Tema 7. Tipos abstractos. Módulos

- TAD: nombre tipo + conjunto de operaciones sobre él.
- Las operaciones del tipo forman la signatura
  - Nombre de cada operación
  - Tipo (y orden) de sus argumentos y resultado
- Representación del tipo: se implementa mediante algún tipo (usualmente algebraico)
- Implementación de las operaciones del tipo: en términos de la representación elegida

Todo ello “encapsulado” en un MÓDULO

Signatura del TAD “Pila”

Nombre del TAD: Pila α

Operaciones del TAD:

pvacia :: Pila α

apilar :: α –> Pila α –> Pila α

desapilar :: Pila α –> Pila α

cima :: Pila α –> α

esvacia :: Pila α –> Bool

Dos posibles implementaciones:

- *mediante constructoras Vac y Ap*
- *mediante listas (con una constructora P)*

Una implementación del TAD “Pila”

```
-- una representación del TAD
data Pila α  = Vac | Ap α  (Pila α )
-- implementación de las operaciones
pvacia = Vac
apilar x p = Ap x p
desapilar (Ap _ p) = p
desapilar Vac = error "desapilar de pila vacia"
cima (Ap x _) = x
cima Vac = error "cima de pila vacia"
esvacia Vac = True
esvacia (Ap _ _) = False
```

Un módulo para el TAD "Pila"

module Pila1 ← *nombre del módulo*

(Pila, pvacia, apilar, desapilar, cima, esvacia **)**

← *tipos y operaciones que exporta el módulo*

(exporta el tipo Pila *pero NO exporta sus constructoras* Vac *y* Ap*)*

where

```
data Pila α = Vac | Ap α (Pila α)        -- representación del TAD
-------------------------------------------------------------------
pvacia = Vac                             -- implementación de operaciones
apilar x p = Ap x p
- - - - - - - - - - - - -
```

Importando el módulo Pila1

```
module PruebaPilas  where
import Pila1                    -- Importa el módulo Pila1
p1, p2, p3 :: Pila Int
p1 = apilar 3 (apilar 4 (apilar 7 pvacia))
p2 = apilar 2 (desapilar p1)
p3 = Ap 8 Vac
```

¡Representación oculta!

ERROR adecuado

PruebaPilas> cima p2 2	PruebaPilas> cima (Ap 2 p1)

Otra implementación del TAD “Pila”

```
-- otra representación (en términos del tipo lista)
type Pila α = [α]
-- implementación de las operaciones
pvacia = []
apilar x p  = (x:p)
desapilar (x:p) =  p
desapilar [] = error "desapilar de pila vacia"
cima  (x:p) = x
cima [] = error "cima de pila vacia"
esvacia  p = null p
```

Problema del uso de “type” (1)

```
module Pila2  (Pila, pvacia, apilar, desapilar, cima, esvacia)
where
type Pila α = [α]

pvacia = []
apilar x p = (x:p)
desapilar (x:p) =  p
desapilar [] = error "desapilar de pila vacia"
cima  (x:p) = x
cima [] = error "cima de pila vacia"
esvacia  p = null p
```

Problema con “type”:

desde los módulos que importan

Pila2 *es visible que* Pila α es [α]

Problema del uso de “type” (2)

```
module PruebaPilas  where
import Pila2
p1, p2 :: Pila Int
p1 = apilar 3 (apilar 4 (apilar 7 pvacia))
p2 = apilar 2 (desapilar p1)
x = cima p2
y = tail p1
z = p1 ++ p2
```

¡¡Representación NO está oculta!!

PruebaPilas> y [4,7]	PruebaPilas> length p1 3

Otro módulo para el TAD “Pila”

```
module Pila2  (Pila, pvacia, apilar, desapilar, cima, esvacia)
where
 data Pila α = P [α]

 pvacia = P []
 apilar x (P p) = P (x:p)
 desapilar (P (x:p)) = P p
 desapilar (P []) = error "desapilar de pila vacia"
 cima (P (x:p)) = x
 cima (P []) = error "cima de pila vacia"
 esvacia (P p) = null p
```

Solución:

desde los módulos que importan

Pila2 *NO es visible* P

Importando el módulo Pila2

```
module PruebaPilas  where
import Pila2                    -- Importa el módulo Pila2
p1, p2, p3 :: Pila Int
p1 = apilar 3 (apilar 4 (apilar 7 pvacia))
p2 = apilar 2 (desapilar p1)
p3 = P [2,3,4]
```

¡Representación oculta!

ERROR adecuado

PruebaPilas> cima p2 2	PruebaPilas> length p1

Importando el TAD "Pila"

```
module FuncionesPilas  where
import Pila1
altura :: Pila α –> Int
altura p = if esvacia p then 0 else 1 + altura (desapilar p)
p1, p2 :: Pila Int
p1 = apilar 3 (apilar 4 (apilar 7 pvacia))
p2 = apilar 2 (desapilar p1)
```

Importa el módulo deseado

- *Elección de implementación (cambiar Pila1 por Pila2)*
- *Sin cambiar el contenido de los módulos que usan el TAD (por ejemplo, sin cambiar la función* altura)

Signatura del TAD “Conjunto”

Nombre del TAD: Conj α

Operaciones del TAD:

vacio :: Conj α

simple :: α –> Conj α

miembro :: α –> Conj α –> Bool

union, inter, dif :: Conj α –> Conj α –> Conj α

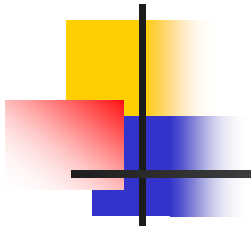
-- dif x y devuelve el conjunto x menos el conjunto y

card :: Conj α –> Int

subConj :: Conj α –> Conj α –> Bool

-- subConj x y decide si x es subconjunto de y

hacerConj :: [α] –> Conj α



Implementación del TAD "Conjunto"

Representación del TAD:

data Conj α = Co [α]

-- *representación mediante listas cualesquiera*

Implementación de las operaciones del TAD:

vacio = Co []

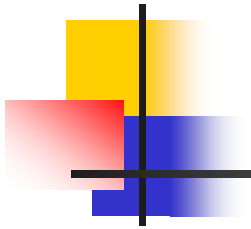
simple x = Co [x]

union (Co c) (Co d) = Co (c++d)

. . . . . .

card (Co s) = length (quitarRep s)

hacerConj s = Co s



Módulo para un TAD

module -- nombre del módulo

(-- tipos y operaciones que exporta el módulo **)**

where

-- módulos importados (opcional)

-- representación del TAD

-- instancia de clases (opcional)

-- Implementación de operaciones exportadas y/o
de otras operaciones auxiliares (no exportadas)

Módulo para el TAD “Conjunto” (1)

module Conjunto

(Conj, vacio, simple, miembro, union, inter, dif, card, subConj, hacerConj)

where

```
import OpListas                -- módulo importado
```

```
data Conj α = Co [α]           -- representación del TAD
```

```
                               -- instancia de la clase Show
instance (Show α, Eq α) => Show (Conj α)
   where  show = mostrarConj
```

Módulo para el TAD “Conjunto” (2)

```
-- Implementación de operaciones exportadas
vacio = Co []
simple x = Co [x]
. . . . . .
-- Implementación de operaciones no exportadas
mostrarConj (Co xs) = mostrar (quitarRep xs)
mostrar [] = “{}”
mostrar (x:xs) = “{”++ show  x ++ resto xs
                 where  resto [] = “}”
                        resto (y:ys) = “,”++ show y ++resto ys
```

Importando el TAD “Conjunto”

```
module PruebaConj where

import Conjunto

c1,c2,c3,c4,c5,c6 :: Conj Char
c1 = hacerConj [‘a’, ‘b’,’c’, ‘a’]
c2 = hacerConj [‘d’, ‘e’, ‘a’, ‘c’]
c3 = union c1 c2
c4 = inter c1 c2
c5 = dif c1 c2
c6 = dif c2 c1
```

```
> c3 –-> {a,b,c,d,e}
> c4 –-> {a,c}
> c5 –-> {b}
> c6 –-> {d,e}
> card c1 –-> 3
```

Sobre módulos: “import” y “export”

Módulo: *Define una colección de valores, tipos de datos, clases, etc, en un entorno creado por un conjunto de “imports”.*
Exporta (parte de) estos recursos para usarlos en otros módulos.

module <nombre-módulo> **(** <lista-export> **) where**
<contenido-módulo>

Dentro de <contenido-módulo> *pueden aparecer “imports”:*

import <nombre-módulo> **(** <lista-import> **)**

antes de las declaraciones y/o definiciones locales del módulo.

Todo ello (local + importado) conforma el entorno del módulo

Sobre módulos: ejemplo1

```
module  M1 ( T(C1,C2), op1, op2)  where
 data T α = C1 | C2 α | C3 α
 op1, op2 :: T α –> α
 ---   ----  ---  ----  ---
```

- M1 *contiene* T(C1,C2) *en su* <lista-export> *por lo que exporta sólo los constructores* C1 *y* C2 *del tipo* T α
- *Si se quiere exportar todos los constructores:* T(..)
- *Si NO se quiere exportar los constructores:* T
- *Las instancias de clases definidas para* T α *se exportan (e importan) junto con el tipo.*

Sobre módulos: ejemplo2

```
module  M2 <lista-exp-M2> where
 import M1 ( T, op1 )
 op3:: T α –> Bool
 ---   ----  ---  ----  ---
```

- M2 *importa de* M1 *sólo* T *y* op1 *(ni* C1 *ni* C2 *ni* op2*)*
- *Si se quiere importar todo lo que* M1 *exporta:* **import** M1
- *El entorno de* M2 *consiste en* T, op1 *y* op3
- *Si no hay* <lista-exp-M2>, M2 *exporta la parte local* (op3) *pero NO la parte importada* (T, op1)
- *M2 exporta M1 si* <lista-exp-M2> = **(module** M1, op3 **)**
- *M2 exporta* T *y* op3 *si* <lista-exp-M2> = **(** T, op3 **)**