

$$\begin{aligned}
&\mathbf{data}\ Func\ a = Func\ (a \rightarrow a) \\
&funcmap :: (a \rightarrow b) \rightarrow Func\ a \rightarrow Func\ b \\
&funcmap\ g\ (Func\ h) = Func\ id
\end{aligned}$$

Functores

- Los constructores de tipos que poseen una función *fmap* con "buen comportamiento" son *functores*.
- El "buen comportamiento" queda especificado por las siguientes ecuaciones:

$$fmap\ id = id \tag{functor.1}$$

$$fmap\ f \circ fmap\ g = fmap\ (f \circ g) \tag{functor.2}$$

Ejercicio (Probar que (*Func*, *funcmap*) no es un functor)

$$\textbf{data}\ Func\ a = Func\ (a \to a)$$

$$funcmap :: (a \to b) \to Func\ a \to Func\ b$$
$$funcmap\ g\ (Func\ h) = Func\ id$$

Functores en Haskell

- En Haskell, expresamos el requerimiento mediante una *clase* de tipos.

  class *Functor f* **where**
  $\quad fmap :: (a \to b) \to f\ a \to f\ b$

- Por ejemplo, podemos definir las siguientes instancias

  instance *Functor* [] **where**
  $\quad fmap\ f\ xs = map\ f\ xs$

  instance *Functor Bin* **where**
  $\quad fmap\ f\ (Leaf\ a) = Leaf\ (f\ a)$
  $\quad fmap\ f\ (Bin\ l\ r) = Bin\ (fmap\ f\ l)\ (fmap\ f\ r)$

Functores en Haskell (cont.)

Observaciones:

- Las ecuaciones ($functor.1$) y ($functor.2$) no pueden ser verificadas por el intérprete/compilador.
- Es responsabilidad del programador hacerlo.
- En las versiones modernas de GHC se puede usar la extensión `-XDeriveFunctor`, y escribir, por ejemplo:

$$\textbf{data}\ T\ a = E \mid N\ (T\ a)\ a\ (T\ a)\ \textbf{deriving}\ Functor$$

Para obtener automáticamente la instancia de la clase *Functor*

Resumen Functores

- Un functor tiene tres componentes:
  1. el constructor de tipos $F : * \to *$;
  2. la función $fmap :: (a \to b) \to (F\ a \to F\ b)$;
  3. y las pruebas de las ecuaciones $(functor.1)$ y $(functor.2)$.
- Decimos "F es un functor" para indicar que tenemos las **tres componentes**.
- Intuitivamente, un functor es una abstracción de todos los constructores de tipos que almacenan información de su argumento de tipo.

Mónadas

Un lenguaje para aritmética

$$t ::= n \mid t + t \mid t \ \times \ t \mid t - t$$

- El AST del lenguaje aritmético en Haskell es:

$$\begin{aligned}\textbf{data}\ A_{\emptyset} = {} & \mathit{Num}\ \mathit{Int} \\ \mid {} & \mathit{Sum}\ A_{\emptyset}\ A_{\emptyset} \\ \mid {} & \mathit{Mul}\ \ A_{\emptyset}\ A_{\emptyset} \\ \mid {} & \mathit{Res}\ \ A_{\emptyset}\ A_{\emptyset}\end{aligned}$$

Agregando Variables Libres

- Consideremos el mismo lenguaje con variables libres de tipo a.

$$
\begin{aligned}
\textbf{data}\ A\ a = {} & \mathit{Num}\ \mathit{Int} \\
\mid {} & \mathit{Sum}\ (A\ a)\ (A\ a) \\
\mid {} & \mathit{Mul}\ \ (A\ a)\ (A\ a) \\
\mid {} & \mathit{Res}\ \ (A\ a)\ (A\ a) \\
\mid {} & \mathit{Var}_A\ a
\end{aligned}
$$

- $3 + x \quad \mapsto \quad \mathit{Sum}\ (\mathit{Num}\ 3)\ (\mathit{Var}\ \texttt{"x"}) :: A\ \mathit{String}$
- Comparado con el tipo $A_\emptyset$,
  - parametrizamos con variables en a;
  - agregamos un constructor para insertar variables.

$$\mathit{Var}_A :: a \to A\ a$$

Substitución simultánea

- Considere el término $y + x$.
- Asignamos a y el término $x \times 2$ (la variable x decidimos dejarla como está).
- $(y + x)\ [y \mapsto x \times 2] = x \times 2 + x$
- Este mapeo de variables a términos se puede implementar en Haskell con la siguiente función:

$$\begin{aligned} &g && :: String \to A\ String \\ &g\ \texttt{"y"} && = Mul\ (Var_A\ \texttt{"x"})\ (Num\ 2) \\ &g\ v && = Var_A\ v \end{aligned}$$

- En general, una regla de substitución de variables en a por términos con variables en b, es una función:

$$a \to A\ b$$

Implementando substitución simultánea

- Implementamos substitución simultánea de la siguiente manera:

$$(\gg\!=_A) :: A\ a \to (a \to A\ b) \to A\ b$$

$$\begin{aligned}
\mathit{Num}\ n\ \ &\gg\!=_A v = \mathit{Num}\ n \\
\mathit{Sum}\ t\ u &\gg\!=_A v = \mathit{Sum}\ (t \gg\!=_A v)\ (u \gg\!=_A v) \\
\mathit{Mul}\ t\ u &\gg\!=_A v = \mathit{Mul}\ (t \gg\!=_A v)\ (u \gg\!=_A v) \\
\mathit{Res}\ t\ u &\gg\!=_A v = \mathit{Res}\ (t \gg\!=_A v)\ (u \gg\!=_A v) \\
\mathit{Var}_A\ a\ \ &\gg\!=_A v = v\ a
\end{aligned}$$

- La substitución g en el término $y + x$ es:

$$\begin{aligned}
&\mathit{Sum}\ (\mathit{Var}_A\ \texttt{"y"})\ (\mathit{Var}_A\ \texttt{"x"}) \gg\!=_A g = \\
&\quad \mathit{Sum}\ (\mathit{Mul}\ (\mathit{Var}_A\ \texttt{"x"})\ (\mathit{Num}\ 2))\ (\mathit{Var}_A\ \texttt{"x"})
\end{aligned}$$

Propiedades de Substitución

- $(\ggcurly_A.1)$

$$Var_A\, x \ggcurly_A f \quad = \quad f\ x$$

- $(\ggcurly_A.2)$

$$t \ggcurly_A Var_A \quad = \quad t$$

- $(\ggcurly_A.3)$

$$(t \ggcurly_A f) \ggcurly_A g \quad = \quad t \ggcurly_A (\lambda x \rightarrow f\ x \ggcurly_A g)$$

Un poco de práctica

Ejercicio

Dar el tipo y definición de g y h tal que

$$\begin{array}{l} \mathit{Sum}\ (\mathit{Var}\ \texttt{"x"})\ (\mathit{Mul}\ (\mathit{Var}\ \texttt{"y"})\ (\mathit{Var}\ \texttt{"z"})) \ggg_A g = \\ \quad \mathit{Sum}\ (\mathit{Var}\ 1)\ (\mathit{Mul}\ (\mathit{Res}\ (\mathit{Var}\ 3)\ (\mathit{Var}\ 1)) \\ \qquad\qquad\qquad\qquad (\mathit{Mul}\ (\mathit{Var}\ 2)\ (\mathit{Var}\ 2))) \end{array}$$

$$\begin{array}{l} \mathit{Sum}\ (\mathit{Var}\ \texttt{"x"})\ (\mathit{Mul}\ (\mathit{Var}\ \texttt{"y"})\ (\mathit{Var}\ \texttt{"z"})) \ggg_A h = \\ \quad \mathit{Sum}\ (\mathit{Var}\ 1)\ (\mathit{Res}\ (\mathit{Var}\ 2)\ (\mathit{Var}\ 3)) \end{array}$$

¿Cuántas soluciones existen en cada caso?

Lenguaje Botón

- Definimos el AST de un lenguaje para controlar un botón.

$$\begin{aligned}\textbf{data}\ BT_\emptyset &= \mathit{IfBoton}\ BT_\emptyset\ BT_\emptyset\\ &\mid\ \mathit{Beep}\ BT_\emptyset\end{aligned}$$

- La idea es que:
  - *IfBoton* x y ejecuta x si el botón está presionado, o y si no lo está.
  - *Beep* x, emite un *beep* y sigue con la ejecución de x.

Ejercicio

¿Qué hacen los siguientes programas?

- $t_1 = \mathit{IfBoton}\ (\mathit{Beep}\ (\mathit{Beep}\ t_1))\ t_1$
- $t_2 = \mathit{ifBoton}\ t_2\ (\mathit{Beep}\ t_2)$

Lenguaje Botón (cont.)

- Equivalentemente podemos definir el mismo AST como:

$$\begin{aligned}\mathbf{data}\ BT_\emptyset &= \mathit{IfBoton}\ (\mathit{Bool} \to BT_\emptyset)\\ &\mid\ \mathit{Beep}\ BT_\emptyset\end{aligned}$$

- Esto es equivalente al anterior.
- Los programas de ejemplo quedan:

$$\begin{aligned}t_1 &= \mathit{IfBoton}\ (\lambda b \to \mathbf{if}\ b\ \mathbf{then}\ \mathit{Beep}\ (\mathit{Beep}\ t_1)\ \mathbf{else}\ t_1)\\ t_2 &= \mathit{ifBoton}\ (\lambda b \to \mathbf{if}\ b\ \mathbf{then}\ t_2\ \mathbf{else}\ \mathit{Beep}\ t_2)\end{aligned}$$

Agregando Variables Libres

- Consideremos el mismo lenguaje con variables libres de tipo a.

$$
\begin{aligned}
\textbf{data}\ BT\ a = {} & \mathit{IfBoton}\ (\mathit{Bool} \to BT\ a) \\
\mid {} & \mathit{Beep}\ (BT\ a) \\
\mid {} & \mathit{Var}_{BT}\ a
\end{aligned}
$$

- Comparado con el tipo $BT_{\emptyset}$,
  - parametrizamos con variables en a;
  - agregamos un constructor para insertar variables.

$$\mathit{Var}_{BT} :: a \to BT\ a$$

Implementando substitución

- Implementamos substitución simultánea

$$
\begin{aligned}
&(\gg\!=_{BT}) :: BT\ a \to (a \to BT\ b) \to BT\ b \\
&\mathit{IfBoton}\ f \gg\!=_{BT} v = \mathit{IfBoton}\ (\lambda b \to f\ b \gg\!=_{BT} v) \\
&\mathit{Beep}\ t \quad\ \ \gg\!=_{BT} v = \mathit{Beep}\ (t \gg\!=_{BT} v) \\
&\mathit{Var}_{BT}\ a\ \ \gg\!=_{BT} v = v\ a
\end{aligned}
$$

- Ejemplo: substituir *Bool* por un término cerrado:

$$
\begin{aligned}
&g && :: \mathit{Bool} \to BT\ a \\
&g\ \mathit{True} && = t_1 \\
&g\ \mathit{False} && = t_2
\end{aligned}
$$

$$
\begin{aligned}
& \quad \mathit{IfBoton}\ (\lambda b \to \mathbf{if}\ b\ \mathbf{then}\ \mathit{Var}_{BT}\ \mathit{False}\ \mathbf{else}\ \mathit{Var}_{BT}\ \mathit{True}) \gg\!=_{BT} g \\
&= \\
& \quad \mathit{IfBoton}\ (\lambda b \to \mathbf{if}\ b\ \mathbf{then}\ t_2\ \mathbf{else}\ t_1)
\end{aligned}
$$

Propiedades de Substitución

- $(\ggg_{BT}.1)$
$$Var_{BT}\ x \ggg_{BT} f \quad = \quad f\ x$$
- $(\ggg_{BT}.2)$
$$t \ggg_{BT} Var_{BT} \quad = \quad t$$
- $(\ggg_{BT}.3)$
$$(t \ggg_{BT} f) \ggg_{BT} g \quad = \quad t \ggg_{BT} (\lambda x \rightarrow f\ x \ggg_{BT} g)$$

Programando un botón

- Ejemplo: el siguiente programa hace n beeps:

$$
\begin{aligned}
&\mathit{beep} :: \mathit{Int} \to \mathit{BT}\ ()\\
&\mathit{beep}\ 0 = \mathit{Var}_{\mathit{BT}}\ ()\\
&\mathit{beep}\ n = \mathit{Beep}\ (\mathit{beep}\ (n-1))
\end{aligned}
$$

- Ejemplo, el siguiente programa hace tantos beeps como veces se haya pulsado el bóton.

$$
\begin{aligned}
&\mathit{ej} :: \mathit{BT}\ a\\
&\mathit{ej} = \mathit{ejAux}\ 0\\
&\quad \textbf{where}\ \mathit{ejAux}\ n = \mathit{ifBoton}\ (\lambda b \to \textbf{if}\ b\\
&\qquad \textbf{then}\ \mathit{beep}\ n \gg\!= \lambda() \to \mathit{ejAux}\ (n+1)\\
&\qquad \textbf{else}\ \mathit{ejAux}\ n)
\end{aligned}
$$

- Ejercicio: Escribir un programa que haga un beep cada dos pulsaciones de botón.

Entrada/Salida

- Definimos el AST de un lenguaje para entrada/salida.
- Lee un caracter de entrada y en base a eso decide como seguir o escribe un caracter y continua la ejecución.

$$\begin{aligned} \textbf{data } ES_{\emptyset} &= Read\ (Char \to ES_{\emptyset}) \\ &\mid\ Write\ Char\ ES_{\emptyset} \end{aligned}$$

Ejercicio

¿Qué hacen los siguientes programas?

- $t_1 = Read\ (\lambda c \to Write\ c\ (Write\ c\ t_1))$
- $t_2 = Read\ (\lambda c \to Write\ \texttt{"("}\ (Write\ c\ (Write\ \texttt{")"}\ t_2)))$
- $t_3 = Write\ \texttt{"("}\ (Read\ (\lambda c \to Write\ c\ (Write\ \texttt{")"}\ t_3)))$
- $t_4 = Read\ (\backslash\_ \to t_4)$

Agregando Variables Libres

- Consideremos el mismo lenguaje con variables libres de tipo a.

$$
\begin{aligned}
\textbf{data}\ ES\ a = \ & Read\ (Char \rightarrow ES\ a) \\
\mid \ & Write\ Char\ (ES\ a) \\
\mid \ & Var_{ES}\ a
\end{aligned}
$$

- Comparado con el tipo $ES_{\emptyset}$,
  - parametrizamos con variables en a;
  - agregamos un constructor para insertar variables.

$$Var_{ES} :: a \rightarrow ES\ a$$

Implementando substitución

- Implementamos substitución simultánea

$$
\begin{aligned}
&(\ggg\!=_{ES}) :: ES\ a \to (a \to ES\ b) \to ES\ b \\
&Read\ k \quad \ggg\!=_{ES} v = Read\ (\lambda c \to k\ c \ggg\!=_{ES} v) \\
&Write\ c\ t \ggg\!=_{ES} v = Write\ c\ (t \ggg\!=_{ES} v) \\
&Var_{ES}\ a \ \ \ggg\!=_{ES} v = v\ a
\end{aligned}
$$

- Ejemplo: substituir *Bool* por un término cerrado:

$$
\begin{aligned}
&g \qquad\quad :: Bool \to ES\ a \\
&g\ True\ = t_1 \\
&g\ False = t_4
\end{aligned}
$$

$$
Read\ (\lambda c \to Var_{ES}\ False) \ggg\!=_{ES} g = Read\ (\lambda c \to t_4)
$$

Propiedades de Substitución

- $(\ggcurly_{ES}.1)$
$$Var_{ES}\ x \ggcurly_{ES} f \quad = \quad f\ x$$
- $(\ggcurly_{ES}.2)$
$$t \ggcurly_{ES} Var_{ES} \quad = \quad t$$
- $(\ggcurly_{ES}.3)$
$$(t \ggcurly_{ES} f) \ggcurly_{ES} g \quad = \quad t \ggcurly_{ES} (\lambda x \rightarrow f\ x \ggcurly_{ES} g)$$

Ejercicios

Recordemos

$$\textbf{data}\ ES\ a = Var_{ES}\ a \mid Read\ (Char \to ES\ a) \mid Write\ Char\ (ES\ a)$$

Ejercicio

Escribir un programa writeChar $:: Char \to ES\ ()$ *que escriba su argumento y finalice con una variable* $()$.

Ejercicio

Escribir un programa readChar $:: ES\ Char$ *que lea un caracter y finalice con la variable cuyo nombre es el caracter leído.*

Ejercicio

Usando writeChar, escribir un programa writeStr $:: String \to ES\ ()$ *que imprima la cadena que se pasa como argumento.*

Usando $\ggg=_{ES}$ y Var_{ES}

- Podemos usar las variables para pasar información, usando $\ggg=_{ES}$ y Var_{ES}.

Ejercicio

¿Qué hace el siguiente programa?

$$
\begin{array}{l}
f :: ES\ String \\
f = readChar \ggg=_{ES} g \\
\quad \textbf{where}\ g\ \texttt{'\textbackslash n'} = Var_{ES}\ [\,] \\
\qquad\qquad g\ c \qquad\ = f \ggg=_{ES} \lambda xs \rightarrow Var_{ES}\ (c : xs)
\end{array}
$$

Computaciones y Valores

- Un término de tipo *ES String*, ejecuta una serie de operaciones de entrada/salida y después devuelve un valor.
- Pensamos las variables como el valor que resulta de una computación.
- Una función $f :: a \to ES\ b$ es una función que toma un valor a, y devuelve un valor b, luego de haber ejecutado algunas operaciones en *Read* y *Write*.
- El tipo *ES a* me representa los programas que luego de ejecutar algunos comandos de entrada y salida me devuelven un valor de tipo a.

Secuenciando operaciones

- En lenguajes imperativos las operaciones se componen con un operador de secuencia.
- Implementamos un operador de secuencia usando $\gg\!=_{ES}$.

$$
\begin{aligned}
&(\gg_{ES}) \quad :: ES\ a \to ES\ b \to ES\ b\\
&t \gg_{ES} u = t \gg\!=_{ES} \backslash\_ \to u
\end{aligned}
$$

- Reescribimos *writeStr*:

$$
\begin{aligned}
&writeStr &&:: String \to ES\ ()\\
&writeStr\ [\,] &&= Var\ ()\\
&writeStr\ (x : xs) &&= writeChar\ x \gg_{ES} writeStr\ xs
\end{aligned}
$$

Mónadas

- La clase *Monad* clasifica los constructores de tipos con una (generalización de la) noción de inserción de variables y substitución.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{return} :: a \rightarrow m\ a \\
\quad (\gg\!=)\ :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b
\end{array}
$$

- $(\gg\!=)$ se pronuncia *bind*.
- La instancia para *ES* es:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ \mathit{ES}\ \textbf{where} \\
\quad \mathit{return} = \mathit{Var}_{\mathit{ES}} \\
\quad (\gg\!=) = (\gg\!=_{\mathit{ES}})
\end{array}
$$

Leyes de las mónadas

Las instancias de *Monad* deben verificar las siguientes ecuaciones:

- $(monad.1)$
$$return\ x \ggg f \quad = \quad f\ x$$

- $(monad.2)$
$$t \ggg return \quad = \quad t$$

- $(monad.3)$
$$(t \ggg f) \ggg g \quad = \quad t \ggg (\lambda x \rightarrow f\ x \ggg g)$$

Ejercicios

Ejercicio

Probar que ES es una mónada. Es decir, probar que $\gg\!=_{ES}$ *y Var cumplen con las leyes de las mónadas.*

Ejercicio

Probar que toda mónada es un functor, es decir, proveer una instancia

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Functor}\ m\ \textbf{where} \\ \quad \mathit{fmap}\ \ldots\end{array}$$

y probar que las leyes de los functores se cumplen para su definición de fmap.

Resumen

- Podemos ver las mónadas como constructores de tipos con una noción de inserción de valores y substitución.
- Podemos entender una computación monádica como una secuencia de operaciones que retorna un valor.

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)