Tipos en Haskell

Mauro Jaskelioff

11/04/2022

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.

  $$\begin{array}{ll} \multicolumn{2}{l}{\textbf{type}\ \mathit{Pos} = (\mathit{Int}, \mathit{Int})} \\ \mathit{origen} & :: \mathit{Pos} \\ \mathit{origen} & = (0,0) \\ \mathit{izq} & :: \mathit{Pos} \to \mathit{Pos} \\ \mathit{izq}\ (x,y) & = (x-1, y) \end{array}$$

Sinónimos de Tipos

- Los sinónimos de tipo pueden tener parámetros

$$\textbf{type } \mathit{Par}\ a = (a, a)$$
$$\mathit{copiar} :: a \rightarrow \mathit{Par}\ a$$
$$\mathit{copiar}\ x = (x, x)$$

- Los sinónimos de tipo pueden anidarse

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

pero no pueden ser recursivos

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

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

$$\textbf{data}\ \mathit{Respuesta} = \mathit{Si} \mid \mathit{No} \mid \mathit{Desconocida}$$

$$\begin{aligned}&\mathit{respuestas} :: [\mathit{Respuesta}]\\&\mathit{respuestas} = [\mathit{Si}, \mathit{No}, \mathit{Desconocida}]\end{aligned}$$

$$\begin{aligned}&\mathit{invertir} && :: \mathit{Respuesta} \to \mathit{Respuesta}\\&\mathit{invertir}\ \mathit{Si} && = \mathit{No}\\&\mathit{invertir}\ \mathit{No} && = \mathit{Si}\\&\mathit{invertir}\ \mathit{Desconocida} && = \mathit{Desconocida}\end{aligned}$$

Usando **data**

- Ejemplo en que los constructores tienen parámetros

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

$$\begin{aligned} &\mathit{square} \quad :: \mathit{Float} \to \mathit{Shape} \\ &\mathit{square}\ n = \mathit{Rect}\ n\ n \end{aligned}$$

$$\begin{aligned} &\mathit{area} && :: \mathit{Shape} \to \mathit{Float} \\ &\mathit{area}\ (\mathit{Circle}\ r) && = \pi * r \uparrow 2 \\ &\mathit{area}\ (\mathit{Rect}\ x\ y) && = x * y \end{aligned}$$

- 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:

$$\textbf{data}\ \mathit{Alumno} = \mathit{A}\ \mathit{String}\ \mathit{String}\ \mathit{Int}\ \mathit{String}\ \textbf{deriving}\ \mathit{Show}$$

- Definimos un alumno

$$\mathit{juan} = \mathit{A}\ \texttt{"Juan"}\ \texttt{"Perez"}\ 21\ \texttt{"jperez999999@gmail.com"}$$

- Para acceder a los datos es útil definir funciones:

$$\begin{aligned}
&\mathit{nombre} :: \mathit{Alumno} \to \mathit{String}\\
&\mathit{nombre}\ (\mathit{A}\ n\ _\ _\ _) = n\\
&\mathit{apellido} :: \mathit{Alumno} \to \mathit{String}\\
&\mathit{apellido}\ (\mathit{A}\ _\ a\ _\ _) = a\\
&\mathit{edad} :: \mathit{Alumno} \to \mathit{Int}\\
&\mathit{edad}\ (\mathit{A}\ _\ _\ e\ _) = e\\
&\mathit{email} :: \mathit{Alumno} \to \mathit{String}\\
&\mathit{email}\ (\mathit{A}\ _\ _\ _\ m) = m
\end{aligned}$$

Sintaxis para Records (cont.)

- Haskell provee sintaxis para aliviarnos el trabajo:

$$\begin{array}{l}\textbf{data}\ \mathit{Alumno} = A\ \{\ \mathit{nombre} :: \mathit{String} \\ \qquad\qquad\qquad\quad ,\mathit{apellido} :: \mathit{String} \\ \qquad\qquad\qquad\quad ,\mathit{edad} :: \mathit{Int} \\ \qquad\qquad\qquad\quad ,\mathit{email} :: \mathit{String} \\ \qquad\qquad\qquad\quad \}\ \textbf{deriving}\ \mathit{Show}\end{array}$$

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

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

Constructores de Tipos

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

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

- *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 tipo *Maybe*

$$\mathbf{data}\ Maybe\ a = Nothing \mid Just\ a$$

- *Maybe* suele usarse para señalar una condición de error, o hacer total una función parcial.

$$\begin{aligned} & safehead && :: [a] \to Maybe\ a \\ & safehead\ [\,] && = Nothing \\ & safehead\ xs && = Just\ (head\ xs) \end{aligned}$$

- Si tenemos una lista de pares clave/valor

$$\begin{aligned} & lookup :: Eq\ c \Rightarrow c \to [(c, val)] \to Maybe\ val \\ & lookup\ \_\ [\,] = Nothing \\ & lookup\ k\ ((c, v) : xs) \mid k == c \quad = Just\ v \\ & \qquad\qquad\qquad\qquad\ \ \mid otherwise = lookup\ k\ xs \end{aligned}$$

- *Maybe* no da información sobre la naturaleza del error.

El constructor de Tipos *Either*

data $Either\ a\ b = Left\ a \mid Right\ b$

- 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*?

Uso de *Either*

$$\textbf{data}\ \mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b$$

- Se suele usar *Either* para manejar errores que devuelven información.

$$\begin{array}{l}
\mathit{safehead} \quad :: [a] \to \mathit{Either}\ \mathit{String}\ a \\
\mathit{safehead}\ [\,] = \mathit{Left}\ \texttt{"head the lista vacía!"} \\
\mathit{safehead}\ \mathit{xs} = \mathit{Right}\ (\mathit{head}\ \mathit{xs})
\end{array}$$

- Por razones que verán más adelante, el error se codifica siempre en el *Left*.

Tipos Recursivos

- Las declaraciones **data** pueden ser recursivos

$$\mathbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}$$

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

- Ejercicio: definir la multiplicación para *Nat*

Tipos Recursivos con Parámetros

- Por supuesto, los tipos recursivos pueden tener parámetros.

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

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

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

Expresiones **case**

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

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

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

Árboles

- Podemos representar diferentes árboles con tipos recursivos

$$
\begin{aligned}
&\textbf{data } T_1\ a &&= \mathit{Tip}\ a \mid \mathit{Bin}\ (T_1\ a)\ (T_1\ a)\\
&\textbf{data } T_2\ b &&= \mathit{Empty} \mid \mathit{Branch}\ (T_2\ b)\ b\ (T_2\ b)\\
&\textbf{data } T_3\ a\ b &&= \mathit{Leaf}\ a \mid \mathit{Node}\ (T_3\ a\ b)\ b\ (T_3\ a\ b)\\
&\textbf{data } T_4\ a &&= E \mid N_2\ a\ (T_4\ a)\ (T_4\ a)\\
& && \quad\ \mid N_3\ a\ (T_4\ a)\ (T_4\ a)\ (T_4\ a)\\
&\textbf{data } T_5\ a &&= \mathit{Rose}\ a\ [T_5\ a]
\end{aligned}
$$

- Ejercicio: ¿Cuál es el tipo de *Node*?
- Ejercicio: Dibujar el árbol correspondiente al siguiente término:

$$N_2\ 4\ (N_3\ 6\ E\ (N_2\ 5\ E\ E)\ (N_3\ 7\ E\ E\ E))\ (N_2\ 9\ E\ E)$$

Programando con árboles

$$\textbf{data}\ T_1\ a = \mathit{Tip}\ a \mid \mathit{Bin}\ (T_1\ a)\ (T_1\ a)$$

- Midamos árboles:

$$\begin{aligned}
&\mathit{size} :: T_1\ a \to \mathit{Int}\\
&\mathit{size}\ (\mathit{Tip}\ _) \quad\quad = 1\\
&\mathit{size}\ (\mathit{Bin}\ t_1\ t_2) = \mathit{size}\ t_1 + \mathit{size}\ t_2
\end{aligned}$$

$$\begin{aligned}
&\mathit{depth} :: T_1\ a \to \mathit{Int}\\
&\mathit{depth}\ (\mathit{Tip}\ _) \quad\quad = 0\\
&\mathit{depth}\ (\mathit{Bin}\ t_1\ t_2) = 1 + (\mathit{depth}\ t_1)\ \text{`}\mathit{max}\text{`}\ (\mathit{depth}\ t_2)
\end{aligned}$$

- Medir un árbol de tamaño n es $O(n)$.
- Se puede hacer en $O(1)$ si se lleva la medida en el árbol,

Programando con árboles (cont)

$$\textbf{type } \mathit{Weight} = \mathit{Int}$$
$$\textbf{data } T\ a = \mathit{Tip}\ \mathit{Weight}\ a \mid \mathit{Bin}\ \mathit{Weight}\ (T\ a)\ (T\ a)$$

- La función que obtiene la medida es $O(1)$:

$$\begin{aligned} &\mathit{weight} :: T\ a \to \mathit{Weight} \\ &\mathit{weight}\ (\mathit{Tip}\ w\ \_) \quad = w \\ &\mathit{weight}\ (\mathit{Bin}\ w\ \_\ \_) = w \end{aligned}$$

- Queremos preservar la siguiente invariante:

$$\mathit{weight}\ (\mathit{Bin}\ w\ t_1\ t_2) = \mathit{weight}\ t_1 + \mathit{weight}\ t_2$$

- Definimos un constructor "inteligente":

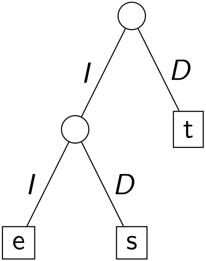
$$\begin{aligned} &\mathit{bin} :: T\ a \to T\ a \to T\ a \\ &\mathit{bin}\ t_1\ t_2 = \textbf{let } w = \mathit{weight}\ t_1 + \mathit{weight}\ t_2 \\ &\qquad\qquad\ \ \textbf{in } \mathit{Bin}\ w\ t_1\ t_2 \end{aligned}$$

Ejemplo: Codificación de Huffman

- La codificación de *Huffman* (1951) permite **comprimir** una secuencia de símbolos.
- Asigna a cada símbolo una secuencia de bits según su probabilidad.
  - A los símbolos más frecuentes les asigna una secuencia corta.
  - A los menos frecuentes, una secuencia más larga.
- Es óptimo (bajo los supuestos correspondientes).

Árboles de Huffman

- Se codifica la tabla de códigos en un árbol binario con información en las hojas.
- Por ejemplo, la secuencia "test" da origen al siguiente árbol:



- Vemos que los códigos correspondientes son:

$$\begin{array}{ccc} t & \mapsto & D \\ e & \mapsto & II \\ s & \mapsto & ID \end{array}$$

Representando árboles de Huffman

- Representamos el árbol con el tipo T definido anteriormente.

$$\textbf{type}\ Weight = Int$$
$$\textbf{data}\ T\ a = Tip\ Weight\ a \mid Bin\ Weight\ (T\ a)\ (T\ a)$$

- Para decodificar seguimos un camino hasta una hoja.

$$\textbf{data}\ Step = I \mid D\ \textbf{deriving}\ Show$$
$$\textbf{type}\ Camino = [Step]$$

$$\begin{aligned}
&trace_1 :: T\ a \to Camino \to a\\
&trace_1\ (Tip\ _\ x)\quad\ \ [\,]\qquad\quad = x\\
&trace_1\ (Bin\ _\ t_1\ t_2)\ (I : cs)\ \ = trace_1\ t_1\ cs\\
&trace_1\ (Bin\ _\ t_1\ t_2)\ (D : cs) = trace_1\ t_2\ cs
\end{aligned}$$

Decodificando árboles de Huffman

- Extendemos la función $trace_1$ para decodificar varios símbolos:

$$
\begin{array}{l}
trace :: T\ a \rightarrow Camino \rightarrow (a, Camino) \\
trace\ (Tip\ \_\ x)\quad\ \ resto\quad\ = (x, resto) \\
trace\ (Bin\ \_\ t_1\ t_2)\ (I : cs)\ \ = trace\ t_1\ cs \\
trace\ (Bin\ \_\ t_1\ t_2)\ (D : cs) = trace\ t_2\ cs
\end{array}
$$

- La función de decodificación queda:

$$
\begin{array}{l}
decodexs :: T\ a \rightarrow Camino \rightarrow [a] \\
decodexs\ t\ ps = \mathbf{case}\ trace\ t\ ps\ \mathbf{of} \\
\qquad\qquad\qquad\qquad (a, [\,])\ \ \rightarrow [a] \\
\qquad\qquad\qquad\qquad (a, ps') \rightarrow a : decodexs\ t\ ps'
\end{array}
$$

Codificación de árboles de Huffman

$$codexs :: Eq\ a \Rightarrow T\ a \rightarrow [a] \rightarrow Camino$$
$$codexs\ t\ xs = concat\ (map\ (codex\ t)\ xs)$$

$$codex :: Eq\ t \Rightarrow T\ t \rightarrow t \rightarrow Camino$$
$$codex\ t\ x = head\ (go\ t\ x)$$
$$\begin{aligned}
\textbf{where}\ go\ (Tip\ _\ y)\ x\ &|\ x == y &&= [[\,]] \\
&|\ otherwise &&= [\,] \\
go\ (Bin\ _\ t_1\ t_2)\ x\ & &&= map\ (I{:})\ (go\ t_1\ x) +\!\!+ \\
& && \quad map\ (D{:})\ (go\ t_2\ x)
\end{aligned}$$

Construcción de árboles de Huffman

- Asumimos una lista de símbolos asociados a su peso en orden creciente.

$$
\begin{aligned}
&\mathit{build} :: [(a, \mathit{Weight})] \to T\ a \\
&\mathit{build}\ t = \mathbf{let}\ \mathit{tip}\ (a, w) = \mathit{Tip}\ w\ a \\
&\qquad\qquad\qquad\quad [x] = \mathit{until}\ \mathit{single}\ \mathit{combine}\ (\mathit{map}\ \mathit{tip}\ t) \\
&\qquad\qquad\ \ \mathbf{in}\ x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{single} :: [a] \to \mathit{Bool} \\
&\mathit{single}\ [x] = \mathit{True} \\
&\mathit{single}\ \_\ \ \ = \mathit{False}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{combine} :: [T\ a] \to [T\ a] \\
&\mathit{combine}\ (t_1 : t_2 : \mathit{ts}) = \mathit{insert}\ (\mathit{bin}\ t_1\ t_2)\ \mathit{ts}
\end{aligned}
$$

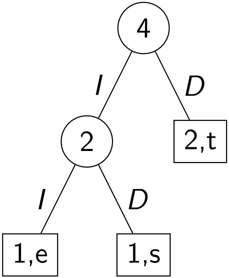
$$
\begin{aligned}
&\mathit{insert} :: T\ a \to [T\ a] \to [T\ a] \\
&\mathit{insert} = \mathit{insertBy}\ (\mathit{comparing}\ \mathit{weight})
\end{aligned}
$$

Construcción de árboles de Huffman

- Por la palabra `"test"`, tenemos las siguientes frecuencias:

$$\mathit{freq} = [(\texttt{'e'}, 1), (\texttt{'s'}, 1), (\texttt{'t'}, 2)]$$

- El árbol con los pesos correspondientes es:

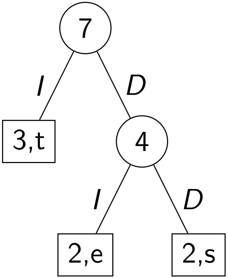


Construcción de árboles de Huffman

- Por la palabra `"testest"`, tenemos las siguientes frecuencias:

$$\mathit{freq} = [(\texttt{'e'}, 2), (\texttt{'s'}, 2), (\texttt{'t'}, 3)]$$

- El árbol con los pesos correspondientes es:



Referencias

- *Programming in Haskell*. Graham Hutton, CUP 2007 (1ra ed), CUP 2016 (2da ed).
- *Introducción a la Programación Funcional con Haskell*. Richard Bird, Prentice Hall 1997.