PROGRAMACIÓN FUNCIONAL

Tipos de Datos: Tipos Algebraicos

Tipos de Datos

- Noción de tipo de datos
- Formas de definición de tipos de datos
- Tipos de datos algebraicos
- Pattern matching

Tipos de Datos

- Un tipo de datos se compone de:
  - un conjunto de *elementos* con ciertas características comunes
  - un conjunto de *operaciones* para manipular dichos elementos
- ¿Cómo definimos tipos de datos?
- ¿Cómo utilizamos tipos de datos?

Definición de Tipos 1

- Para definir un tipo de datos podemos:
  - establecer qué *forma* tendrá cada *elemento*, y
  - dar un *mecanismo único* para *inspeccionar* cada elemento
  - entonces: TIPO ALGEBRAICO

ó

  - determinar cuáles serán las *operaciones* que manipularán los elementos, SIN decir cuál será la forma exacta de éstos o aquéllas
  - entonces: TIPO ABSTRACTO

Definición de Tipos 2

- Tipos Algebraicos
  - dar la forma de los elementos
  - dar un mecanismo único de acceso
- Tipos Abstractos
  - dar sólo las operaciones
  - NO dar la forma de elementos ni operaciones
- Tipos predefinidos
  - a->b: tipo abstracto con sintaxis especial
  - Bool, (a,b): tipos algebraicos con sintaxis especial
  - Int, Float, Char: tipos abstractos con componentes algebraicos y sintaxis especial

Tipos Algebraicos

- ¿Cómo damos en Haskell la forma de un elemento de un tipo algebraico?
  - Mediante **constantes** llamadas *constructores*
    - nombres con mayúsculas
    - no tienen asociada una regla de reducción
    - pueden tener argumentos
- Ejemplos:

False :: Bool True :: Bool

Tipos Algebraicos

- La cláusula data
  - introduce un nuevo tipo algebraico
  - introduce los nombres de los constructores
  - define los tipos de los argumentos de los constructores
- Ejemplos:

```
data SensacionTermica = HaceFrio | HaceCalor

data Shape = Circle Float | Rect Float Float
```

Tipos Algebraicos

- data Shape = Circle Float | Rect Float Float

Ejemplos de elementos:

```
c1 = Circle 1.0
c2 = Circle (4.0-3.0)
r1 = Rect 2.5  3.0
```

Ejemplos de funciones que arman Shapes:

```
circuloPositivo x = Circle (abs x)
cuadrado x = Rect x x
```

Tipos Algebraicos

- data Shape = Circle Float | Rect Float Float

Ejemplo de alto orden:

```
construyeShNormal :: (Float -> Shape) -> Shape
construyeShNormal c = c 1.0
```

Uso de funciones de alto orden:

```
c3 = construyeShNormal circuloPositivo
c4 = construyeShNormal cuadrado
c5 = construyeShNormal (Rect 2.0)
```

- ¿Cuál es el tipo de Circle? ¿Y el de Rect?

Pattern Matching

- ¿Cuál es el mecanismo único de acceso?
  - *Pattern matching* (correspondencia de patrones (?))
- Pattern: expresión especial
  - sólo con constructores y variables sin repetir
  - argumento en el lado izquierdo de una ecuación
- Matching: operación asociada a un pattern
  - inspecciona el valor de una expresión
  - puede fallar o tener éxito
  - si tiene éxito, liga las variables del pattern

Pattern Matching

- Ejemplos:

```
area :: Shape -> Float
area (Circle radio) = pi * radio^2
area (Rect base altura) = base * altura

isCircle :: Shape -> Bool
--isCircle1 (Circle radio) = True
--isCircle1 (Rectangle base altura) = False
isCircle (Circle _) = True
isCircle _ = False
```

Pattern Matching

- Uso de pattern matching:
  - Al evaluar (area (circuloPositivo (-3.0)))
    - se reduce (circuloPositivo (-3.0)) a (Circle 3.0)
    - luego se verifica cada ecuación, para hacer el matching
    - si lo hace, la variable toma el valor correspondiente
  - radio se liga a 3.0, y la expresión retorna 28.2743
- ¿Cuánto valdrá (area (cuadrado 2.5))?
- ¿Y (area c2)?

Tuplas

- Son tipos algebraicos con sintaxis especial

```
fst :: (a,b) -> a
fst (x,y) = x

snd :: (a,b) -> b
snd (x,y) = y

distance :: (Float, Float) -> Float
distance (x,y) = sqrt (x^2 + y^2)
```

  - ¿Cómo definir distance sin usar pattern matching?

```
distance p = sqrt ((fst p)^2 + (snd p)^2)
```

Tipos Algebraicos

- Pueden tener argumentos de tipo
- Ejemplo:

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

- ¿Qué elementos tiene (Maybe Bool)?
  ¿Y (Maybe Int)?
- En general:
  - tiene los mismos elementos que el tipo a (pero con Just adelante) más uno adicional (Nothing)

Tipos Algebraicos

- ¿Para qué se usa el tipo Maybe?
- Ejemplo:

```
buscar :: clave -> [(clave,valor)] -> valor
buscar k [] = error "La clave no se encontró"
                    -- Única elección posible con polimorfismo!
buscar k ((k',v):kvs) = if k==k'
                        then v
                        else buscar k kvs
```

- ¿La función buscar es total o parcial?

Tipos Algebraicos

- ¿Para qué se usa el tipo Maybe?
- Ejemplo:

```
lookup :: clave -> [(clave,valor)] -> Maybe valor
lookup k []             = Nothing
lookup k ((k',v):kvs) = if k==k'
                        then Just v
                        else lookup k kvs
```

- ¿La función lookup es total o parcial?

Tipos Algebraicos

- El tipo Maybe
  - permite expresar la posibilidad de que el resultado sea erróneo, sin necesidad de usar 'casos especiales'
  - evita el uso de ⊥ hasta que el programador decida, permitiendo controlar los errores

```
sueldo :: Nombre -> [Empleado] -> Int
sueldo nombre empleados
        = analizar (lookup nombre empleados)
analizar Nothing = error "No es de la empresa!"
analizar (Just s) = s
```

Tipos Algebraicos

- El tipo Maybe (cont.)
  - evita el uso de ⊥ hasta que el programador decida, permitiendo controlar los errores

```
sueldoGUI :: Nombre -> [Empleado] -> GUI Int
sueldoGUI nombre empleados =
    case (lookup nombre empleados) of
      Nothing -> ventanaError "No es de la empresa!"
      Just s  -> mostrarInt “El sueldo es ” s
```

Tipos Algebraicos

- ¿Maybe o excepciones?

```
data MayFail a = Raise Exception | Ok a
data Exception = DivByZero | NotFound | NullPointer
               | Other String
```

- ¿Cómo se computa con excepciones?

```
lookupE :: clave -> [(clave,valor)] -> MayFail valor
lookupE k []           = Raise NotFound
lookupE k ((k',v):kvs) = if k==k'
                         then Ok v
                         else lookup k kvs
```

Tipos Algebraicos

- Manejo de excepciones (cont.)

```
type ExHandler a = Exception -> a

tryCatch :: MayFail a -> (a -> b) -> ExHandler b -> b
tryCatch (Raise e) exito handler = handler e
tryCatch (Ok a) exito handler = exito a
```

  - ¡tryCatch es una función de alto orden!
    - El primer argumento es el cómputo que puede fallar
    - El segundo argumento es cómo continuar si todo va bien
    - El tercer argumento es el manejador de excepciones

Tipos Algebraicos

- Excepciones (cont.)

```
sueldoGUIE :: Nombre -> [Empleado] -> GUI Int
sueldoGUIE nombre empleados =
    tryCatch (lookupE nombre empleados)
        mostrarInt
        (\e -> case e of
                NotFound -> ventanaError "Catched!"
                _        -> error "Todo mal!")
```

Tipos Algebraicos

- Excepciones (cont.)

```
type ExHandler a = Exception -> a

tryCatch :: MayFail a -> (a -> b) -> ExHandler b -> b
tryCatch (Raise e) exito handler = handler e
tryCatch (Ok a) exito handler = exito a

sueldoGUI :: Nombre -> [Empleado] -> GUI Int
sueldoGUI nombre empleados =
    tryCatch (lookupE nombre empleados)
             mostrarInt
             (\e -> case e of
                      NotFound -> ventanaError "Catched!"
                      _        -> error "Todo mal!")
```

Tipos Algebraicos

- Otro ejemplo:

  ```
  data Either a b = Left a | Right b
  ```

- ¿Qué elementos tiene (Either Int Bool)?
- En general:
  - representa la unión disjunta de dos conjuntos (los elementos de uno se identifican con Left y los del otro con Right)

Tipos Algebraicos

- ¿Para qué sirve Either?
  - Para mantener el tipado fuerte y poder devolver elementos de distintos tipos
    - Ejemplo: [Left 1, Right True] :: [Either Int Bool]
  - Para representar el origen de un valor
    - Ejemplo: lectora de temperaturas

```
data Temperatura = Celsius Int | Fahrenheit Int

convertir :: Either Int Int -> Temperatura
convertir (Left t)  = Celsius t
convertir (Right t) = Fahrenheit t
```

Tipos Algebraicos

- ¿Por qué se llaman tipos algebraicos?
- Por sus características:
  - toda combinación válida de constructores y valores es elemento de un tipo algebraico (y sólo ellas lo son)
  - dos elementos de un tipo algebraico son iguales si y sólo si están construídos utilizando los mismos constructores aplicados a los mismos valores

Tipos Algebraicos

- Expresividad: números complejos
  - Toda combinación de dos flotantes es un complejo
  - Dos complejos son iguales si tienen las mismas partes real e imaginaria

```
data Complex = C Float Float

realPart, imagePart :: Complex -> Float
realPart (C r i) = r
imagePart (C r i) = i

mkPolar :: Float -> Float -> Complex
mkPolar r theta = C (r * cos theta) (r * sin theta)
```

Tipos Algebraicos

- Expresividad: números racionales
  - No todo par de enteros es un número racional (R 1 0)
  - Hay racionales iguales con distinto numerador y denominador (R 4 2 = R 2 1)

```
data NoRacional = R Int Int
numerador, denominador :: NoRacional -> Int
numerador (R n d) = n
denominador (R n d) = d
```

- ¡No se puede representar a los racionales como tipo algebraico!

Tipos Algebraicos

- Expresividad: ejemplos
  - Se pueden armar tipos ad-hoc, combinando las ideas

```
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto (Maybe Baño)
            | Capelina Gusto Gusto [Agregado]
            | Pote Gusto Gusto Gusto
data Gusto = Chocolate | …
data Agregado = Almendras | Rocklets | …
data Baño = Blanco | Negro
```

- Así se pueden expresar elementos de dominios específicos

Tipos Algebraicos

- Expresividad: ejemplos
  - Se pueden armar funciones por pattern matching

```
precio :: Helado -> Float
precio h = costo h * 1.3 + 5

costo :: Helado -> Float
costo (Vasito g) = 1 + costoGusto g
costo (Cucurucho g1 g2 mb) = 2 + costoGusto g1
                               + costoGusto g2
                               + costoBaño mb
...
```

Tipos Algebraicos

- Expresividad: ejemplos
  - Se pueden armar funciones por pattern matching (cont.)

```
costoGusto :: Gusto -> Float
costoGusto Chocolate = 2
...

costoBaño :: Maybe Baño - > Float
costoBaño Nothing          = 0
costoBaño (Just Negro)   = 2
costoBaño (Just Blanco)  = 1
```

Tipos Algebraicos

- Podemos clasificarlos en:
  - Enumerativos (SensacionTermica, Bool, Gusto)
    - Sólo constructores sin argumentos
  - Productos (Complex, Tuplas, registros)
    - Un único constructor con varios argumentos
  - Sumas (Shape, Maybe, Either, Helado)
    - Varios constructores con argumentos
  - Recursivos (Listas, árboles)
    - Utilizan el tipo definido como argumento

Resumen

- Formas de definición de tipos de datos
- Tipos algebraicos
- Pattern matching
- Expresividad de los tipos algebraicos
- Ejemplos
  - Maybe, Either
  - Listas