

Definición de tipos de datos

Algoritmos y Estructuras de Datos I

Martín Ariel Domínguez

Polimorfismo *ad hoc*

- Una función que puede tener **distintos comportamientos** dependiendo del tipo concreto con que se use.
- Este polimorfismo se llama **ad hoc**.
- En Haskell se logra mediante **Type Classes**.

Polimorfismo *ad hoc*

`elem :: a -> [a] -> Bool`

¿Conocen la función `elem`?

Polimorfismo *ad hoc*

```
elem :: Eq a => a -> [a] -> Bool
```

podrá utilizarse con un tipo que instancie la clase `Eq`.

Polimorfismo *ad hoc*

`elem :: Eq a => a -> [a] -> Bool`

podrá utilizarse con un tipo que instancie la clase **Eq**.

- Una clase define requisitos que debe satisfacer un tipo.
- Por ejemplo, una instancia de **Eq** tendrá definidas las funciones de igualdad y desigualdad.

Polimorfismo *ad hoc*

- Utilizando **typeclasses** podemos definir funciones sobre tipos para los cuales pedimos algunas restricciones.
- El comportamiento dependerá de cómo el tipo defina las funciones especificadas en la clase.
- Algunas clases: **Eq**, **Ord**, **Show**, **Num**, ...

Tipos de datos

Haskell es un lenguaje **fuertemente tipado**: toda expresión tiene un **tipo de datos**.

Tiene muchos muchos tipos de datos predefinidos: **Int**, **Bool**, **Char**, **Listas**, **Tuplas**, ...

¿Podemos extender el lenguaje definiendo más tipos?

Tipos de datos

Haskell es un lenguaje **fuertemente tipado**: toda expresión tiene un **tipo de datos**.

Tiene muchos muchos tipos de datos predefinidos: **Int**, **Bool**, **Char**, **Listas**, **Tuplas**, ...

¿Podemos extender el lenguaje definiendo más tipos?

¡Claro que sí!

Nuevos tipos ¿para qué?

- Queremos desarrollar una agenda electrónica muy básica en la cual tenemos tareas fijas que realizamos cada día de la semana.
- Necesitamos una función que para cada *día de la semana* devuelva un **string** con el texto correspondiente a la tarea.

Nuevos tipos ¿para qué?

```
tareaDiaria :: Int -> String
tareaDiaria 0 = "Trabajar"    -- corresponde al lunes
tareaDiaria 1 = "Trabajar"    -- corresponde al martes
tareaDiaria 2 = "Trabajar"    -- corresponde al miercoles
...
```

Nuevos tipos ¿para qué?

```
tareaDiaria :: Int -> String
tareaDiaria 0 = "Trabajar"      -- corresponde al lunes
tareaDiaria 1 = "Trabajar"      -- corresponde al martes
tareaDiaria 2 = "Trabajar"      -- corresponde al miercoles
...
```

Luego debemos *acordarnos* que el “0” representa el lunes, el “1” el martes, etc...

Nuevos tipos ¿para qué?

- ¿Qué pasa si evaluamos `tareaDiaria` 8? ¿Donde queda la verificación de tipos?
- Si bien una *codificación ad hoc* podría funcionar, requiere que tengamos mucho cuidado!
- Definamos un **tipo de dato nuevo** que **represente** a los días de la semana.

Tipos algebraicos sin parámetros

Definimos *cada valor posible* del tipo:

```
data Dia = Lunes | Martes | Miercoles | Jueves | Viernes  
         | Sabado | Domingo
```

- Se les suele llamar también **enumerados**.
- Con **data** definimos un nuevo tipo de dato dando alternativas de construcción.
- Un **constructor** puede ser cualquier palabra que empiece con mayúscula.
- ¿Qué es un constructor?

Tipos algebraicos sin parámetros

```
data Dia = Lunes | Martes | Miercoles | Jueves | Viernes  
         | Sabado | Domingo
```

¿Cómo definimos funciones con los nuevos tipos?

Tipos algebraicos sin parámetros

```
data Dia = Lunes | Martes | Miercoles | Jueves | Viernes  
         | Sabado | Domingo
```

El **pattern matching** es un mecanismo para definir funciones, donde damos reglas para las alternativas de un tipo de dato, mediante sus **constructores**.

Tipos algebraicos sin parámetros

```
data Dia = Lunes | Martes | Miercoles | Jueves | Viernes  
         | Sabado | Domingo
```

```
tareaDiaria :: Dia -> String  
tareaDiaria Lunes      = "Trabajar"  
tareaDiaria Martes     = "Trabajar"  
tareaDiaria Miercoles  = "Trabajar"  
tareaDiaria Jueves     = "Trabajar"  
tareaDiaria Viernes   = "Trabajar poco"  
tareaDiaria Sabado    = "Salir de joda"  
tareaDiaria Domingo   = "Descansar"
```


Tipos algebraicos sin parámetros

```
data Dia = Lunes | Martes | Miercoles | Jueves | Viernes  
         | Sabado | Domingo
```

```
tareaDiaria :: Dia -> String  
tareaDiaria Viernes = "Trabajar poco"  
tareaDiaria Sabado  = "Salir de joda"  
tareaDiaria Domingo = "Descansar"  
tareaDiaria otroDia = "Trabajar"
```

Podemos usar variables para dar definiciones más compactas

Tipos algebraicos sin parámetros

```
data Dia = Lunes | Martes | Miercoles | Jueves | Viernes  
         | Sabado | Domingo
```

```
tareaDiaria :: Dia -> String  
tareaDiaria Viernes = "Trabajar poco"  
tareaDiaria Sabado  = "Salir de joda"  
tareaDiaria Domingo = "Descansar"  
tareaDiaria _       = "Trabajar"
```

Incluso *comodines*!

Vamos por todo

```
data Dia = Lunes | Martes | Miercoles | Jueves | Viernes  
         | Sabado | Domingo
```

```
data Tarea = Trabajar | TrabajarPoco | Joda | Descansar
```

```
tareaDiaria :: Dia -> Tarea  
tareaDiaria Viernes = TrabajarPoco  
tareaDiaria Sabado   = Joda  
tareaDiaria Domingo  = Descansar  
tareaDiaria _        = Trabajar
```

Sintaxis *case*

- Supongamos ahora que queremos definir una función `horasTrabajo` que nos devuelve la cantidad de horas que trabajo en el día.

Sintaxis *case*

```
tareaDiaria :: Dia -> Tarea  
tareaDiaria Viernes = TrabajarPoco  
tareaDiaria Sabado  = Joda  
tareaDiaria Domingo = Descansar  
tareaDiaria _       = Trabajar
```

```
horasTrabajo :: Dia -> Int  
horasTrabajo Viernes = 4  
horasTrabajo Sabado  = 0  
horasTrabajo Domingo = 0  
horasTrabajo _       = 8
```

Sintaxis case

```
tareaDiaria :: Dia -> Tarea  
tareaDiaria Viernes = TrabajarPoco  
tareaDiaria Sabado  = Joda  
tareaDiaria Domingo = Descansar  
tareaDiaria _       = Trabajar
```

```
horasTrabajo :: Dia -> Int  
horasTrabajo Viernes = 4  
horasTrabajo Sabado  = 0  
horasTrabajo Domingo = 0  
horasTrabajo _       = 8
```

***Pero las hs.
dependen de la tarea
diaria, no del dia!***

Sintaxis *case*

```
tareaDiaria :: Dia -> Tarea
tareaDiaria Viernes = TrabajarPoco
tareaDiaria Sabado  = Joda
tareaDiaria Domingo = Descansar
tareaDiaria _       = Trabajar
```

```
horasTrabajo :: Dia -> Int
horasTrabajo d = case tareaDiaria d of
    Trabajar -> 8
    TrabajarPoco -> 4
    _ -> 0
```

Sintaxis case

```
tareaDiaria :: Dia -> Tarea
tareaDiaria Viernes = TrabajarPoco
tareaDiaria Sabado  = Joda
tareaDiaria Domingo = Descansar
tareaDiaria _       = Trabajar
```

```
horasTrabajo :: Dia -> Int
horasTrabajo d = case tareaDiaria d of
    Trabajar -> 8
    TrabajarPoco -> 4
    _ -> 0
```

*Permite hacer
pattern matching en
expresiones
complejas.
(resultado de
funciones)*

Instancias derivadas y def. locales

¿Podríamos haber definido lo siguiente?

```
horasTrabajo :: Dia -> Int
horasTrabajo d | tareaDiaria d == Trabajar = 8
                | tareaDiaria d == TrabajarPoco = 4
                | otherwise == 0
```

Instancias derivadas y def. locales

¿Podríamos haber definido lo siguiente?

```
horasTrabajo :: Dia -> Int
horasTrabajo d | tareaDiaria d == Trabajar = 8
                | tareaDiaria d == TrabajarPoco = 4
                | otherwise = 0
```

No podemos comparar **Tarea**.

Repetimos la llamada a función!

Instancias derivadas y def. locales

¿Podríamos haber definido lo siguiente?

```
horasTrabajo :: Dia -> Int
```

```
horasTrabajo d | tarea == Trabajar = 8
```

```
               | tarea == TrabajarPoco = 4
```

```
               | otherwise = 0
```

```
where tarea = tareaDiaria d
```

Instancias derivadas y def. locales

- Para comparar **Tarea**, debe pertenecer a la clase **Eq**.
- ¿Cómo hacer que tipo definido pertenezca a una clase?
 - Se define cada función de la clase
 - ... mejor que lo haga Haskell sólo!

Instancias derivadas y def. locales

```
data Dia = Lunes | Martes | Miercoles | Jueves | Viernes  
         | Sabado | Domingo  
         deriving (Show, Eq, Ord, Bounded, Enum)
```

- ¿Cómo se muestra cada valor? Clase Show
- ¿Qué valores son iguales y distintos entre sí? Clase Eq
- Entre un par de valores, ¿cuál es el mayor, y el menor? Ord
- ¿Cuál el máximo? ¿Cuál el mínimo? Bounded
- ¿Para poder escribir [Lunes ... Viernes]? Clase Enum

Volvemos al ejemplo ...

¿Como queda con la instancia derivada el ejemplo de tarea diaria?:

```
data Tarea = Trabajar | TrabajarPoco | Joda | Descansar
           deriving Eq
```

```
horasTrabajo :: Dia -> Int
horasTrabajo d | tareaDiaria d == Trabajar = 8
                | tareaDiaria d == TrabajarPoco = 4
                | otherwise == 0
```

Tipos algebraicos con parámetros

- Queremos representar las figuras geométricas **círculo** y **rectángulo** en un plano.
- Un **círculo** se define con un par de números que representen el centro y otro número que representa el radio.
- Un **rectángulo** se define con dos pares de números: la esquina inferior izquierda y la esquina superior derecha.

Tipos algebraicos con parámetros

```
data Figura = Circulo (Float, Float) Float  
            | Rectangulo (Float, Float) (Float, Float)
```

- Los constructores **Circulo** y **Rectangulo** tienen **parámetros**.
- ¿De qué tipo son estos constructores?
- Para construir un elemento **Figura** tenemos dos alternativas, pero cada una se construye de manera distinta.

Tipos algebraicos con parámetros

```
data Figura = Circulo (Float, Float) Float  
            | Rectangulo (Float, Float) (Float, Float)
```

Definamos una función que calcule el área de una **Figura**:

Tipos algebraicos con parámetros

```
data Figura = Circulo (Float, Float) Float
            | Rectangulo (Float, Float) (Float, Float)
```

Definamos una función que calcule el área de una **Figura**:

```
area :: Figura -> Float
area (Circulo (x, y) r)          = 3.1416 * r * r
area (Rectangulo (x, y) (w, z)) = base * altura
                                where base   = w-x
                                    altura  = z-y
```

Tipos algebraicos con parámetros

*Los identificadores **x**, **y**, **w**, **z**, **r** representan a cualquier valor de tipo **Float**, y los usamos para referenciar los parámetros de los constructores... como cualquier otra función!*

```
area :: Figura -> Float
area (Circulo (x, y) r)          = 3.1416 * r * r
area (Rectangulo (x, y) (w, z)) = base * altura
                                where base    = w-x
                                    altura    = z-y
```

Sinónimos de tipo

- Algunas veces queremos definir un tipo de dato simplemente como **sinónimo de otro**:
 - aporta claridad en el código;
 - pero son equivalentes!
- Por ejemplo, para **Figura**, el tipo (**Float**, **Float**) representa un *punto* en el plano.
- En lugar de **data**, usamos **type**.

Sinónimos de tipo

```
type Punto = (Float, Float)
```

```
data Figura = Circulo Punto Float  
           | Rectangulo Punto Punto
```

Sinónimos de tipo

```
type Punto = (Float, Float)
```

```
type Radio = Float
```

```
data Figura = Circulo Punto Radio  
           | Rectangulo Punto Punto
```

¿Y cómo se definen las funciones?

Sinónimos de tipo

```
type Punto = (Float, Float)
```

```
type Radio = Float
```

```
data Figura = Circulo Punto Radio  
            | Rectangulo Punto Punto
```

*Igual! los sinónimos
son equivalentes*

```
area :: Figura -> Float
```

```
area (Circulo (x, y) r) = 3.1416 * r * r
```

```
area (Rectangulo (x, y) (w, z)) = base * altura  
                                where base    = w-x  
                                    altura    = z-y
```

Sinónimos de tipo

```
type Punto = (Float, Float)
```

```
type Radio = Float
```

```
data Figura = Circulo Punto Radio  
            | Rectangulo Punto Punto
```

```
area :: Figura -> Float
```

```
area (Circulo p r)      = 3.1416 * r * r
```

```
area (Rectangulo p q) = base * altura
```

```
    where base      = ?
```

```
          altura    = ?
```


Sinónimos de tipo

```
type Punto = (Float, Float)
```

```
type Radio = Float
```

```
data Figura = Circulo Punto Radio  
            | Rectangulo Punto Punto
```

*Es importante elegir
el pattern más
adecuado!*

```
area :: Figura -> Float
```

```
area (Circulo p r) = 3.1416 * r * r
```

```
area (Rectangulo p q) = base * altura
```

```
    where base = fst q - fst p
```

```
          altura = snd q - snd p
```

¿Podemos definir nosotros la igualdad?

- ¿Qué sucede cuando necesitamos algo más complejo que la definida por Haskell?.
- Por ejemplo, que dos figuras sean iguales si solo si tienen la misma área.
- La definimos nosotros, volvamos a Figura :

```
data Figura = Circulo Punto Radio
            | Rectangulo Punto Punto
```

```
area :: Figura -> Float -- recuerden la defincion
```

Definiendo nuestra igualdad:

```
instance Eq Figura
```

```
  where
```

```
    Circulo p r1 == Circulo q r2 =
```

```
      area (Circulo p r1) == area (Circulo q r2)
```

```
    Rectangulo p1 p2 == Rectangulo p3 p4 =
```

```
      area (Rectangulo p1 p2) == area (Rectangulo p3 p4)
```

```
    Circulo p r1 == Rectangulo p1 p2 =
```

```
      area (Circulo p r1) == area (Rectangulo p1 p2)
```

```
    Rectangulo p1 p2 == Circulo p r1 =
```

```
      area (Circulo p r1) == area (Rectangulo p1 p2)
```

Más simple

```
instance Eq Figura
```

```
  where
```

```
    f1 == f2 = area f1 == area f2
```

Resumen

- Tipos algebraicos sin parámetros (enumerados)

```
data Dia = Lunes | Martes | Miercoles | Jueves | Viernes  
         | Sabado | Domingo
```

- Tipos algebraicos con parámetros

```
data Figura = Circulo (Float, Float) Float  
            | Rectangulo (Float, Float) (Float, Float)
```

- Instancias derivadas.

- Sinónimos de tipo

```
type Punto = (Float,Float)
```

- Definición explícita de Instancias

Tipos de Datos Recursivos

¿Cuándo un tipo de datos es recursivo?

- Cuando puede tomar un “tamaño” arbitrario.
- ¿Cómo nos damos cuenta que es recursivo mirando la definición?
 - Del lado derecho del comando Data se nombra al tipo nuevamente.

Un ejemplo

¿Cómo representamos una palabra?

```
data Palabra = PVacia | Agregar Char Palabra
```

Un ejemplo

¿Cómo mostramos una palabra?

```
mostrar :: Palabra -> String
```

```
mostrar PVacia      = ""
```

```
mostrar (Agregar l p) = l : mostrar p
```


Un ejemplo

En GHCi

```
> let p = Agregar 'h' (Agregar 'o' (Agregar 'l'
(Agregar 'a' PVacia)))
```

```
> mostrar p
“hola”
```

Veamos otro ejemplo ...

Otro ejemplo

Listas de enteros

```
data ListaInt = LVacia | ConsI Int ListaInt
```

¿cómo se corresponde cada constructor con los que ya conocemos (: y [])?

Otro ejemplo

¿Cómo se ve la lista [1,2,3,4,5] representada con el tipo que definimos antes?

```
ConsI 1 (ConsI 2 (ConsI 3 (ConsI 4 (ConsI 5 LVacia))))
```

Tipos Recursivos y Polimórficos

¿Cómo definimos listas polimórficas?

```
data Lista a = Vacía | Cons a (Lista a)
```

```
> let l = Cons True Vacía
```

```
> let l' = Cons (10::Int) (Cons 0 ( Cons 7 Vacía))
```

¿Que tipo tienen `l` y `l'`? ... :t

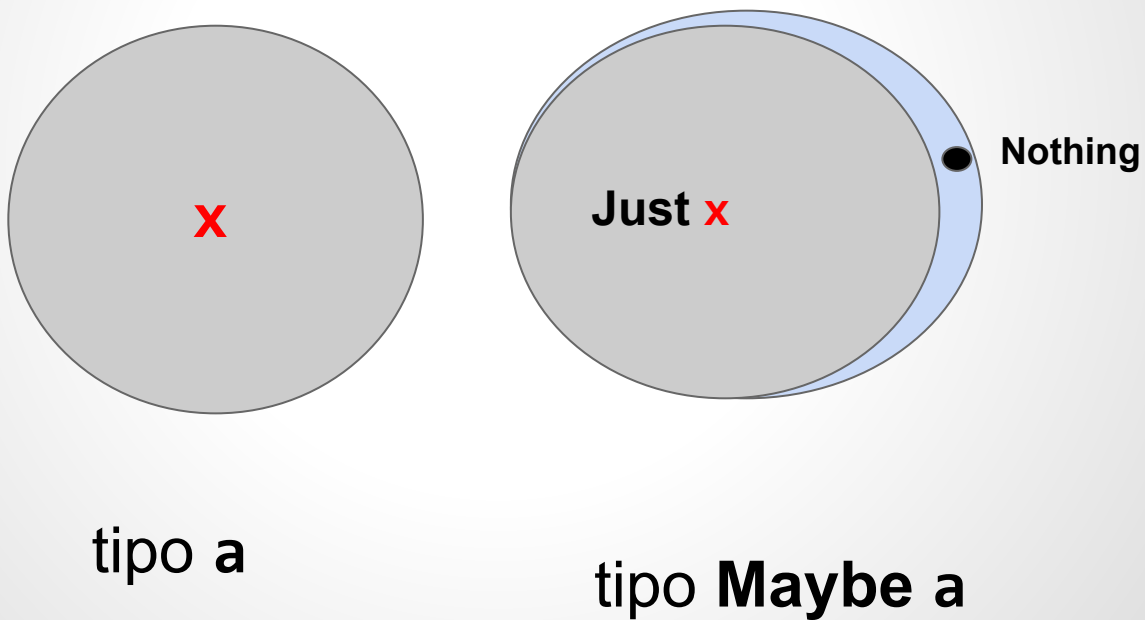
Tipo Maybe

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

- Se ve simple pero es muy útil ¿cuando?:
 - Cuando queremos definir una función, y existe uno o más casos en los que el resultado no se encuentra en el tipo que es imagen de la función.
- Creamos un nuevo tipo que extiende al tipo original agregando un punto distinguido: `Nothing`

Tipo Maybe

Agrega un elemento distinguido al tipo llamado **Nothing**,
el resto de los elementos de **x**, ahora se llaman **Just x**



Ejemplo de uso tipo Maybe

```
data Clase = Teorico | Taller
```

```
hayClase :: Dia -> Maybe Clase
```

```
hayClase Lunes    = Just Taller
```

```
hayClase Martes   = Just Teorico
```

```
hayClase Jueves   = Just Teorico
```

```
hayClase _        = Nothing
```

```
data Clase = Teorico | Taller
hayClase :: Dia -> Maybe Clase
hayClase Lunes    = Just Taller
hayClase Martes   = Just Teorico
hayClase Jueves   = Just Teorico
hayClase _        = Nothing
```

Combinando Maybe y Case

```
actividad :: Dia -> String
actividad d = case hayClase d of
    Nothing      -> "Tareas"
    Just Teorico  -> "Teorico"
    Just Taller   -> "Taller"
```


Qué leer para aprender más:

- <http://learnyouahaskell.com/making-our-own-types-and-typeclasses>
- <http://aprendehaskell.es> (cap. 8)