Programación funcional con Haskell
Tipos

Juan Manuel Rabasedas

Declaración de Tipos

- En Haskell podemos definir un nuevo nombre para un tipo existente usando una declaración **type**.

  type String = [Char]

  String es un sinónimo del tipo [Char].

- Los sinónimos de tipo hace que ciertas declaraciones de tipos sean más fáciles de leer.

  type Pos = (Int, Int)

  $$origin :: \text{Pos}$$
  $$origin = (0, 0)$$
  $$left :: \text{Pos} \rightarrow \text{Pos}$$
  $$left\ (x, y) = (x - 1, y)$$

Declaración de Tipos

- Como en las funciones, la declaración de tipos puede tener parámetros:

$$\textbf{type}\ \text{Par}\ a = (a, a)$$
$$mult :: \text{Par Int} \to \text{Int}$$
$$mult\ (x, y) = x * y$$
$$copy\ \ :: a \to \text{Par}\ a$$
$$copy\ x = (x, x)$$

- Las declaraciones de tipo pueden anidarse

  type Pos = (Int, Int)

  type Trans = Pos → Pos ✓

  pero no pueden ser recursivos

  type Tree = (Int, [Tree]) ×

Declaraciones **data**

- Un nuevo tipo completo puede ser definido dando sus valores con la declaración **data**.

  data Bool = False | True

  declara un nuevo tipo Bool con dos nuevos valores False y True.
- True y False son los llamados constructores del tipo Bool
- Los nombres de los constructores deben empezar con mayúsculas.
- Los constructores definen los distintos elementos del tipo.

Declaraciones **data**

- Los valores de un nuevo tipo se usan igual que los predefinidos

$$\textbf{data}\ \text{Answer} = \text{Yes} \mid \text{No} \mid \text{Unknown}$$

$$\text{Answers} :: [\text{Answer}]$$
$$\text{Answers} = [\text{Yes}, \text{No}, \text{Unknown}]$$

$$\mathit{flip} :: \text{Answer} \to \text{Answer}$$
$$\mathit{flip}\ \text{Yes} = \text{No}$$
$$\mathit{flip}\ \text{No} = \text{Yes}$$
$$\mathit{flip}\ \text{Unknown} = \text{Unknown}$$

Declaraciones **data**

- Los constructores en las declaraciones **data** pueden tener parámetros

$$\textbf{data}\ \text{Shape} = \text{Circle Float} \mid \text{Rect Float Float}$$

$$\begin{aligned} square\ n &= \text{Rect}\ n\ n \\ area\ (\text{Circle}\ r) &= \pi * r \uparrow 2 \\ area\ (\text{Rect}\ x\ y) &= x * y \end{aligned}$$

- Los constructores pueden ser vistos como funciones

$$\begin{aligned} &\text{Circle} :: \text{Float} \to \text{Shape} \\ &\text{Rect} :: \text{Float} \to \text{Float} \to \text{Shape} \end{aligned}$$

Declaraciones **data**

No es sorpresa que las declaraciones **data** pueden también tener parámetros de tipos.

data Maybe a = Nothing | Just a

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

$safehead$ :: $[a]$ → Maybe a
$safehead\ [\,]$ = Nothing
$safehead\ xs$ = Just $(head\ xs)$

Maybe *es un* *constructor de tipos* *ya que dado un tipo* a*, construye el tipo* Maybe a.

Declaraciones **data**

- Las declaraciones **data** pueden ser recursivas:

$$\begin{aligned}
&\textbf{data}\ \text{Nat} = \text{Zero} \mid \text{Succ Nat} \\
&add\ n\ \text{Zero} &&= n \\
&add\ n\ (\text{Succ}\ m) &&= \text{Succ}\ (add\ m\ n)
\end{aligned}$$

- Nat es un nuevo tipo con constructores Zero :: Nat y Succ :: Nat $\rightarrow$ Nat
- Un valor de tipo Nat puede ser Zero o de la forma Succ n donde n :: Nat
  Zero, Succ Zero, Suc (Succ Zero), Suc (Suc (Succ Zero)), ...
- Podemos pensar a los valores del tipo Nat como números naturales donde:
  - Zero representa a 0 y Succ representa la función sucesor $(1+)$
  - Suc (Suc (Succ Zero)) $= (1 + 1 + 1 + 0) = 3$

Declaraciones **data**

Usando recursión es fácil definir funciones que conviertan los valores entre los tipos Nat e Int

$$
\begin{aligned}
&nat2int :: \mathsf{Nat} \to \mathsf{Int} \\
&nat2int\ \mathsf{Zero} \quad\ = 0 \\
&nat2int\ (\mathsf{Succ}\ n) = 1 + nat2int\ n
\end{aligned}
$$

$$
\begin{aligned}
&int2nat :: \mathsf{Int} \to \mathsf{Nat} \\
&int2nat\ 0 = \mathsf{Zero} \\
&int2nat\ n = \mathsf{Succ}\ (int2nat\ (n-1))
\end{aligned}
$$

Declaraciones **data**

- Dos naturales pueden ser sumados convirtiéndolos en enteros, sumados, y convertidos nuevamente a naturales

$$add :: \text{Nat} \to \text{Nat} \to \text{Nat}$$
$$add\ m\ n = int2nat\ (nat2int\ m + nat2int\ n)$$

- Sin embargo usando recursión la misma función se puede definir sin la necesidad de conversión:

$$add\ \text{Zero}\ n = n$$
$$add\ (\text{Succ}\ m)\ n = \text{Succ}\ (add\ m\ n)$$

- Ejercicio: definir la multiplicación para Nat
- Ejercicio: definir la exponenciación para Nat.

Declaraciones **data**

- Las declaraciones **data** pueden ser recursivas y con parámetros
  data List a = Nill | Cons a (List a)
- Usando recursión es facil definir funciones que conviertan los valosres entre los tipos List a y $[a]$

$$
\begin{aligned}
&to :: \text{List } a \to [a] \\
&to \text{ Nil} = [\,] \\
&to \ (\text{Cons } x \ xs) = x : (to \ xs) \\
&from :: [a] \,!\, \text{List } a \\
&from \ [\,] = \text{Nil} \\
&from \ (x : xs) = \text{Cons } x \ (from \ xs)
\end{aligned}
$$

Sintaxis de registro (Record)

Podemos crear un tipo que describa a una persona:

- Con los campos: nombre, apellidos, edad, altura, número de teléfono y el sabor de su helado favorito.

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

- creemos una persona (una instancia):

```
>let guy =
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
>guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

Sintaxis de registro (Record)

Para obtener la información guardada en un tipo Person debemos definir funciones para cada campo:

$$
\begin{aligned}
&\mathit{firstName} && (\text{Person } \mathit{firstname} \ \_ \ \_ \ \_ \ \_ \ \_) && = \mathit{firstname} \\
&\mathit{lastName} && (\text{Person } \_ \ \mathit{lastname} \ \_ \ \_ \ \_ \ \_) && = \mathit{lastname} \\
&\mathit{age} && (\text{Person } \_ \ \_ \ \mathit{age} \ \_ \ \_ \ \_) && = \mathit{age} \\
&\mathit{height} && (\text{Person } \_ \ \_ \ \_ \ \mathit{height} \ \_ \ \_) && = \mathit{height} \\
&\mathit{phoneNumber} && (\text{Person } \_ \ \_ \ \_ \ \_ \ \mathit{number} \ \_) && = \mathit{number} \\
&\mathit{flavor} && (\text{Person } \_ \ \_ \ \_ \ \_ \ \_ \ \mathit{flavor}) && = \mathit{flavor}
\end{aligned}
$$

Debe de haber un método mejor.

Sintaxis de registro (Record)

Podemos declarar el mismo tipo de la siguiente manera:

$$
\begin{aligned}
\textbf{data}\ \text{Person} = \text{Person}\ \{\ & \mathit{firstName} :: \text{String} \\
,\ & \mathit{lastName} :: \text{String} \\
,\ & \mathit{age} :: \text{Int} \\
,\ & \mathit{height} :: \text{Float} \\
,\ & \mathit{phoneNumber} :: \text{String} \\
,\ & \mathit{flavor} :: \text{String} \\
\}\ & \textbf{deriving}\ (\text{Show})
\end{aligned}
$$

Se crean automáticamente las funciones para obtener los campos: $\mathit{firstName}$, $\mathit{lastName}$, age, height, $\mathit{phoneNumber}$ y flavor.

Sintaxis de registro (Record)

Ahora podemos definir:

$$
\begin{aligned}
\text{Person } \{ & \mathit{lastName} && = \texttt{"Finklestein"} \\
, & \mathit{firstName} && = \texttt{"Buddy"} \\
, & \mathit{heigt} && = 184.2 \\
, & \mathit{age} && = 43 \\
, & \mathit{flavor} && = \texttt{"Chocolate"} \\
, & \mathit{phoneNumbre} && = \texttt{"526-2928"} \\
\} &
\end{aligned}
$$

Notar que no es necesario respetar el orden en que fueron declarado los campos.

Expresiones **case**

- Además de pattern matching en el lado izq. de una definición, podemos usar una expresión **case**

$$
\begin{aligned}
&esCero :: \text{Nat} \to \text{Bool} \\
&esCero\ n = \textbf{case}\ n\ \textbf{of} \\
&\qquad\qquad\qquad \text{Zero} \to \text{True} \\
&\qquad\qquad\qquad \_\quad\ \to \text{False}
\end{aligned}
$$

- Los patrones de los diferentes casos son intentados en orden
- Se usa la indentación para marcar un bloque de casos

Referencias

- Programming in Haskell. Graham Hutton, CUP 2007.
- Introducción a la Programación Funcional con Haskell. Richard Bird, Prentice Hall 1997.
- ¡Aprende Haskell por el bien de todos! - Capitulo 8
  Creando nuestros propios tipos y clases de tipos