Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2013

Departamento de Computación - FCEyN - UBA

Programación funcional - clase 2

Tipos algebraicos

Tipos algebraicos y abstractos

Tipo: conjunto de valores asociados a operaciones.

- **Tipo algebraico**
  - Conocemos la forma que tiene cada elemento
  - Tenemos un mecanismo para inspeccionar cómo está construido cada expresión del tipo
- **Tipo abstracto**
  - No conocemos cómo se representan los valores
  - Sólo conocemos sus operaciones (es la única manera de obtener información sobre ellos)

Tipos algebraicos y abstractos

- Ejemplos de tipos algebraicos (primitivos):
  - Tipo **Bool**
  - Tipo **Char**
  - Tipo **(Tipo1, Tipo2)**
- Ejemplos de tipos abstractos (primitivos):
  - Tipo **Int**
  - Tipo **Float**

Tipos algebraicos

- Para crear un tipo algebraico decimos qué forma va a tener cada elemento
- Se hace definiendo constantes que se llaman constructores
  - Empiezan con mayúscula (como los tipos)
  - Pueden tener argumentos, pero no hay que confundirlos con funciones
    - No tienen reglas de inferencia asociada
    - Forman expresiones atómicas (valores)
- Por ejemplo, el tipo algebraico **Bool** tiene dos constructores, sin argumentos

```
True :: Bool
False :: Bool
```

Definición de tipos algebraicos

- Cláusulas de definición de tipos algebraicos
  - Empiezan con la palabra `data`
  - Definen el tipo y sus constructores
  - Cada constructor da una alternativa distinta para construir un elemento del tipo
  - Los constructores se separan entre sí por barras verticales
- `data Sensación = Frío | Calor`
  - Tiene dos constructores sin parámetros
  - El tipo tiene únicamente dos elementos, como el tipo Bool

Definición de tipos algebraicos

- `data Figura = Círc Float | Rect Float Float`
  - Dos constructores con parámetros
  - Algunas figuras son círculos y otras rectángulos
    - Los círculos se diferencian por un número (su radio)
    - Los rectángulos, por dos (su base y su altura)
  - Ejemplos:
    - `c1 = Círc 1`
    - `c2 = Círc (4.5 - 3.5)`
    - `círculo x = Círc (x+1)`
    - `r1 = Rect 2.5 3`
    - `cuadrado x = Rect x x`

Pattern matching

- Correspondencia o coincidencia de patrones
- Mecanismo para ver cómo está construido un elemento de un tipo algebraico
- Si definimos una función que recibe un argumento que es una `Figura`, podemos averiguar si es un círculo o un rectángulo (y con qué argumentos fue construida)
- Patterns: expresiones del lenguaje formadas solamente por constructores y variables que no se repiten
  - `Rect x y` es un patrón
  - `3 + x` no es un patrón
  - `Rect x x` tampoco porque tiene una variable repetida
- Matching: operación asociada a un patrón
  - Dada una expresión cualquiera dice si su valor coincide por su forma con el patrón
  - Si la correspondencia existe, entonces liga las variables del patrón a las subexpresiones correspondientes

Ejemplo de pattern matching

```
área :: Figura -> Float

área (Círc radio)       = pi * radio * radio
área (Rect base altura) = base * altura

círculo :: Float -> Figura
círculo x = Círc (x+1)
```

- Lado izquierdo: función que estamos definiendo aplicada a un patrón
- Evaluemos la expresión `área (círculo 2)`
  1. El intérprete debe elegir cuál de las ecuaciones de `área` utilizar
  2. Primero debe evaluar `círculo 2` para saber a qué constructor corresponde
  3. La reducción da `Círc (2+1)`
  4. Ya se puede verificar cada ecuación de `área` para buscar el matching
  5. Se logra con la primera ecuación, y `radio` queda ligada a `(2+1)`
- Luego de varias reducciones (aritméticas) más, se llega al valor de la expresión: 28.2743

Tipos algebraicos recursivos

El tipo definido es argumento de alguno de los constructores

Ejemplo:

- `data N = Z | S N`
- Tiene dos constructores:
  1. `Z` es un constructor sin argumentos
  2. `S` es un constructor con argumentos (de tipo `N`)
- Elementos del tipo `N`:

```
Z   ,  S Z  ,  S (S Z)  ,  S (S (S Z))  ,  ...
↓      ↓       ↓           ↓
0      1       2           3
```

Este tipo puede representar a los números naturales.

Recursión estructural

Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural. La recursión se hace sobre la estructura de los datos: las invocaciones recursivas se hacen dobre expresiones "de forma más simple".

Ejemplos:

- ```
  suma :: N -> N -> N
  suma n Z     = n
  suma n (S m) = S (suma n m)
  ```
- ```
  producto :: N -> N -> N
  producto n Z     = Z
  producto n (S m) = suma n (producto n m)
  ```
- ```
  menorOIgual :: N -> N -> Bool
  menorOIgual Z     m     = True
  menorOIgual (S n) Z     = False
  menorOIgual (S n) (S m) = menorOIgual n m
  ```


Listas

- Son un tipo algebraico recursivo paramétrico.
- Se usan mucho (al igual que en el lenguaje de especificación).
  - En especificación las vimos definidas con observadores
  - En Haskell, se definen con constructores
- Definición del tipo Lista en Haskell (preludio)

  ```
  data List a = Nil | Cons a (List a)
  ```
- Interpretamos
  - `Nil` como la lista vacía
  - `Cons x l` como la lista que resulta de agregar `x` como primer elemento de `l`

Listas

Por ejemplo,

- `List Int` es el tipo de las listas de enteros. Son de este tipo:
  - `Nil`;
  - `Cons 2 Nil`;
  - `Cons 3 (Cons 2 Nil)`
- `List (List Int)` es el tipo de las listas de listas de enteros. Son de este tipo:
  - `Nil` y representa [ ];
  - `Cons Nil Nil` y representa [[ ]];
  - `Cons (Cons 2 Nil) (Cons Nil Nil)` y representa [[2],[ ]]
  ```

Ejemplos de recursión estructural con listas

- Calcular la longitud de una lista

```
longitud :: List a -> Int
longitud Nil         = 0
longitud (Cons x xs) = 1 + (longitud xs)
```

- Sumar los elementos de una lista de enteros

```
sumar :: List Int -> Int
sumar Nil         = 0
sumar (Cons x xs) = x + (sumar xs)
```

- Concatenar dos listas

```
concat :: List a -> List a -> List a
concat Nil         ys = ys
concat (Cons x xs) ys = Cons x (concat xs ys)
```


Notación de listas en Haskell

- `List a` se escribe `[a]`
- `Nil` se escribe `[]`
- `(Cons x xs)` se escribe `(x:xs)`
- los constructores son `:` y `[]`
  - `Cons 2 (Cons 3 (Cons 2 (Cons 0 Nil)))`
  - `(2 : (3 : (2 : (0 : []))))`
  - `2 : 3 : 2 : 0 : []`
- Notación más cómoda: `[2,3,2,0]`

Notación de listas en Haskell

Ejemplos (todas están en el preludio)

- ```
  length :: [a] -> Int
  length []     = 0
  length (x:xs) = 1 + (length xs)
  ```
- ```
  sum :: [Int] -> Int
  sum []     = 0
  sum (x:xs) = x + (sum xs)
  ```
- ```
  (++) :: [a] -> [a] -> [a]
  []     ++ ys = ys
  (x:xs) ++ ys = x:(xs ++ ys)
  ```


Otro ejemplo

Veamos un tipo algebraico recursivo no paramétrico.

- `data P = T | F | A P P | N P`
- Tiene cuatro constructores:
  1. `T` y `F` son constructores sin argumentos
  2. `A` es un constructor con dos argumentos (de tipo `P`)
  3. `N` es un constructor con un argumento (de tipo `P`)
- Elementos del tipo `P`:

`T`	`F`	`A T F`	`N (A T F)`	...
↓	↓	↓	↓	
true	false	(true $\wedge$ false)	$\neg$(true $\wedge$ false)	

Este tipo puede representar a las fórmulas proposicionales con un conectivo unario y otro binario.
  ```

Ejemplos

- `contarAes :: P -> Int`

```
contarAes T       = 0
contarAes F       = 0
contarAes (N x)   = contarAes x
contarAes (A x y) = 1 + (contarAes x) + (contarAes y)
```

- `valor :: P -> Bool`

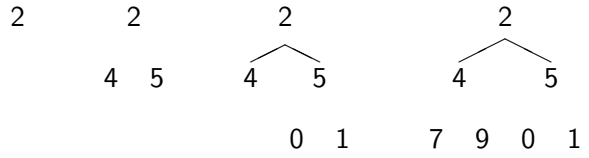
```
valor T       = True
valor F       = False
valor (N x)   = not (valor x)
valor (A x y) = (valor x) && (valor y)
```


Árboles

Tipo algebraico recursivo paramétrico.

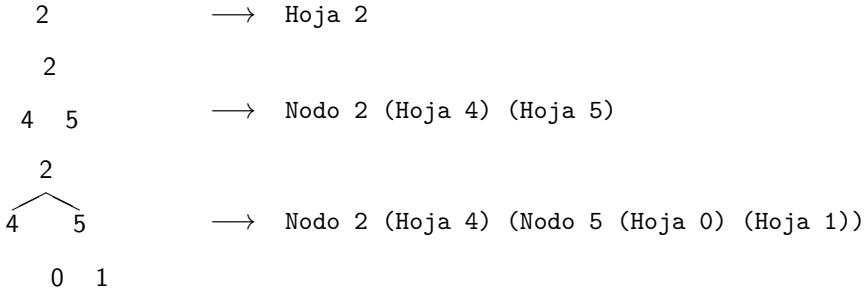
- Estructuras formadas por nodos
- Almacenan valores de manera jerárquica (cada nodo guarda un valor)
- Vamos a trabajar con árboles binarios
  - De cada nodo salen cero o dos ramas
  - Hoja: es un nodo que no tiene ramas

Ejemplos:



Definición del tipo Árbol

```
data Árbol a = Hoja a | Nodo a (Árbol a) (Árbol a)
```




```
hojas :: Árbol a -> Int
hojas (Hoja x)     = 1
hojas (Nodo x i d) = (hojas i) + (hojas d)
altura :: Árbol a -> Int
altura (Hoja x)     = 1
altura (Nodo x i d) = 1 + ((altura i) 'max' (altura d))
```


Funciones para recorrer un árbol

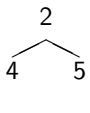
Recorren un árbol de manera ordenada y operan con cada nodo.
Hay distintos órdenes para recorrer un árbol.

```
inOrder, preOrder :: Árbol a -> [a]
```

```
inOrder (Hoja x)     = [x]
inOrder (Nodo x i d) = (inOrder i) ++ [x] ++ (inOrder d)
```

```
preOrder (Hoja x)     = [x]
preOrder (Nodo x i d) = x : ((preOrder i) ++ (preOrder d))
```

Por ejemplo, para el árbol `A`




```
inOrder A     es     [4,2,0,5,1]
preOrder A    es     [2,4,5,0,1]
```

Y postorder?

Sinónimos de tipos

Se usa la cláusula `type` para darle un nombre nuevo a un tipo existente

- No se crea un nuevo tipo, sino un sinónimo de tipo
- Los dos nombres son equivalentes

Ejemplos:

- Nombrar una instancia particular de un tipo paramétrico:
  ```
  type String = [Char]
  ```
- Renombrar un tipo existente con un nombre más significativo:
  ```
  type Nombre = String
  type Sueldo = Int
  type Empleado = (Nombre, Sueldo)
  type Dirección = String
  type Persona = (Nombre, Dirección)
  ```
  `Persona` es un par de `String`, pero `(String, String)`, es más difícil de entender
- También hay sinónimos de tipos paramétricos:
  ```
  type IntY a = (Int, a)
  ```