Lección 03

Tipos por Sinonimia

Un sinónimo de un tipo introduce un nuevo nombre para un tipo existente, pero no introduce un nuevo tipo. Es posible usar en cualquier momento cualquiera de los nombres en paridad funcional.

```
type Entero = Integer
type DeEnteroEnEntero = Entero -> Entero
type ParFlotantes = (Float, Float)
type Fecha = (Int,Int,Int)
```

Podremos definir instancias y funciones con el nuevo nombre:

```
uno :: Entero
uno = 1

sucesor :: DeEnteroEnEntero
sucesor x = x + 1

parCeros :: ParFlotantes
parCeros = (0.0,0.0)
```

El ejemplo más famoso, consagrado en `Prelude`, es `String` cuya definición es:

```
type String = [Char] está en prelude
```

y permite usar código como:

```
nombre :: String
nombre = ['F','r','a','n','c','i','s','c','o']

juicio :: String
juicio = "Habrá libros más actuales que el de Euclides, pero no
más modernos"
```

Tipos por Enumeración

El nivel de mayor simplicidad en la enumeración de tipos es a través de una enumeración:

```
data DiaSemana = Lunes | Martes | Miercoles | Jueves | Viernes | Sabado | Domingo
```

Con este solo código ninguna instancia del tipo `DiaSemana` sería visible; es necesaria su inclusión en la clase de tipos `Show` y la forma de urgencia es añadiendo `deriving Show`:

```
data DiaSemana = Lunes | Martes | Miercoles | Jueves | Viernes | Sabado | Domingo
                 deriving Show
```

Esto permite ahora un uso como el que sigue:

```
esFinDe :: DiaSemana -> Bool
esFinDe Sabado  = True
esFinDe Domingo = True
esFinDe _       = False
```

La inclusión manual en la clase `Show` podría ser llevada a cabo como sigue. Primero definiríamos el tipo `WeekDay`:

```
data WeekDay = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

y seguidamente sería introducido en la clase `Show` por definición de su "método" `showsPrec` para cada instancia del tipo, al ser éste enumerado:

```
instance Show WeekDay where
    showsPrec _ Mon = showChar 'M'
    showsPrec _ Tue = showChar 'T'
    showsPrec _ Wed = showChar 'W'
    showsPrec _ Thu = showChar 'R'
    showsPrec _ Fri = showChar 'F'
    showsPrec _ Sat = showChar 'S'
    showsPrec _ Sun = showChar 'U'
```

y entonces podríamos operar como sigue:

```
nextDay :: WeekDay -> WeekDay
nextDay Mon = Tue
nextDay Tue = Wed
nextDay Wed = Thu
nextDay Thu = Fri
nextDay Fri = Sat
nextDay Sat = Sun
nextDay Sun = Mon
```

Unión de Tipos

Si deseamos unir dos tipos de datos previamente definidos, la primera idea que viene a la cabeza es:

```
data LetraOEntero = Char | Integer deriving Show
```

y esto es incorrecto, pues cada uno de los casos de una definición de tipo debe comenzar por un constructor de dato (éste puede ser también infijo) y `Char` o `Integer` son constructores de tipo, no de dato.

La forma posible según esto es usar un constructor de tipo específico para cada uno de los tipos a unir. Por ejemplo, si quisiéramos unir en un mismo tipo los caracteres y los enteros, estaríamos obligados a escribir el siguiente código (salvo nombres):

```
data LetraOEntero = Letra Char | Entero Integer deriving Show
```

y entonces sería posible operar así:

```
unValor :: LetraOEntero
unValor = Letra 'x'

otroValor :: LetraOEntero
otroValor = Entero 15

listaMixta :: [LetraOEntero]
listaMixta = [Letra 'a', Entero 12, Entero 10, Letra 'b']
```

y si importásemos la librería `Data.Char` en la cabecera con la orden `import Data.Char` entonces podríamos hacer lo siguiente:

```
incLoE :: LetraOEntero -> LetraOEntero
incLoE (Entero n) = Entero. (+1) $ n
incLoE (Letra c)  = Letra . chr . (+1) . ord $ c
```

Productos

Es posible definir tipos con un único constructor y varias "componentes":

```
data Point = Point Float Float deriving Show
```

El tipo de dato `Point` tiene un constructor con su mismo nombre, lo cual es posible. El constructor `Point` tiene dos campos aceptando `Float` cada uno y que representan las coordenadas del punto.

Los constructores son realmente funciones que aportan un valor del tipo de dato. Los tipos definidos funcionan como cualquier tipo primitivo:

```
ghci> :t Point
Point :: Float -> Float -> Point
```

y sobre los tipos definidos podemos definir otros:

```
data Shape = Circle Point Float |
             Rectangle Point Point
             deriving Show
```

El constructor `Circle` tiene dos campos: uno acepta instancias del tipo `Point`, por el centro, y otro del tipo `Float`, por el radio. El constructor `Rectangle` tiene dos campos aceptando `Point` cada uno de ellos y representando el primero las coordenadas de su esquina superior izquierda y el segundo, las de su esquina inferior derecha.

Los constructores son realmente funciones que aportan un valor del tipo de dato. Funcionan como otro tipo de dato cualquiera:

```
ghci> :t Circle
Circle :: Point -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Point -> Point -> Shape
ghci> Circle (Point 10 20) 5
Circle (Point 10.0 20.0) 5.0
```

```
ghci> Rectangle (Point 50 230) (Point 60 90)
Rectangle (Point 50.0 230.0) (Point 60.0 90.0)
```

y es posible hacer implementaciones sobre él. Por ejemplo:

```
area :: Shape -> Float
area (Circle _  r) = pi * r ^ 2
area (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

lo que ofrece el siguiente diálogo:

```
*ghci> area (Circle (Point 10 20) 10)
314.15927
*ghci> area (Rectangle (Point 0 0) (Point 100 100))
10000.0
```

Registros

Consideremos el siguiente tipo de dato:

```
data Person = Person String String Int Float String String
              deriving (Show)
```

para incluir los datos de un individuo haremos (nombre, apellido, edad, estatura, teléfono, color de ojos:

```
ghci> let guy = Person "Dolores" "Vargas de la Vega" 43 184.2 "526-2928" "Negro"
ghci> guy
Person "Dolores" "Vargas de la Vega" 43 184.2 "526-2928" "Negro"
```

Podríamos escribir algunos métodos para el tipo de dato:

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

age :: Person -> Int
age (Person _ _ age _ _ _) = age

height :: Person -> Float
height (Person _ _ _ height _ _) = height

phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number

eColor :: Person -> String
eColor (Person _ _ _ _ _ eColor) = eColor
```

y esto ofrece el siguiente diálogo:

```
ghci> firstName guy
"Dolores"
ghci> height guy
184.2
ghci> eColor guy
"Negro"
```

Pero la misma funcionalidad puede ser obtenida de otra forma:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , eColor :: String
                     } deriving (Show)
```

y esto provee el siguiente diálogo:

```
ghci> :t eColor
flavor :: Person -> String
ghci> :t firstName
firstName :: Person -> String
```

Entre ambas opciones hay una diferencia en lo relacionado con la presentación de los objetos. En cuanto a la primera opción, el diálogo sería éste:

```
*ghci> guy
Person "Dolores" "Vargas de la Vega" 43 184.2 "526-2928" "Negro"
```

pero en cuanto a la segunda opción, sería éste:

```
*ghci> guy
Person {firstName = "Dolores", lastName = "Vargas de la Vega", age
= 43, height = 184.2, phoneNumber = "526-2928", eColor = "Negro"}
```

Incluso podríamos hacer algo como esto:

```
guy1 :: Person
guy1 = Person { eColor = "Verde"
              , lastName = "Velez Huertas"
              , age = 28
              , height = 190.5
              , phoneNumber = "333-4444"
              , firstName = "Macarena"
              }
```

sin que el desorden altere la creación de la instancia solicitada:

```
ghci> guy1
Person {firstName = "Macarena", lastName = "Velez Huertas", age = 28, height = 190.5,
phoneNumber = "333-4444", eColor = "Verde"}
```

```
ghci> firstName (guy1)
"Macarena"
```

Parámetros de Tipo

Un constructor de tipos puede tomar tipos como parámetro y producir nuevos tipos. El ejemplo arquetípico es el siguiente (incluido en `Prelude`):

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

aquí la `a` es un parámetro de tipo. Dependiendo de lo que queramos que este tipo contenga cuando un valor no es Nothing, este tipo puede acabar produciendo tipos como `Maybe Int`, `Maybe Char`, `Maybe String`, etc. Ningún valor puede tener un tipo que sea simplemente `Maybe`, ya que eso no es un tipo por si mismo, es un constructor de tipos. Para que sea un tipo real que algún valor pueda tener, tiene que tener todos los parámetros de tipo definidos.

Si pasamos `Char` como parámetro de tipo a `Maybe`, obtendremos el tipo `Maybe Char`. Por ejemplo, el valor `Just 'a'` tiene el tipo `Maybe Char`. La situación es similar a los casos: `[Int]`, `[Char]`, `[[String]]`, etc. pero no puede haber un valor cuyo tipo sea simplemente `[]`.

Tipo de Datos Recursivos

Hasta ahora hemos definido tipos de datos con un número finito de "tipos" de instancia; no obstante esto no tiene por qué ser necesariamente así. Veamos el siguiente ejemplo:

```
data Nat = Cero | Suc Nat deriving Show
```

algunas instancias suyas son:

```
uno :: Nat
uno = Suc Cero

dos :: Nat
dos = Suc (Suc Cero)    -- podríamos haber escrito Suc uno

indefinidoN :: Nat
indefinidoN = undefined
```

donde podemos notar el uso de `undefined` que en `Prelude` está definido como:

```
undefined :: a
undefinded
 | False = undefined
```

y pertenece a cualquier tipo. Algunas funciones para el tipo son:

```
esCero :: Nat -> Bool
esCero Cero = True
esCero _    = False
```

```
infinitoN :: Nat
infinitoN = Suc infinitoN

esPar :: Nat -> Bool
esPar Cero    = True
esPar (Suc x) = not (esPar x)

infixl 6 <+>
(<+>)         :: Nat -> Nat -> Nat
m <+> Cero    = m
m <+> (Suc n) = Suc (m <+> n)

-- Un código alternativo sería

infixl 6 <+>
(<+>)         :: Nat -> Nat -> Nat
Cero <+> n    = n
(Suc m) <+> n = Suc (m <+> n)

construirN :: Int -> Nat
construirN 0 = Cero
construirN n = Suc (construirN (n-1))

desmontarN :: Nat -> Int
desmontarN Cero    = 0
desmontarN (Suc x) = 1 + desmontarN x

infixl 7 <**> -- <*> está reservado en el núcleo
(<**>)         :: Nat -> Nat -> Nat
m <**> Cero    = Cero
m <**> (Suc n) = m <**> n <+> m

infixr 8  <^>
(<^>)          :: Nat -> Nat -> Nat
m <^> Cero     = uno
m <^> (Suc n)  = m <*> m <^> n
```

Definiciones con `newtype`

En Haskell la declaración `newtype` crean un nuevo tipo isomórfico (con los mismos valores) a uno existente, pero con una identidad propia par el sistema de tipos. Por ejemplo, los números naturales pueden ser representados usando el tipo `Integer` mediante lo siguiente:

```
newtype Natural = ANatural Integer deriving Show
```

Para convertir una instancia de `Integer` en `Natural` podemos usar una función de conversión:

```
toNatural :: Integer -> Natural
toNatural m
```

```
 | m < 0     = error "No es posible usar enteros negativos"
 | otherwise = ANatural m
```

y para pasar de `Natural` a `Integer` esta otra:

```
fromNatural :: Natural -> Integer
fromNatural (ANatural n) = n
```

Pero algo más eficaz podría ser lo siguiente despues de importar el módulo Data.Maybe:

```
newtype Natural = ANatural Integer deriving Show

toNatural :: Maybe Integer -> Maybe Natural
toNatural m
 | isNothing m || isJust m  && fromJust m < 0 = Nothing
 | otherwise = Just (ANatural (fromJust m))

fromNatural :: Maybe Natural -> Maybe Integer
fromNatural m  = case m of
Nothing -> Nothing
Just (ANatural n) -> Just n
```

Cómo Exportar

Si quisiéramos exportar las funciones y tipos creados (ver ejercicio complementario) en un módulo que contuviese lo tratado para `Shape`, podríamos empezar con esto:

```
module Shapes
( Point(..)
, Shape(..)
, area
, nudge
, baseCircle
, baseRect
) where
```

Haciendo `Shape (..)` estamos exportando todos los constructores de datos de `Shape`, lo que significa que cualquiera que importe nuestro módulo puede crear figuras usando los constructores `Circle` y `Rectangle`. Sería lo mismo que escribir `Shape (Rectangle, Circle)`.

También podríamos optar por no exportar ningún constructor de datos para `Shape` simplemente escribiendo `Shape` en dicha sentencia. De esta forma, quien importe nuestro módulo solo podrá crear figuras utilizando las funciones auxiliares `baseCircle` y `baseRect`. `Data.Map` utiliza este método. No puedes crear un diccionario utilizando `Map.Map [(1,2),(3,4)]` ya que no se exporta el constructor de datos. Sin embargo, podemos crear un diccionario utilizando funciones auxiliares como `Map.fromList`. Recuerda, los constructores de datos son simples funciones que toman los campos del tipo como parámetros y devuelven un valor de un cierto tipo (como `Shape`) como resultado. Así que cuando elegimos no exportarlos, estamos previniendo que la gente que importa nuestro módulo pueda utilizar esas funciones, pero si alguna otra función devuelve el tipo que estamos exportando, las podemos utilizar para crear nuestros propios valores de ese tipo.

No exportar los constructores de datos de un tipo de dato lo hace más abstracto en el sentido de que oculta su implementación. Sin embargo, los usuarios del módulo no podrán usar el ajuste de patrones sobre ese tipo.

Ejercicios

1. Defina una función que calcule el día de semana, según el calendario gregoriano, de una fecha. Sugerimos usar la fórmula conocida como "Congruencia de Zeller".

2. Defina una función que calcule el día de semana, según el calendario juliano, de una fecha. Sugerimos usar la fórmula conocida como "Congruencia de Zeller".

3. Defina un tipo de dato que una grados centígrados y grados Fahrenheit y escriba funciones que traduzcan de una de las escalas a la otra. Por ejemplo, darían el siguiente diálogo:

   ```
   *EjerciciosPrac03> fToC (Fahrenheit 68)
   Centigrados 20.0
   *EjerciciosPrac03> cToF (Centigrados 20)
   Fahrenheit 68.0
   *EjerciciosPrac03> fToC (Fahrenheit (-40))
   Centigrados (-40.0)
   *EjerciciosPrac03> cToF (Centigrados (-40))
   Fahrenheit (-40.0)
   *EjerciciosPrac03>
   ```

4. Sobre la base del tipo de dato `Shape` de la explicación defina una función `baseCircle` de un argumento `Float` que construya el círculo base de radio `r` (un círculo de centro el punto `(0,0)` y de radio `r`). Haga lo mismo para el `Rectangle` definiendo una función `baseRect` de dos argumentos, cada uno de tipo `Float`, que construya el rectángulo que tenga uno de sus dos vértices definitorios apoyados en `(0,0)` y el otro libre. Defina una función `nudge` que transforme la figura base, círculo o rectángulo, en otra dada por los argumentos y que sea de la misma forma.

5. Defina el tipo de dato `Triangulo` e implemente una función `area` que sea capaz de calcular su área. (Sug.: use al efecto la *Fórmula de Herón*.)

6. Diseñe un tipo de dato escrito en modo registro "funcional" que describa los coches usados de un negocio de venta de los mismos. Seguidamente escriba una función que recoja en una frase explicativa los detalles que contiene cualquier instancia concreta del tipo de dato.

7. Defina el tipo de dato `Vector a` y escriba funciones que codifiquen el producto vectorial y el escalar.

8. Defina el tipo de dato `Complex` y defina una función que extraiga el módulo y otros para codificar: la suma, la resta, la multiplicación y la división.

9. Defina las funciones `divNat` y `modNat` que calculen respectivamente el conciente entero y el resto de dividir dos números naturales.