Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2013

Departamento de Computación - FCEyN - UBA

Programación funcional - clase 3

Tipos abstractos

Acerca de los tipos algebraicos

1. Su forma se declara explícitamente a través de los constructores
2. Toda expresión del tipo representa un valor válido
   - Constructor
   - Valores cualesquiera para sus parámetros
3. Igualdad por construcción
   - Dos valores son iguales solamente si se construyen del mismo modo
     - Mismo constructor
     - Mismos valores denotados por sus respectivos argumentos
4. Mecanismo de pattern matching

Ejemplo de tipo algebraico: Complejo

- Toda combinación de dos Float es un complejo
- Dos complejos son iguales si y sólo si sus partes reales y sus partes imaginarias coinciden

- ```
  data Complejo = C Float Float

  parteReal, parteImag :: Complejo -> Float

  parteReal (C r i) = r
  parteImag (C r i) = i

  hacerPolar :: Float -> Float -> Complejo
  hacerPolar rho theta =
          C (rho * cos theta) (rho * sin theta)
  ```


Racionales

```
data Racional = R Int Int
numerador, denominador :: Racional -> Int

numerador (R n d)  = n
denominador (R n d) = d
```

¿Es una buena representación de los números racionales?
```

Racionales

- Porque no todo par de enteros es un racional: R 1 0
- Hay racionales iguales con distinto numerador y denominador: R 4 2 y R 2 1
- No estamos representando el signo!

Racionales como tipo algebraico

Se puede usar, pero con mucho cuidado.

- Al construir ...
  - Nunca la segunda componente debe ser 0
- Al definir funciones ...
  - Se debe retornar el mismo resultado para toda representación del mismo racional
  - Se debería retornar el resultado "normalizado"

Conclusión sobre tipos algebraicos

- Sea C el conjunto de elementos que queremos representar.
- Un tipo algebraico T es adecuado para representar C si existe una relación uno a uno entre T y C.
- En otras palabras, cada elemento de T representa exactamente un elemento de C, y cada elemento de C es representado por exactamente un elemento de T.

Tipos abstractos

Un tipo abstracto está compuesto por:

- un **nombre**,
- **constructores de elementos del tipo**, previamente especificados como problemas,
- **observadores**, y
- **condiciones invariantes**.

Un tipo abstracto no es otra cosa que un tipo algebraico con su representación encapsulada y oculta.

- Su representación se encuentra solamente en **una parte** del programa (encapsulamiento).
- Su representación **no es visible** en el resto del programa (ocultamiento).

Uso de tipos abstractos

- Cuando recibimos un tipo de datos abstracto, tenemos ...
  - ... el nombre del tipo,
  - ... los nombres de sus operaciones básicas,
  - ... los tipos de las operaciones, y
  - ... una especificación de su funcionamiento.
    - Puede ser más o menos formal
    - En Haskell no es chequeada por el lenguaje
- Un tipo de datos abstracto sólo se utiliza a través de sus operaciones.
- No se puede usar pattern matching, dado que los constructores están **ocultos**.

Racionales como tipo abstracto

- Recibimos operaciones, que no sabemos cómo están implementadas ...

```
crearR :: Int -> Int -> Racional
numerR :: Racional -> Int
denomR :: Racional -> Int
```

- ... y también tenemos su especificación:

```
tipo Racional {
    observador numerR(r : Racional): Int;
    observador denomR(r: Racional): Int;
    invariante denomR(r) > 0;
    invariante mcd(numerR(r), denomR(r)) == 1;
}
problema crearR(n, d: Int) = rac: Racional {
    requiere d ≠ 0;
    asegura numerR(rac) * d == denomR(rac) * n;
}
```


Definición de nuevas operaciones

- Tal vez incluya operaciones básicas
  - Suma, multiplicación, división
- También podemos definirlas nosotros:

```
sumaR, multR, divR :: Racional -> Racional -> Racional

r1 `sumaR` r2 = crearR
    (denomR r2 * numerR r1 + denomR r1 * numerR r2)
    (denomR r1 * denomR r2)

r1 `multR` r2 = crearR
    (numerR r1 * numerR r2)
    (denomR r1 * denomR r2)

r1 `divR` r2 = crearR
    (denomR r2 * numerR r1)
    (denomR r1 * numerR r2)
```


Creación de tipos abstractos

- `module` introduce el módulo: nombre, qué exporta
- `Tipo(..)` exporta el nombre del tipo y sus constructores
  - Permite hacer pattern matching fuera del módulo
- Después del `where` van las definiciones
- Si no hay lista de exportación se exportan todos los nombres definidos
  - `module Complejos where...`

Se puede:

- Encapsular y no ocultar. Sirve para definir y exportar tipos algebraicos
- Encapsular y ocultar. Sirve para definir tipos abstractos. Exportamos solamnete el nombre, constructores y observadores del tipo.

Ejemplo de módulo que exporta un tipo algebraico

El siguiente módulo define la funcionalidad de los complejos, que se representan bien mediante un tipo algebraico.

```
module Complejos (Complejo(..), parteReal, parteIm) where

data Complejo = C Float Float

parteReal, parteIm:: Complejo -> Float
parteReal(C r i) = r
parteIm (C r i) = i
```


Ejemplo de tipo abstracto

```
module Racionales (Racional, crearR, numerR, denomR)
where

data Racional = R Int Int

crearR :: Int -> Int -> Racional
crearR n d = reduce (n*signum d) (abs d)

reduce :: Int -> Int -> Racional
reduce x 0 = error ''Racional con denom. 0''
reduce x y = R (x `quot` d) (y `quot` d)
    where d = gcd x y

numerR, denomR :: Racional -> Int

numerR (R n d) = n
denomR (R n d) = d
```


Aclaraciones

- Las funciones signum (signo), abs (valor absoluto), quot (división entera) y gcd (gratest common divisor (MCD)) están en el preludio de Haskell.
- En la lista de exportación de un módulo no dice (..) después de `Racional`
  - Se exporta solamente el nombre del tipo
  - **No se exportan** sus constructores!
  - Esto convierte el tipo en **abstracto** para los usuarios del tipo
- Tampoco exportamos `reduce` (auxiliar)

Uso del tipo

Volvemos al rol de usuario

- Hay que indicar (en otro módulo) que queremos incorporar este tipo de datos
- Se usa la cláusula `import`
  - Todos los nombres exportados por un módulo ...
  - ... o solamente algunos de ellos (aclarando entre paréntesis cuáles)

```
module Main where
    import Complejos
    import Racionales (Racional, crearR)
    miPar :: (Complejo, Racional)
    miPar = (C 1 0, crearR 4 2)
```

Otro ejemplo: Conjuntos

El tipo de datos primitivo que tenemos para representar colecciones es el de las listas. Pero no es adecuado para representar conjuntos.

Solución:

- Crear un tipo abstracto para los conjuntos
- Los elementos de este tipo de datos representan conjuntos
- Proveer al tipo de las operaciones necesarias para manipular conjuntos

Operaciones de conjuntos

Vamos a definir conjuntos de enteros

- El conjunto vacío
  ```
  vacío :: IntSet
  ```
- ¿El conjunto dado es vacío?
  ```
  esVacío :: IntSet -> Bool
  ```
- ¿Un elemento pertenece al conjunto?
  ```
  pertenece :: Int -> IntSet -> Bool
  ```
- Agregar un elemento al conjunto, si no estaba. Si estaba, dejarlo igual
  ```
  agregar :: Int -> IntSet -> IntSet
  ```
- Elegir el menor número y quitarlo del conjunto
  ```
  elegir :: IntSet -> (Int, IntSet)
  ```


Otra forma de crear tipos

- Vimos cómo crear sinónimos de tipos
  - Nombre adicional para un tipo existente
  - Son intercambiables
- A veces, queremos un tipo nuevo con la representación de uno existente
  - Que **no** puedan intercambiarse
  - Por ejemplo, para crear un tipo abstracto
- Para esto, tenemos la cláusula `newtype`

newtype

Ejemplo: conjuntos

- Los representamos internamente con listas
- Encerramos la representación en un tipo abstracto

```
newtype IntSet = Set [Int]
```

Diferencias con `data`:

- Llama la atención sobre renombre ...
  - ... a otro implementador encargado de modificarla
  - ... a alguien que tenga que revisar el código
  - ... al mismo programador dentro de un tiempo
- Admite un solo constructor con un parámetro
  - No crea nuevos elementos
  - Renombra elementos existentes (no intercambiable)
- Mejor rendimiento

Implementación de conjuntos como tipo abstracto

```
module ConjuntoInt (IntSet, vacío, esVacío, pertenece,
agregar, elegir) where

import List (insert)

newtype IntSet = Set [Int]

vacío :: IntSet
vacío = Set []

esVacío :: IntSet -> Bool
esVacío (Set xs) = null xs

pertenece :: Int -> IntSet -> Bool
pertenece x (Set xs) = x `elem` xs

agregar :: Int -> IntSet -> IntSet
agregar x (Set xs) | elem x xs = Set xs
agregar x (Set xs) | otherwise = Set (insert x xs)

elegir :: IntSet -> (Int, IntSet)
elegir (Set (x:xs)) = (x, Set xs)
```


Nuevas operaciones

Definamos una operación nueva: `unión`

```
unión :: IntSet -> IntSet -> IntSet

unión p q | esVacío p = q
unión p q | otherwise = uniónAux (elegir p) q

uniónAux (x, p') q = agregar x (unión p'q)
```

Pudimos hacerlo sin conocer la representación de los conjuntos

- Usamos las operaciones provistas

Si cambia la representación

- Habrá que rescribir las ecuaciones para las operaciones del tipo abstracto (`vacío`, `esVacío`, `pertenece`, `agregar`, `elegir`)
- `unión` y cualquier otra definida en otros programas quedan intactas

Se puede hacer recursión sin pattern matching

- `unión` está definida en forma recursiva (a través de `uniónAux`)
- Pero no usa recursión estructural, ¡no conocemos su estructura!