PROGRAMANDO EN HASKELL

Capítulo 8 – Declaración de Tipos y Clases

Declaraciones de Tipo

En Haskell, se puede definir un nuevo nombre para un tipo ya existente usando una declaración de tipo.

```
type String = [Char]
```

String es un sinónimo para el tipo [Char].

Las declaraciones de tipo se utilizan para aumentar la legibilidad de otros tipos. Por ejemplo, dado

```
type Pos = (Int,Int)
```

podemos definir:

```
origin :: Pos
origin = (0,0)

left :: Pos → Pos
left (x,y) = (x-1,y)
```

Al igual que las definiciones de funciones, las declaraciones de tipo también pueden tener parámetros. Por ejemplo, dado

```
type Pair a = (a,a)
```

podemos definir:

```
mult :: Pair Int → Int
mult (m,n) = m*n

copy :: a → Pair a
copy x = (x,x)
```

Las declaraciones de tipo pueden anidarse:

```
type Pos = (Int,Int)

type Trans = Pos → Pos
```

Sin embargo, no pueden ser recursivas:

```
type Tree = (Int,[Tree])
```

Declaraciones de Datos

Un tipo completamente nuevo puede definirse al especificar sus valores usando una declaración de datos.

```
data Bool = False | True
```

Bool es un tipo nuevo, con dos nuevos valores False y True.

Nota:

- Los dos valores False y True se llaman constructores del tipo Bool.

- Los nombres de tipo y constructor siempre deben comenzar con una letra mayúscula.

- Las declaraciones de datos son similares a las gramáticas libres de contexto. Las primeras especifica los valores de un tipo, y las segundas las sentencias de un lenguaje.

Los valores para los tipos nuevos se utilizan de la misma forma como los tipos que ya trae Haskell. Por ejemplo, dado

```
data Answer = Yes | No | Unknown
```

podemos definir:

```
answers :: [Answer]
answers = [Yes,No,Unknown]

flip :: Answer → Answer
flip Yes     = No
flip No      = Yes
flip Unknown = Unknown
```

Los constructores en la declaración de un dato también pueden tener parámetros. Por ejemplo, dado

```
data Shape = Circle Float
           | Rect Float Float
```

podemos definir:

```
square :: Float → Shape
square n = Rect n n

area :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Nota:

- Shape tiene valores de la forma Circle r donde r es un Float, y Rect x y donde ‘x’ y ‘y’ son del tipo Float.

- Circle y Rect pueden ser vistos como funciones que construyen valores de tipo Shape:

```
Circle :: Float → Shape

Rect :: Float → Float → Shape
```

Sin sorpresa alguna, las declaraciones de datos en sí mismas también pueden tener parámetros. Por ejemplo, dado

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

podemos definir:

```
safediv :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] → Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

Tipos Recursivos

En Haskell, los tipos nuevos pueden declararse en términos de sí mismos. Esto es, los tipos pueden ser recursivos.

```
data Nat = Zero | Succ Nat
```

Nat es un tipo nuevo, con constructores Zero :: Nat y Succ :: Nat $\rightarrow$ Nat.

Nota:

- Un valor de tipo Nat o es Zero, o es de la forma Succ n donde n :: Nat. Esto es, Nat contiene la siguiente secuencia infinita de valores:

```
Zero

Succ Zero

Succ (Succ Zero)

⋮
```

- Podemos pensar sobre los valores de tipo Nat como números naturales, donde Zero representa 0, y Succ representa la función de sucesión 1+.

- Por ejemplo, el valor

```
Succ (Succ (Succ Zero))
```

representa el número natural

```
1 + (1 + (1 + 0))  =  3
```

Usando recursión, es fácil definir funciones que realizan conversiones entre valores de tipo Nat e Int:

```
nat2int :: Nat → Int
nat2int Zero     = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int → Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

Dos números naturales pueden ser sumados al convertirlos a enteros, sumarlos, y después convertirlos nuevamente al tipo Nat:

```
add :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

Sin embargo, usando recursión la función add puede definirse sin la necesidad de realizar conversiones:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

Por ejemplo:

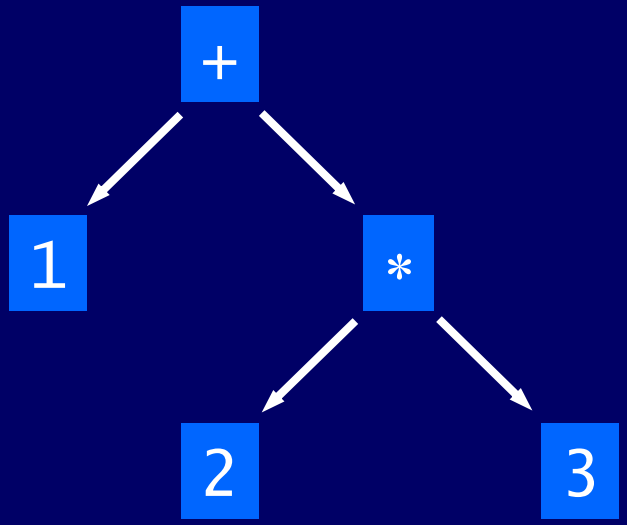
```
  add (Succ (Succ Zero)) (Succ Zero)
=
  Succ (add (Succ Zero) (Succ Zero))
=
  Succ (Succ (add Zero (Succ Zero))
=
  Succ (Succ (Succ Zero))
```

Nota:

- La definición recursiva para sumar se corresponde con las leyes 0+n = n y (1+m)+n = 1+(m+n).

Expresiones Aritméticas

Considera una forma simple de expresiones a base de enteros, utilizando suma y multiplicación.



Con recursión, un tipo adecuado para representar tales expresiones puede declararse como:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

Por ejemplo, la expresión de la diapositiva anterior se representaría así:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Con recursión, ahora es fácil definir funciones que procesan expresiones. Por ejemplo:

```
size :: Expr → Int
size (Val n)   = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval :: Expr → Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Nota:

- Los tres constructores tienen tipo:

```
Val :: Int → Expr
Add :: Expr → Expr → Expr
Mul :: Expr → Expr → Expr
```

- Muchas funciones sobre expresiones pueden definirse reemplazando los constructores por otras funciones, utilizando una función fold adecuada. Por ejemplo:

```
eval = folde id (+) (*)
```

Ejercicios

(1) Utilizando recursión y la función add, define una función que multiplica dos números naturales.

(2) Define una función folde para expresiones y proporciona algunos ejemplos de uso.

(3) Define un tipo Tree a de árboles binarios construido de valores Leaf de un tipo a utilizando un constructor Node que toma dos árboles binarios como parámetros.

Fuente.

Profesor Graham Hutton.
http://www.cs.nott.ac.uk/~pszgmh/

Traducción al español.

Caleb Josue Ruiz Torres.
https://www.calebjosue.com