Sintaxis de registro

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Definición de tipos

Recordatorio

- Hemos visto cómo definir nuevos tipos con `data`. Por ejemplo, para crear personas, podemos hacer:

```
data Persona = Persona String String Int Int
  deriving (Show,Eq)
```

- Podemos crear así una persona:

```
ghci> let p = Persona "Cervantes" "Literatura" 1547 1616
```

- Al solicitar el valor de p en el intérprete, nos aparece:

```
ghci> p
Persona "Cervantes" "Literatura" 1547 1616
```

- Ahora bien, ¿qué hacer si queremos usar el año de nacimiento de p?

Definición de tipos

Accediendo a sus datos

- Bueno, si queremos acceder a los distintos *campos* contenidos en el tipo, debemos usar patrones que tomen el valor de cada elemento.
- Para evitar repetir el acceso a los elementos en cada función que use el tipo, podemos definir funciones que usen patrones para extraerlos:

```
nombre :: Persona -> String
nombre (Persona n _ _ _) = n

actividad :: Persona -> String
actividad (Persona _ a _ _) = a

nacimiento :: Persona -> Int
nacimiento (Persona _ _ an _) = an

defuncion :: Persona -> Int
defuncion (Persona _ _ _ ad) = ad
```

- ¿Y si en lugar de 4 hablamos de 20 datos en el tipo? ¿Recordaremos en el orden que debemos pasar los elementos al tipo?

```
data T = C Int String String String Int Int Double Float Int Int...
```

Sintaxis de registro

Para subsanar lo anterior tenemos la sintaxis de registro, explicitando nombres de campos. Veamos cómo se trabaja con esta notación.

- Definición del tipo:

```
data Persona =
  Pers {
    nombre :: String, actividad :: String,
    nacimiento :: Int, defuncion :: Int
    }
    deriving (Show, Eq)
```

- Creación de una instancia del tipo:

```
ghci> let p1 = Pers "Cervantes" "Literatura" 1547 1616
ghci> let p2 = Pers {actividad="Literatura", nacimiento=1547, nombre="Cervantes", defuncion=1616}
```

Como vemos, el orden en que pasemos los campos no importa.
Además, como podemos observar, en ambos casos se devuelve lo mismo:

```
ghci>p1
Pers {nombre = "Cervantes", actividad = "Literatura", nacimiento = 1547, defuncion = 1616}
ghci>p2
Pers {nombre = "Cervantes", actividad = "Literatura", nacimiento = 1547, defuncion = 1616}
```

Acceso a los campos

- Automáticamente tenemos acceso a los campos del registro:

```
ghci> nombre p1
"Cervantes"
ghci> actividad p1
"Literatura"
ghci> nacimiento p1
1547
ghci> defuncion p1
1616
```

- También podemos devolver un derivado del anterior, cambiando únicamente los datos que desees (sin alterar el dato original):

```
ghci> p1 {nombre="Murillo",actividad="Pintura"}
Pers {nombre = "Murillo", actividad = "Pintura", nacimiento = 1547, defuncion = 1616}
ghci> p1
Pers {nombre = "Cervantes", actividad = "Literatura", nacimiento = 1547, defuncion = 1616}
```

Valores por defecto

- Hemos creado personas, pasándole los 4 campos requeridos, pero... ¿Qué ocurre si tenemos 20 campos y no queremos aportar todos?
  - Haskell no permite indicar valores por defecto en los campos del registro, ya que no existen variables que luego cambien esos valores.

Valores por defecto

- Hemos creado personas, pasándole los 4 campos requeridos, pero... ¿Qué ocurre si tenemos 20 campos y no queremos aportar todos?
  - Haskell no permite indicar valores por defecto en los campos del registro, ya que no existen variables que luego cambien esos valores.
- Podemos subsanarlo de varias formas:

Valores por defecto

- Hemos creado personas, pasándole los 4 campos requeridos, pero... ¿Qué ocurre si tenemos 20 campos y no queremos aportar todos?
  - Haskell no permite indicar valores por defecto en los campos del registro, ya que no existen variables que luego cambien esos valores.
- Podemos subsanarlo de varias formas:
  - Crear un valor del tipo, `personaPorDefecto`, y usar la devolución de valores derivados para instanciar otros incluyendo únicamente los campos que deseemos.

Valores por defecto

- Hemos creado personas, pasándole los 4 campos requeridos, pero... ¿Qué ocurre si tenemos 20 campos y no queremos aportar todos?
  - Haskell no permite indicar valores por defecto en los campos del registro, ya que no existen variables que luego cambien esos valores.
- Podemos subsanarlo de varias formas:
  - Crear un valor del tipo, `personaPorDefecto`, y usar la devolución de valores derivados para instanciar otros incluyendo únicamente los campos que deseemos.
  - Hacer que nuestro tipo instancie la clase `Default`, importando `Data.Default`.

Valores por defecto

Creando un valor del tipo y derivando del mismo

- Creamos la constante `personaPorDefecto` en nuestro fichero:

```
personaPorDefecto :: Persona
personaPorDefecto = Pers {nombre="", actividad="", nacimiento=0, defuncion=0}
```

- Ya podemos obtener distintas personas a partir de la persona por defecto:

```
ghci> personaPorDefecto
Pers {nombre = "", actividad = "", nacimiento = 0, defuncion = 0}
ghci> personaPorDefecto { nombre = "Luis" }
Pers {nombre = "Luis", actividad = "", nacimiento = 0, defuncion = 0}
ghci> personaPorDefecto { nombre = "Ramón y Cajal", actividad = "Medicina" }
Pers {nombre = "Ramón y Cajal", actividad = "Medicina", nacimiento = 0, defuncion = 0}
ghci> personaPorDefecto
Pers {nombre = "", actividad = "", nacimiento = 0, defuncion = 0}
```

Como se puede ver, no queda alterado el valor original.

- Inconveniente: método *ad-hoc*, no estándar, el usuario de nuestro tipo no sabe *a priori* qué esperar.

Valores por defecto

Instanciando la clase `Default`

- Método más *estándar*
- Importamos el módulo `Data.Default`, e instanciamos `Default`

```
import Data.Default

instance Default Persona where
  def = Pers { nombre = def, actividad = def, nacimiento = def, defuncion = def}
```

Solo hemos definido la función `def`, que proporcionará el valor por defecto. Como vemos, nos hemos apoyado en los propios `def` de los tipos `Int` y `String`, ya predefinidos.

- Ya podemos usar `def` para obtener personas, con los datos de que dispongamos:

```
ghci> let p3 = def {nombre = "Ortega y Gasset", actividad = "Filosofía"}
ghci> p3
Pers {nombre = "Ortega y Gasset", actividad = "Filosofía", nacimiento = 0, defuncion = 0}
ghci> :t p3
p3 :: Persona
```

Bibliografía


M. Lipovača. *¡Aprende Haskell por el bien de todos!*.
Apartado 8.2: Sintaxis de registro


mauke (GitHub developer). *Módulo Data.Default en Hackage*.