Tipos en Haskell

Mauro Jaskelioff

01/04/2016

Tipos en Haskell

- Haskell posee un sistema de tipos expresivo.
- Haskell tiene tipado estático.
  - El tipado estático significa que los tipos son chequeados en tiempo de compilación.
- Haskell tiene inferencia de tipos
  - Sin embargo, dar el tipo de las funciones es ventajoso.
- Veremos varias formas de definir tipos de datos en Haskell
  - Sinónimos de tipos
  - Tipos Algebraicos
  - Constructores de tipo

Sinónimos de tipos

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

  $$\textbf{type}\ \mathit{String} = [\mathit{Char}]$$

  String es un sinónimo del tipo $[\mathit{Char}]$.

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

  $$\begin{aligned}
  &\textbf{type}\ \mathit{Pos} = (\mathit{Int}, \mathit{Int})\\
  &\mathit{origen} \quad :: \mathit{Pos}\\
  &\mathit{origen} \quad = (0, 0)\\
  &\mathit{izq} \quad :: \mathit{Pos} \to \mathit{Pos}\\
  &\mathit{izq}\ (x, y) = (x - 1, y)
  \end{aligned}$$

Sinónimos de Tipos

- Los sinónimos de tipo pueden tener parámetros

$$\textbf{type } Par\ a = (a, a)$$
$$copiar \quad :: a \to Par\ a$$
$$copiar\ x = (x, x)$$

- Los sinónimos de tipo pueden anidarse

$$\textbf{type } Punto = (Int, Int)$$
$$\textbf{type } Trans\ = Punto \to Punto$$

pero no pueden ser recursivos

$$\textbf{type } Tree = (Int, [\,Tree\,])$$

Declaraciones **data**

- los **data** declaran un nuevo tipo cuyos valores se especifican en la declaración.

  $$\textbf{data}\ \mathit{Bool} = \mathit{False} \mid \mathit{True}$$

  declara un nuevo tipo *Bool* con dos nuevos valores *False* y *True*.
- *True* y *False* son los *constructores* del tipo *Bool*
- Los nombres de los constructores deben empezar con mayúsculas.
- Dos constructores diferentes siempre construyen diferentes valores del tipo.

Usando **data**

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

$$
\begin{aligned}
&\textbf{data}\ \mathit{Respuesta} = \mathit{Si} \mid \mathit{No} \mid \mathit{Desconocida}\\
&\mathit{respuestas} :: [\mathit{Respuesta}]\\
&\mathit{respuestas} = [\mathit{Si}, \mathit{No}, \mathit{Desconocida}]\\
&\mathit{invertir} :: \mathit{Respuesta} \to \mathit{Respuesta}\\
&\mathit{invertir}\ \mathit{Si} = \mathit{No}\\
&\mathit{invertir}\ \mathit{No} = \mathit{Si}\\
&\mathit{invertir}\ \mathit{Desconocida} = \mathit{Desconocida}
\end{aligned}
$$

Usando **data**

- Ejemplo en que los constructores tienen parámetros

$$
\begin{aligned}
&\textbf{data}\ \mathit{Shape} = \mathit{Circle}\ \mathit{Float} \mid \mathit{Rect}\ \mathit{Float}\ \mathit{Float}\\
&\mathit{square} \quad :: \mathit{Float} \to \mathit{Shape}\\
&\mathit{square}\ n = \mathit{Rect}\ n\ n\\
&\mathit{area} \qquad\qquad :: \mathit{Shape} \to \mathit{Float}\\
&\mathit{area}\ (\mathit{Circle}\ r) \ = \pi * r \uparrow 2\\
&\mathit{area}\ (\mathit{Rect}\ x\ y) = x * y
\end{aligned}
$$

- Los constructores son funciones

$$
\begin{aligned}
&> :t\ \mathit{Circle}\\
&\mathit{Circle} :: \mathit{Float} \to \mathit{Shape}\\
&> :t\ \mathit{Rect}\\
&\mathit{Rect} :: \mathit{Float} \to \mathit{Float} \to \mathit{Shape}
\end{aligned}
$$

¿Qué complejidad tienen estas funciones?

Sintaxis para Records

- Definimos un tipo de datos para guardar datos de alumnos:

  data *Alumno* = *A String String Int String* **deriving** *Show*

- Definimos un alumno

  juan = *A* `"Juan"` `"Perez"` 21 `"jperez999999@gmail.com"`

- Para acceder a los datos es útil definir funciones:

  nombre :: *Alumno* → *String*
  nombre (*A n* _ _ _) = *n*

  apellido :: *Alumno* → *String*
  apellido (*A* _ *a* _ _) = *a*

  edad :: *Alumno* → *Int*
  edad (*A* _ _ *e* _) = *e*

  email :: *Alumno* → *String*
  email (*A* _ _ _ *m*) = *m*

Sintaxis para Records (cont.)

- Haskell provee sintaxis para aliviarnos el trabajo:

$$\begin{aligned}\textbf{data}\ \mathit{Alumno} = A\ \{&\mathit{nombre} :: \mathit{String}\\ ,&\mathit{apellido} :: \mathit{String}\\ ,&\mathit{edad} :: \mathit{Int}\\ ,&\mathit{email} :: \mathit{String}\\ \}\ &\textbf{deriving}\ \mathit{Show}\end{aligned}$$

- No tenemos que definir las proyecciones por separado.
- Cambia la instancia de *Show*.
- No tenemos que acordarnos el orden de los campos:

$$\begin{aligned}\mathit{juan} = A\ \{&\mathit{apellido} = \texttt{"Perez"}\\ ,&\mathit{nombre} = \texttt{"Juan"}\\ ,&\mathit{email} = \texttt{"jperez999999@gmail.com"}\\ ,&\mathit{edad} = 21\}\end{aligned}$$

Constructores de Tipos

- Las declaraciones **data** pueden tener parámetros de tipos.

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

$$\mathit{safehead} :: [a] \to \mathit{Maybe}\ a$$
$$\mathit{safehead}\ [\,] = \mathit{Nothing}$$
$$\mathit{safehead}\ \mathit{xs} = \mathit{Just}\ (\mathit{head}\ \mathit{xs})$$

- *Maybe* es un constructor de tipos ya que dado un tipo a, construye el tipo *Maybe* a.
- *Maybe* tiene *kind* $* \to *$. En GHCi:

$$> :\mathit{kind}\ \mathit{Maybe}$$
$$\mathit{Maybe} :: * \to *$$

El constructor de Tipos *Either*

data *Either a b* = *Left a* | *Right b*

- Describe un tipo que puede tener elementos de dos tipos.
- En sus elementos está claro de qué tipo es el elemento almacenado.
- Ejercicio: Dar 4 elementos diferentes de tipo *Either Bool Bool*.
- En una interpretación naif de tipos como conjuntos, el tipo *Either* corresponde a la *unión disjunta*.
- Ejercicio: ¿Qué *kind* tiene *Either*?

Tipos Recursivos

- Las declaraciones **data** pueden ser recursivos

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

- Ejercicio: definir la multiplicación para *Nat*

Tipos Recursivos con Parámetros

- Por supuesto, los tipos recursivos pueden tener parámetros.

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

- El tipo *List* es *isomorfo* a las listas predefinidas

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

Expresiones **case**

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

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

- 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.