(Vigesimotercera clase: Mónadas en la programación)

Las mónadas en programación están muy bien explicadas en numerosos trabajos científicos, por ejemplo, "The essence of functional programming" de Philip Wadler (1992).

En el ejemplo que sigue, usaré la notación que se obtiene con las mónadas como las hemos definido, es decir, una mónada es un funtor $T : \mathbf{C} \longrightarrow \mathbf{C}$ junto a dos transformaciones naturales $\eta : 1_{\mathbf{C}} \longrightarrow T$ y $\mu : T \circ T \longrightarrow T$ tales que ciertos diagramas, que expresan la asociatividad de la "multiplicación" μ y la "neutralidad" de η, conmutan.

Las mónadas han sido utilizadas con mucho éxito en programación funcional para obtener código más reusable. Sigue un ejemplo.

Sea el sencillo lenguaje de expresiones dado por la gramática

$$\begin{array}{rcll} < exp > & ::= & i \qquad i \in \mathbb{Z} \\ & | & < exp > + < exp > \\ & | & < exp > - < exp > \\ & | & < exp > * < exp > \end{array}$$

Si queremos hacer un evaluador en Haskell para este lenguaje, primero escribiríamos el tipo de estas expresiones

```
data Exp = Val Integer
         | Mas Exp Exp
         | Menos Exp Exp
         | Por Exp Exp
```

El evaluador, por supuesto, resulta muy sencillo:

```
eval :: Exp → Integer
eval (Val i) = i
eval (Mas a b) = eval a + eval b
eval (Menos a b) = eval a - eval b
eval (Por a b) = eval a * eval b
```

Si más adelante se quiere extender el lenguaje, es posible descubrir que es necesario reescribir todo el programa. Por ejemplo, si agregamos al lenguaje la división:

$$\begin{array}{rcl} < exp > & ::= & \ldots \\ & | & < exp > / < exp > \end{array}$$

eso dará lugar a la modificación correspondiente en la definición del tipo Exp:

```
data Exp = ...
         | Div Exp Exp
```

Pero el evaluador no es tan sencillo, si simplemente agragáramos la línea de código

```
eval (Div a b) = eval a ‘div‘ eval b
```

el programa daría un runtime error al evaluar, por ejemplo, Div (Val 1) (Val 0). Es deseable que el evaluador sea robusto (por ejemplo, si se quiere luego agregar algún mecanismo de manejo de excepciones). Esto sí nos lleva a reescribir todo el evaluador. Sea Maybe el constructor de tipos definido por

```
data Maybe a = Just a | Nothing
```

el evaluador se reescribe como sigue

```
eval :: Exp → Maybe Integer
eval (Val i) = Just i
eval (Mas a b) = case eval a of
                   Just x → case eval b of
                              Just y → Just (x + y)
                              Nothing → Nothing
                   Nothing → Nothing
```

y de manera similar para los casos correspondientes a los constructores Menos y Por. Para el constructor Div:

```
eval (Div a b) = case eval a of
                   Just x → case eval b of
                              Just y → if y == 0 then Nothing
                                       else Just (x `div` y)
                              Nothing → Nothing
                   Nothing → Nothing
```

Ahora el evaluador se puede correr incluso sobre expresiones como la del ejemplo anterior, Div (Val 1) (Val 0), en cuyo caso devuelve Nothing, pero por lo menos su ejecución no resulta interrumpida por un runtime error.

El resultado es satisfactorio, pero esto nos llevó a reescribir todo el programa. Si ahora quisiera agregar nuevas expresiones, como por ejemplo, que incluyeran variables, o la posibilidad de tener efectos laterales como modificación de la memoria, o entrada y salida, o no determinismo, etc. cada una de ellas daría lugar a una modificación de todo o casi todo el código.

Las mónadas vienen en nuestra ayuda. En efecto, si observamos el tipo de eval antes y después del cambio (Integer antes, y Maybe Integer, después) podemos notar que estamos en presencia de mónadas aplicadas a Integer. En el primer caso se trata de la mónada identidad (aplicada a Integer). Y en el segundo se trata de la mónada Maybe (también aplicada a Integer). Dicho sea de paso, la mónada Maybe es simplemente una representación en Haskell de la mónada $T(C) = C + 1$ que vimos la clase pasada.

Ejercicio 332. *Comrobar que el funtor identidad* $I : \mathbf{C} \longrightarrow \mathbf{C}$ *es una mónada.*

Vale la pena mencionar que las demás posibles extensiones mencionadas en el párrafo anterior también dan lugar a mónadas.

¿Y para qué sirve esta observación?

Bueno, si hubiéramos escrito nuestra función eval utilizando únicamente las operaciones de mónadas (el funtor mismo y las transformaciones naturales η y μ) al cambiar una mónada por otra, se podría reutilizar todo, ya que la nueva mónada también tendrá el funtor, η y μ.

Volvamos a nuestro ejemplo. Si originalmente hubiéramos definido la mónada identidad así:

```
type T a = a

t :: (a → b) → T a → T b
```

```
t f = f

η :: a → T a
η a = a

μ :: T (T a) → T a
μ tta = tta
```

y a continuación hubiéramos escrito el evaluador usando solamente la mónada:

```
eval :: Exp → T Integer
eval (Val i) = η i
eval (Mas a b) = μ $ t (λx → μ $ t (λy → η (x+y)) (eval b)) (eval a)
eval (Menos a b) = μ $ t (λx → μ $ t (λy → η (x-y)) (eval b)) (eval a)
eval (Por a b) = μ $ t (λx → μ $ t (λy → η (x*y)) (eval b)) (eval a)
```

al agregar división, el cambio necesario se localizaría en redefinir la mónada

```
type T a = Maybe a

t :: (a → b) → T a → T b
t f = λta → case ta of
                Just a → Just (f a)
                Nothing → Nothing

η :: a → T a
η = Just

μ :: T (T a) → T a
μ tta = case tta of
            Just ta → ta
            Nothing → Nothing
```

agregar la constante

```
div0 :: T a
div0 = Nothing
```

y la ecuación correspondiente a la división

```
eval (Div a b) =
μ $ t (λx → μ $ t (λy → if y == 0 then div0 else η (x `div` y)) (eval b)) (eval a)
```

Este enfoque presenta al menos dos problemas: el cambio en la definición de la mónada también requirió reescribir código y la notación es menos comprensible.

Sobre el primer problema, nos limitamos a mencionar que en la práctica son apenas unas pocas líneas de código que definen los operadores, y muchas líneas de código que los utilizan. Resulta muy conveniente limitar a unas pocas líneas (aquéllas donde se define la mónada) los cambios futuros.

Sobre el segundo problema, es posible utilizar mónadas con una notación más conveniente que la de los operadores t, η y μ. En la práctica, se define el operador $\rightsquigarrow$ (llamado "bind"), que puede definirse con los de la mónada:

```
(⇝) :: T a → (a → T b) → T b
ta ⇝ f = μ $ t f ta
```

Con esta notación, el evaluador resulta

```
eval :: Exp → T Integer
eval (Val i) = η i
eval (Mas a b) = eval a ⇝ (λx → eval b ⇝ (λy → η (x+y)))
eval (Menos a b) = eval a ⇝ (λx → eval b ⇝ (λy → η (x-y)))
eval (Por a b) = eval a ⇝ (λx → eval b ⇝ (λy → η (x*y)))
eval (Div a b) = eval a ⇝ (λx → eval b ⇝ (λy → if y == 0 then div0 else η (x `div` y)))
```

Ejercicio 333. *¿Cómo puede definirse μ a partir de t, η y $\rightsquigarrow$?*

Ejercicio 334. *¿Qué ecuaciones puede derivar de la combinación de t, η y $\rightsquigarrow$ (deducibles de la definición de mónada)?*

El uso de mónadas se ha difundido lo suficiente como para justificar la inclusión, en la sintaxis de Haskell, de notación especial para facilitar su legibilidad:

```
eval :: Exp → T Integer
eval (Val i) = η i
eval (Mas a b) = do
                   x ← eval a
                   y ← eval b
                   η (x+y)
...
eval (Div a b) = do
                   x ← eval a
                   y ← eval b
                   if y == 0 then div0 else η (x `div` y)
```

Ahora podríamos querer agregar variables al lenguaje:

```
data Exp = ...
         | Var String
```

Esto da lugar a una nueva complicación ya que para evaluar variables necesitamos consultar su valor en la memoria, para ello la función eval debería recibir un argumento más. Aparentemente no queda otra que reescribir todo el código de eval ya que ahora tiene un argumento más.

Sin embargo, las mónadas nos permiten evitar esa reescritura. Claro que para ello tenemos que redefinir la mónada

```
type Mem = String → Maybe Integer
type T a = Mem → Maybe a

t :: (a → b) → T a → T b
t f = λta mem → case ta mem of
                  Just a → Just (f a)
                  Nothing → Nothing
```

```
η :: a → T a
η a mem = Just a

μ :: T (T a) → T a
μ tta mem = case tta mem of
                    Just ta → ta mem
                    Nothing → Nothing

div0 :: T a
div0 mem = Nothing

consultar :: String → T Integer
consultar v mem = mem v
```

Todas las líneas del evaluador quedan intactas y se agrega la que corresponde a las variables:

```
eval :: Exp → T Integer
...
eval (Var v) = consultar v
```

Ejercicio 335. *Demostrar que en una categoría con exponenciales, $T(C) = C^M$ es una mónada.*

Ahora podríamos querer agregar la posibilidad de modificar variables:

```
data Exp = ...
         | Asig String Exp
```

Nuevamente redefinimos la mónada

```
type Mem = String → Maybe Integer
type T a = Mem → (Mem, Maybe a)

t :: (a → b) → T a → T b
t f = λta mem → case ta mem of
                    (mem2, Just a) → (mem2, Just (f a))
                    (mem2, Nothing) → (mem2, Nothing)

η :: a → T a
η a mem = (mem, Just a)

μ :: T (T a) → T a
μ tta mem = case tta mem of
                    (mem2, Just ta) → ta mem2
                    (mem2, Nothing) → (mem2,Nothing)

div0 :: T a
div0 mem = (mem,Nothing)
```

```
consultar :: String → T Integer
consultar v mem = (mem, mem v)

modificar :: String → Integer → T Integer
modificar v i mem = (mem2, Just i)
                    where mem2 w = if v == w then Just i else mem w
```

Nuevamente, todas las líneas del evaluador quedan intactas y se agrega

```
eval :: Exp → T Integer
...
eval (Asig v a) = do
                    x ← eval a
                    modificar x
```

Ejercicio 336. *Demostrar que en una categoría con exponenciales, $T(C) = (M \times C)^M$ es una mónada.*

Ejercicio 337. *Agregar al lenguaje expresiones de la forma a□b de no determinismo. El valor de esta expresión es el valor de a o el de b. Se quiere definir eval de modo de que devuelva ahora una lista de todos los valores posibles de una expresión. Modificar la mónada y reescribir las ecuaciones de eval que resulten necesarias (además de escribir la correspondiente al operador □).*

Una posibilidad es definir type T a = Mem → [(Mem,Maybe a)]. Otra posibilidad es considerar solo los valores no erróneos: type T a = Mem → [(Mem,a)].

Ejercicio 338. *En el lenguaje sin □, considerar expresiones que generan output. Se agrega la expresión !a que genera como output el valor de a y devuelve dicho valor.*

Una posibilidad es definir type T a = Mem → (Mem,[Integer],Maybe a).

Ejercicio 339. *Además de output, agregarle input. La expresión ?v, donde v es una variable, lee un entero de entrada, se lo asigna a v en la memoria y devuelve dicho valor como resultado.*

Una posibilidad es definir type T a = Mem → [Integer] → (Mem,[Integer],Maybe a).