Contenido

Enumeraciones

Podemos definir enumeraciones de la misma manera que definimos el tipo de dato Empleado, escribiendo los constructores uno a continuación del otro para determinar así el orden:

```
data Bool = False | True
data Color = Rojo | Amarillo | Azul | Verde | Naranja | Violeta
data Temperatura = Frio | Caliente
data Estacion = Primavera | Verano | Otonio | Invierno

> :t Azul
Azul :: Color
```

Un ejemplo práctico del libro de José Labra[1]:

```
tiempo :: Estacion -> Temperatura
tiempo Primavera = Caliente
tiempo Verano    = Caliente
tiempo _         = Frio
```

Tipos de datos parametrizados

Al definir un nuevo tipo de dato, podemos utilizar tipos parametrizados:
data Par a b = Par a b
data Complejo a = Complejo a a

En el primero de los casos, la definición de Par se asemeja a una tupla: el constructor Par espera dos parámetros: a y b pueden ser de cualquier tipo (incluso iguales). En el caso de los complejos, el constructor Complejo espera dos argumentos de cualquier tipo (pero

[1] *Introducción al lenguaje Haskell*, José Labra, Universidad de Oviedo, 1998

ambos deben coincidir).

En la definición del tipo hay que distinguir

1) el a y el b de la izquierda, que refieren a los tipos parametrizados, mientras que
2) cada letra que sigue al constructor marca un parámetro, cuyo tipo es de la letra que está a la izquierda

1) 2)

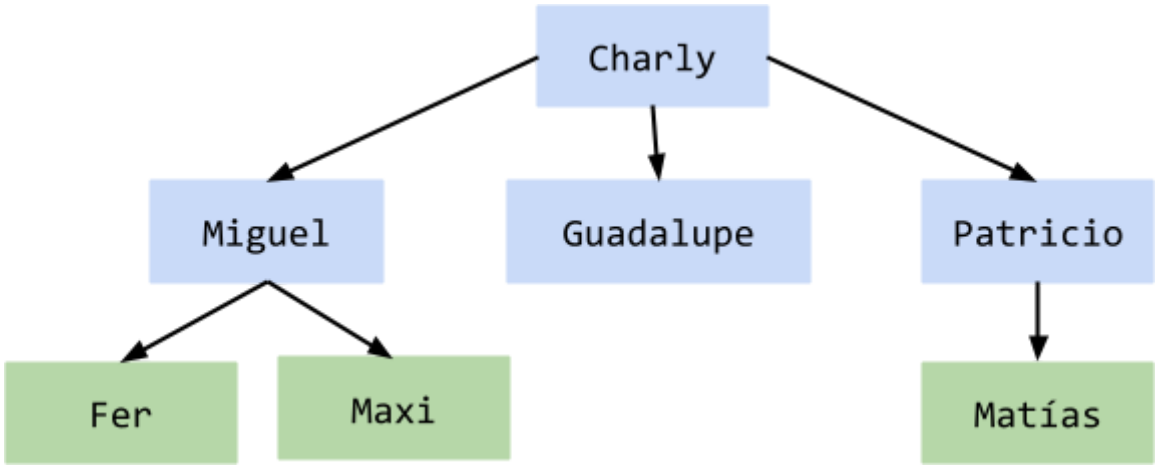
data Par a b = Par a b

> Par 8 "hermanos"
Es válido, dado que a y b pueden ser de diferente tipo: a es Int, b es String.

> Complejo 8 "hermanos"
Da error, porque el constructor espera dos argumentos del mismo tipo

Tipos de datos recursivos

Si queremos modelar el organigrama de una empresa:



La estructura que más se asemeja es un árbol, que puede tener:

- hojas (Fer, Maxi, Guadalupe, Matías)
- o ramas, que a su vez pueden componerse de hojas o ramas.

Para aceptar n niveles, podemos definir dos constructores para el tipo de dato Arbol:

```
data Arbol a = Hoja a | Rama (Arbol a) (Arbol a)
```

Mmm… claro, pero ¿cómo creo a Charly, que no tiene hermanos? Y Charly tiene a cargo tres empleados… entonces vamos a definir el constructor Rama para que reciba un

conjunto de hijos (o sea, una lista de árboles, para que acepte tanto hojas como ramas):

```
data Arbol a = Hoja a | Rama a [Arbol a]
```

Entonces podemos construir el organigrama de la siguiente manera:

```
organigrama = Rama "Charly"
                   [
                       Rama "Miguel" [Hoja "Fer", Hoja "Maxi"],
                       Hoja "Guadalupe",
                       Rama "Patricio" [Hoja "Matias"]
                   ]
```

Hay otras opciones, pero nos quedamos con ésta que es bastante simple.
Si queremos conocer la lista de los empleados de la empresa, tenemos que trabajar con pattern matching ramas y hojas (o sea, jefes y empleados):

- Para conocer la lista de empleados cuando hay una hoja, es una lista con ese empleado solo
- Para conocer la lista de empleados cuando tengo una rama, es una lista que componen el jefe y toda la gente que tiene a cargo.

La definición es recursiva:

```
empleados (Hoja nombre) = [nombre]
empleados (Rama nombre aCargo) =
     foldl (++) [nombre] (map empleados aCargo)
```

Lo evaluamos:

```
>empleados organigrama
["Charly","Miguel","Fer","Maxi","Guadalupe","Patricio","Matias"]
```

Con un pequeño ajuste podemos determinar quiénes no tienen gente a cargo:

```
pichis (Hoja nombre) = [nombre]
pichis (Rama nombre aCargo) = foldl (++) [] (map pichis aCargo)
```

Otros ejemplos de tipos de datos recursivos son:

- árboles binarios y sus aplicaciones, como un árbol de expresiones lógicas (and, or) o aritméticas (suma, resta, multiplicación, etc.)
- las listas, que en Haskell se trabajan con syntactic sugars pero que podríamos definir

```
data List a = Nil | Cons a (List a)
```

Definimos las funciones length, head, tail y foldl:

```
listLength Nil = 0
listLength (Cons x xs) = 1 + listLength xs

listHead (Cons x xs) = x

listTail (Cons x xs) = xs

listFoldl f z Nil = z
listFoldl f z (Cons x xs) = foldl f (f z x) xs
```

El syntactic sugar de Haskell permite resolver

```
Nil = []
Cons x xs = (x:xs)
Cons x Nil = [x]
```

Para más información el lector puede leer Hal Daumé[2] o José Labra[3].

Tipo de dato Maybe

Nos interesa modelar las notas que un alumno obtiene en un examen. La nota puede estar entre 1 y 10, pero el alumno puede no presentarse al examen…

¿Cómo podemos manejar este concepto?
Podemos definir un tipo de dato que acepte nulos. El tipo Maybe nos sirve para esto:

```
data Maybe a = Nothing | Just a
```

El constructor Nothing está representando un null, que para el negocio es un “ausente en el examen”. La nota se representa utilizando el constructor Just seguido de la nota. Entonces podemos escribir una función que nos permita sacar el promedio de los parciales:

```
promediar Nothing otraNota = otraNota
promediar otraNota Nothing = otraNota
promediar (Just unaNota) (Just otraNota) =
    Just ((unaNota + otraNota) / 2)
```

¿Cuál es el dominio y la imagen de la función promediar?
Si bien no escribimos el tipo de dato, Haskell se da cuenta de que la división trabaja sobre

[2] *Yet another Haskell tutorial*, Hal Daumé III, 2002-2006, cap.61 4.5.3 Recursive dataypes
[3] José Labra, op.cit.

el dominio de las fracciones, entonces infiere:
```
promediar :: Fractional a => Maybe a -> Maybe a -> Maybe a
```
Podríamos definir el tipo nota de la siguiente manera:
```
type Nota = Maybe Float
```

y definir dominio e imagen a la función promediar como:
```
promediar :: Nota -> Nota -> Nota
```

Esto se conoce como sinónimo de tipo (type synonym) y hace más legible/expresiva la definición de las funciones.

Evaluamos:
```
> promediar (Just 10) (Just 7)
Just 8.5

> promediar Nothing (Just 8)
Just 8.0
```

¿Y qué pasa al promediar dos ausentes?
```
> promediar Nothing Nothing
Nothing
```

Otros usos de Maybe

Cuando tenemos que definir la función head siempre hay dudas sobre lo que tenemos que hacer con la lista vacía:

- ¿tirar error? Si trabajamos el pattern matching con una lista divisible en cabeza y cola y no “pensamos” en el caso de la lista vacía, el error ocurre solo. Si nuestra intención es guiar al usuario mostrándole un mensaje más representativo, podemos intentar:

```
head [] = error "No existe la cabeza de una lista vacía"
```

Esto modifica el mensaje de error:
```
Program error: {head []}
```
a uno más descriptivo:
```
Program error: No existe la cabeza de una lista vacía
```

- También podemos cambiar head para que pueda devolver un nulo, que sería la no-existencia de un elemento que sea cabeza de dicha lista:

```
head' [] = Nothing
head' (x:xs) = Just x
```

Revisemos dominio y origen de head':

```
head' :: [a] -> Maybe a
```

Recibimos una lista (de cualquier tipo, pero no son maybe, la lista no tiene elementos que puedan ser nulos, a lo sumo será la lista vacía), y devolvemos un maybe.

Ahora quien llama a head' sabe que la lista puede devolver nulos, en la versión original de head esto no está en el contrato de la función, recordemos que:

```
head :: [a] -> a
```

Evaluando valores Maybe

Si queremos multiplicar un número maybe con otro, podemos definir una función:

```
mPor :: Maybe Float -> Maybe Float -> Maybe Float
mPor Nothing _ = Nothing
mPor _ Nothing = Nothing
mPor (Just a) (Just b) = Just (a * b)
```

Lo evaluamos:

```
> mPor (Just 9) (Just 2)
Just 18.0
```

Pero es molesto tener que hacer lo mismo con la resta, la multiplicación, etc. Nosotros ya conocemos la función map:

```
>:t map
map :: (a -> b) -> [a] -> [b]
```

que se aplicaba sobre listas, entonces haciendo un paralelo:

Para el map de listas	**Para el map de Maybe**
Esperamos una función que recibe un a y devuelve un b	Vamos a recibir una función que reciba un a y devuelva un b (igual)
Recibimos una lista de a	Recibimos un Maybe (de a)
Devolvemos una lista de b	Devolvemos un Maybe (de b)

Map para lista:	Map para Maybe:

`map f Nil = Nil`
`map f (Cons x xs) =`
`  Cons (f x) (map f xs)`
	`mmap f Nothing  = Nothing`
	`mmap f (Just x) = Just (f x)`

Esto nos permite aplicar cualquier función a un Maybe:

```
>mmap (* 2) Nothing
Nothing

>mmap (* 2) (Just 9)
Just 18
```

O bien, utilizando el operador ($[4]), podemos evitar algunos paréntesis:

```
>mmap (* 2) $ Just 9
Just 18
```

El lector notará –no obstante- que hay una diferencia importante entre el mPor y el mmap: mientras que el mPor recibía dos maybe (podíamos multiplicar dos nothings), el mmap recibe cualquier función que va de a a b (con lo cual el segundo operando del map tiene que ser no nulo).

Kinds

En este momento podemos detenernos un instante y recordar los tipos de dato con los que trabajamos en Haskell:

- Por un lado tenemos los tipos concretos, que no necesitan recibir tipos como parámetros. En este conjunto están los tipos primitivos de Haskell, como Bool, Int, Float, Char, Double.
- Por otra parte conocemos también los constructores de tipo (type constructors), como Complejo a, que recibe un tipo (por ejemplo Float) y devuelve un nuevo tipo (Complejo Float). Lo mismo ocurre con Par y Maybe.

Para ejemplificar esta diferencia, vamos a utilizar la nomenclatura de los *kinds*, que permiten trabajar los tipos que tienen los constructores de tipo. Evaluamos en Haskell 98:

```
>:i Float
-- type constructor with kind *
data Float
```

Si estás en el GHCI podés utilizar el comando k (kind):

```
>:k Int
Int :: *
```

"*" se lee "tipo", y la nomenclatura indica que el tipo Int es concreto, no espera ningún tipo como parámetro para instanciarse (por eso recibe el nombre de nullary type constructor).

¿Qué kind tiene el complejo?

[4] El lector puede profundizar con ejemplos en pág.77 de Hal Daumé, *op.cit.*

```
>:i Complejo
-- type constructor with kind * -> *
```
El mismo kind del Maybe, que necesita un tipo para terminar generando otro.

Ah, y el Par:
```
>:i Par
-- type constructor with kind * -> * -> *
```

De la misma manera que podemos aplicar parcialmente una función, podemos variar los kinds del constructor de tipo Par pasándole más o menos tipos:

Constructor de tipo	**Kind**
Par	* -> * -> *
Par Int	* -> *
Par Int Char	*

La lista, ¿qué kind tiene?
[] necesita de un tipo… * -> *
Pero [Int] es de un tipo concreto: *

La función filter,
```
filter :: (a -> Bool) -> [a] -> [a]
```

¿qué kind tiene? Mmm… podríamos marearnos un poco con la definición, pero tenemos que diferenciar:

- Los tipos de una función (qué valores acepta, lo que genera el dominio e imagen de dicha función).
- Los tipos de un constructor de tipo (aquí nuestro input son tipos, no valores).

Entonces, filter, ¿construye algún tipo nuevo en base a un tipo que le paso como parámetro?
No, y en general todas las funciones tienen un kind * (son de un tipo concreto).

¿Qué sentido tiene hablar de kinds? Hasta aquí la resolución de los kinds parece trivial, pero pongamos el siguiente caso:

```
data TipoNuevo f a = MkTipoNuevo a (f a)
```

¿Qué kind tiene?
f no es un tipo concreto, el constructor MkTipoNuevo recibe dos argumentos, uno de tipo a (que sí es concreto) y el otro de tipo f a, con lo cual f podría ser un Maybe, o bien un Complejo.

Entonces:

```
TipoNuevo:: (* -> *) -> * -> *
TipoNuevo Maybe :: * -> *
```

Entender estos ejemplos es la base para poder continuar con temas más avanzados como teoría de tipos y mónadas.

Más sobre tipos (en otros textos)

Otros temas que no entran en este módulo y que el lector puede ampliar con lecturas de la bibliografía son:

- Newtype declaration
- Field labels (también llamados named fields)