Programación Funcional Avanzada

Alberto Pardo Marcos Viera

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

curso 2017

Dependencias funcionales

class *Collection ca a* | *ca* $\rightarrow$ *a* **where**

...

Indica que *ca* determina *a*, restringiendo las instancias admisibles

instance *Collection IntSet Int*

intentar definir luego

instance *Collection IntSet Bool*

no está permitido

Type Families

Una alternativa a dependencias funcionales con un estilo más funcional.

```
type family Elem c :: *
class Collection c where
  union  :: c → c → c
  elem   :: Elem c → c → Bool
  empty  :: c

type instance Elem IntSet = Int
instance Collection IntSet where
  ...
```

Variante de sinónimos de tipos que permite definir familias de tipos indexadas

Restricciones de Igualdad

Las type families agregan restricciones de igualdad, una forma diferente de tipos cualificados

$$elem\ \texttt{'a'} :: (Char \sim Elem\ c, Collection\ c) \Rightarrow c \rightarrow Bool$$

Las restricciones de igualdad $t1 \sim t2$ pueden aparecer en cualquier lugar en que Haskell admita restricciones de clases

Para que sea satisfecha, los dos tipos deben unificar módulo las igualdades introducidas por las instancias de type families

Data Families

Una alternativa a Type Families que es inyectiva

```
data family Elem c
class Collection c where
  union  :: c → c → c
  elem   :: Elem c → c → Bool
  empty  :: c

data instance Elem IntSet = MyInt Int
instance Collection IntSet where
  ...
```

Variante de tipos de datos algebraicos que permite definir familias de tipos indexadas

Tipos asociados

Sinónimos de tipos y datatypes restringidos a ser definidos dentro de las clases

```
class Collection c where
  type Elem c
  union  :: c → c → c
  elem   :: Elem c → c → Bool
  empty  :: c

instance Collection IntSet where
  type Elem IntSet = Int
  ...
```

Compatibilidad y Apartamiento

Dos tipos se consideran **aparte** si para todas las posibles reducciones no pueden reducir a lo mismo

Dos patrones de tipos son **compatibles** si:

1. Todos los tipos correspondientes en los patrones están aparte, o
2. Los patrones unifican produciendo una sustitución y los lados derechos son iguales bajo la sustitución

Las equaciones de las type families están restringidas a ser compatibles

Compatibilidad y Apartamiento - Ejemplo

$$\textbf{data}\ Z\,;\,\textbf{data}\ S\ a$$

$$\begin{aligned}
&\textbf{type}\ \textit{family}\ \mathit{Add}\ n\ m :: *\\
&\textbf{type instance}\ \mathit{Add}\ (S\ x)\ y = S\ (\mathit{Add}\ x\ y)\\
&\textbf{type instance}\ \mathit{Add}\ Z\ \quad x = x
\end{aligned}$$

Ok

$$\begin{aligned}
&\textbf{type}\ \textit{family}\ \mathit{Add}\ n\ m :: *\\
&\textbf{type instance}\ \mathit{Add}\ (S\ x)\ y\ = S\ (\mathit{Add}\ x\ y)\\
&\textbf{type instance}\ \mathit{Add}\ Z\ \quad x\ = x\\
&\textbf{type instance}\ \mathit{Add}\ Z\ \quad Z = Z
\end{aligned}$$

Ok

$$\begin{aligned}
&\textbf{type}\ \textit{family}\ \mathit{Add}\ n\ m :: *\\
&\textbf{type instance}\ \mathit{Add}\ (S\ x)\ y = S\ (\mathit{Add}\ x\ y)\\
&\textbf{type instance}\ \mathit{Add}\ z\ \quad x = x
\end{aligned}$$

Mal

Closed Type Families

Se puede declarar una type family cerrada usando la cláusula **where**, definiendo el conjunto completo de ecuaciones.

$$\begin{aligned}&\textbf{type } \textit{family } \textit{Add } n\ m :: * \textbf{ where}\\ &\quad \textit{Add } (S\ x)\ y = S\ (\textit{Add } x\ y)\\ &\quad \textit{Add } z \quad\ \ x = x\end{aligned}$$

Al simplificar una aplicación, las ecuaciones se intentan de arriba hacia abajo.
GHC tiene que estar seguro que z no puede unificarse con $(S\ x)$, es decir que una ecuación incompatible definida arriba nunca se podrá aplicar.

Type Families - Kinds

Se pueden anotar los kinds del resultado y los parámetros, si se omiten se consideran $*$.

Aridad: número de parámetros en la declaración de una type family
Toda aplicación debe estar completamente saturada con respecto a su aridad.

type *family F a b* :: $* \rightarrow *$

```
F Char [Int]        -- OK
F Char [Int] Bool   -- OK
F IO   Bool         -- Mal
F Bool              -- Mal
```

Kind Polymorphism

Polimorfismo a nivel de kinds permite que clases como

$$\textbf{class } \mathit{Typeable}\ (t :: *) \textbf{ where}$$
$$\quad \mathit{typeOf} :: t \to \mathit{TypeRep}$$

$$\textbf{class } \mathit{Typeable1}\ (t :: * \to *) \textbf{ where}$$
$$\quad \mathit{typeOf1} :: t\ a \to \mathit{TypeRep}$$

$$\textbf{class } \mathit{Typeable2}\ (t :: * \to * \to *) \textbf{ where}$$
$$\quad \mathit{typeOf2} :: t\ a\ b \to \mathit{TypeRep}$$

$$\ldots$$

se combinen en una única clase

$$\textbf{data } \mathit{Proxy}\ t = \mathit{Proxy}$$

$$\textbf{class } \mathit{Typeable}\ t \textbf{ where}$$
$$\quad \mathit{typeOf} :: \mathit{Proxy}\ t \to \mathit{TypeRep}$$

Donde *Proxy* tiene kind $\forall\ k.k \to *$ y *Typeable* tiene kind $\forall\ k.k \to \mathit{Constraint}$

Datatype Promotion

Permite enriquecer al lenguaje de kinds y hacer que programación a nivel de tipos sea más "tipada"

Se promueven tipos a nivel de kinds

$$\textbf{data}\ Nat = Z \mid S\ Nat$$

producen el kind Nat y los (constructores de) tipos:

$$Z :: Nat$$
$$S :: Nat \to Nat$$

$$\textbf{type}\ family\ Add\ (n :: Nat)\ (m :: Nat) :: Nat$$
$$\textbf{type instance}\ Add\ (S\ x)\ y = S\ (Add\ x\ y)$$
$$\textbf{type instance}\ Add\ Z\ \quad x = x$$

Por ejemplo, el tipo $Add\ Z\ Bool$ tiene kind incorrecto

Resolviendo conflictos de nombres

Haskell permite que constructores de tipos y constructores de valores con el mismo nombre:

$$\textbf{data}\ T = T\ Int$$

Entonces al hacer promotion el tipo T puede referir tanto al tipo T (de kind $*$) o al constructor promovido (de kind T).

Para evitar ambigüedades se puede usar la notación $'T$ para referirse al constructor promovido.