Módulos y TADs

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Importación de módulos en archivo

Como hemos visto, podemos importar módulos mediante instrucciones `import` al inicio del fichero (antes de cualquier declaración propia).

- La forma de llamarlo es:

```
import nombre_modulo
```

- Por ejemplo, para importar `QuickCheck`, del paquete `Test`:

```
import Test.QuickCheck
```

Esto importa todas las funciones, tipos y clases de tipos que el módulo correspondiente decida exponer. Podemos ahora emplear los elementos importados mediante su nombre, como es el caso de la función `nub` en:

```
import Data.List

numUniques :: (Eq a) => [a] -> Int
numUniques = length . nub
```

Importación de módulos en sesión

Adición de módulos a la sesión actual

También podemos importar módulos en el intérprete, mediante `:m`

- En el intérprete, podemos teclear algo como:

```
:m + Data.List
```

  o bien

```
:m Data.List
```

- O para múltiples módulos:

```
:m + Data.List Data.Char
```

Importación de módulos en sesión

Supresión de módulos en la sesión actual

También podemos retirar módulos cargados en la sesión actual, mediante `:m -` (nótese el signo menos)

```
:m - Data.Char
```

- Pruebe a teclear en su intérprete:

  ```
  :m Data.List Data.Char
  nub $ map (\p -> map toLower p) ["hola","HOLA","hOlA"]
  ```

  Esto carga los módulos indicados, de modo que podamos usar `toLower` y `nub`, y devolverá `"hola"`.

- Ahora bien, si quitamos de la sesión alguno de los módulos, dará un **error** al dejar de estar disponibles la función `toLower`:

  ```
  :m - Data.Char
  nub $ map (\p -> map toLower p) ["hola","HOLA","hOlA"]
  ```

Colisiones en nombres de funciones

Opciones de importación parcial de fichero

Si importamos varios módulos y/o definimos funciones en nuestro propio archivo, pueden aparecer varias funciones con el mismo nombre. Podemos tratar de evitarlo, por ejemplo, importando solamente una parte de los módulos.

- **Solo** las funciones explícitamente **indicadas**:

```
import Data.List (nub, sort)
```

- **Todas** las funciones **menos** las **indicadas**:

```
import Data.List hiding (nub)
```

Colisiones en nombres de funciones

Importación cualificada

Otras soluciones pasan por distinguir las funciones homónimas de algún modo:

- Importación cualificada por nombre de módulo:

```
import qualified Data.List
f l = Data.List.nub (reverse l)
```

  Vemos el resultado de una llamada:

```
Prelude> f [2,2,2,5,5,3,5,4]
[4,5,3,2]
```

- Importación cualificada con un alias (`as`):

```
import qualified Data.List as L
f l = L.nub (reverse l)
```

  El resultado sería idéntico al anterior.

Creación de módulo simple

Al acometer programas/trabajos/proyectos de cierto calado, es conveniente dividir nuestro código en grupos funcionales, **módulos**, potencialmente reutilizables en distintos contextos.

- Creamos un fichero `NombreModulo.hs`. Por ejemplo:

```
Geometry.hs
```

- Al principio del fichero, declaramos con `module NombreModulo` las funciones y tipos a exponer. Por ejemplo:

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
) where
```

  El resto del fichero definirá como de costumbre nuestras funciones.

- Ya podemos importar el módulo en otro fichero de la forma habitual:

```
import Geometry
```

Creación de una jerarquía de módulos

Conforme nuestros proyectos crecen conveine estructurar más nuestro código, dividiendo en módulos y submódulos de forma jerarquizada.

- Creamos una carpeta para un módulo principal (*e.g.* `Geometry`)
- Por cada submódulo, un archivo (*e.g.* `sphere.hs`) con funciones:

```
module Geometry.Sphere
( volume
, area
) where

volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)

area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

  Análogamente, con cada figura, como cuboides o cubos.

- Podemos importarlos como:

```
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```

Abstracción y tipos abstractos de datos

- La abstracción es un mecanismo para comprender problemas que involucran una gran cantidad de detalles.
- Un TAD (**tipo abstrato de dato**) es una abstracción:
  - Se destacan los detalles (normalmente pocos) de la especificación (el qué).
  - Se ocultan los detalles (normalmente numerosos) de la implementación (el cómo).
- Analogía con las **estructuras algebraicas**.

Descripción informal de las pilas

- Una **pila** es una estructura de datos, caracterizada por ser una secuencia de elementos en la que las **operaciones de inserción y extracción se realizan por el mismo extremo**.
- La pilas también se llaman estructuras **LIFO** (del inglés Last In First Out), debido a que el último elemento en entrar será el primero en salir.
- Analogía con las pilas de platos.

Signatura del TAD de las pilas

- Signatura:

```
vacia :: Pila a
apila :: a -> Pila a -> Pila a
cima  :: Pila a -> a
desapila :: Pila a -> Pila a
esVacia :: Pila a -> Bool
```

- Descripción:
  - vacia es la pila vacía.
  - (apila x p) es la pila obtenida añadiendo x al principio de p.
  - (cima p) es la cima de la pila p.
  - (desapila p) es la pila obtenida suprimiendo la cima de p.
  - (esVacia p) se verifica si p es la pila vacía.

Las pilas mediante tipos de datos algebraicos

- Cabecera del módulo:

```
module PilaTA
    (Pila,
     vacia,    -- Pila a
     apila,    -- a -> Pila a -> Pila a
     cima,     -- Pila a -> a
     desapila, -- Pila a -> Pila a
     esVacia   -- Pila a -> Bool
    ) where
```

- Tipo de dato algebraico de las pilas.

```
  data Pila a = Vacia
              | P a (Pila a)
                deriving Eq
```

Las pilas mediante tipos de datos algebraicos

- Procedimiento de escritura de pilas:

```
instance (Show a) => Show (Pila a) where
    showsPrec p Vacia cad = showChar '-' cad
    showsPrec p (P x s) cad = shows x (showChar '|' (shows s cad))
```

- Ejemplo de pila.

```
> p1 :: Pila Int
> p1 = apila 1 (apila 2 (apila 3 vacia))
> p1
1|2|3|-
```

Las pilas mediante tipos de datos algebraicos

- vacia es la pila vacía.

```
vacia :: Pila a
vacia = Vacia
```

- (apila x p) es la pila obtenida añadiedo x encima de la pila p.

```
apila :: a -> Pila a -> Pila a
apila x p = P x p
```

- (cima p) es la cima de la pila p.

```
cima :: Pila a -> a
cima Vacia  = error "la pila vacia no tiene cima"
cima (P x _) = x
```

Las pilas mediante tipos de datos algebraicos

- (desapila p) es la pila obtenida suprimiendo la cima de la pila p.

```
desapila :: Pila a -> Pila a
desapila Vacia = error "no se puede desapila la pila vacia"
desapila (P _ p) = p
```

- (esVacia p) se verifica si p es la pila vacía.

```
esVacia :: Pila a -> Bool
esVacia Vacia = True
esVacia _     = False
```

- Ejemplos

```
> esVacia vacia
True
> apila 3 vacia
3|-
> desapila (((apila 4) . (apila 3)) vacia)
3
```

Las pilas mediante listas

- Cabecera del módulo:

```
module PilaTA
    (Pila,
     vacia,    -- Pila a
     apila,    -- a -> Pila a -> Pila a
     cima,     -- Pila a -> a
     desapila, -- Pila a -> Pila a
     esVacia   -- Pila a -> Bool
    ) where
```

- Tipo de dato algebraico de las pilas.

```
data Pila a = P [a]
     deriving Eq
```

Las pilas mediante listas

- Procedimiento de escritura de pilas:

```
instance (Show a) => Show (Pila a) where
    showsPrec p (P []) cad = showChar '-' cad
    showsPrec p (P (x:xs)) cad
        = shows x (showChar '|' (shows (P xs) cad))
```

- Ejemplo de pila.

```
> p1 :: Pila Int
> p1 = apila 1 (apila 2 (apila 3 vacia))
> p1
1|2|3|-
```

Las pilas mediante listas

- vacia es la pila vacía.

```
vacia  :: Pila a
vacia = P []
```

- (apila x p) es la pila obtenida añadiedo x encima de la pila p.

```
apila :: a -> Pila a -> Pila a
apila x (P xs) = P (x:xs)
```

- (cima p) es la cima de la pila p.

```
cima :: Pila a -> a
cima (P [])   = error "cima de la pila vacia"
cima (P (x:_)) = x
```

Las pilas mediante listas

- (desapila p) es la pila obtenida suprimiendo la cima de la pila p.

```
desapila :: Pila a -> Pila a
desapila (P [])   = error "desapila la pila vacia"
desapila (P (_:xs)) = P xs
```

- (esVacia p) se verifica si p es la pila vacía.

```
esVacia :: Pila a -> Bool
esVacia (P xs) = null xs
```

- Ejemplos

```
> esVacia vacia
True
> apila 3 vacia
3|-
> desapila (((apila 4) . (apila 3)) vacia)
3
```

Especificación de las propiedades de pilas

- a cima de la pila que resulta de añadir x a la pila p es x.

```
prop_cima_apila :: Int -> Pila Int -> Bool
prop_cima_apila x p = cima (apila x p) == x
```

- La pila que resulta de añadir un elemento en un pila cualquiera no es vacía.

```
prop_apila_no_es_vacia :: Int -> Pila Int -> Bool
prop_apila_no_es_vacia x p =
not (esVacia (apila x p))
```

Especificación de las propiedades de pilas

- Ahora si comprobamos con quickCheck:

```
> quickCheck prop_cima_apila
    - No instance for (Arbitrary (Pila Int))
        arising from a use of 'quickCheck'
    - In the expression: quickCheck prop_cima_apila
      In an equation for 'it': it = quickCheck prop_cima_apila
```

- Este error nos dice que Pila debe ser una instancia de Arbitrary.
- Recordamos que quickCheck genera valores **aleatorios** para los parámetros de la función.

Generador de pilas

- Vamos a necesitar generar pilas aleatorias.

```
> sample genPila
-
0|0|-
-
-6|4|-3|3|0|-
9|5|-1|-3|0|-8|-5|-7|2|-
```

- Esto va en contra de la no mutabilidad:

```
genPila :: (Arbitrary a, Num a) => Gen (Pila a)
genPila = do xs <- listOf arbitrary
             return (foldr apila vacia xs)
```

- El tipo pila es una instancia del arbitrario:

```
instance (Arbitrary a, Num a) => Arbitrary (Pila a) where
    arbitrary = genPila
```

- Ahora si podemos comprobar:

```
> quickCheck prop_cima_apila
+++ OK, passed 100 tests.
```

Descripción informal de las colas

- Una **cola** es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción se realiza por un extremo (el posterior o final) y la operación de extracción por el otro (el anterior o frente).
- Las colas también se llaman estructuras FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir.
- Analogía con las colas del cine.

Signatura del TAD de las pilas

- Signatura:

```
vacia :: Cola a
inserta :: a -> Cola a -> Cola a
primero :: Cola a -> a
resto :: Cola a -> Cola a
desapila :: Cola a -> Cola a
esVacia :: Cola a -> Bool
```

- Descripción:
  - vacia es la cola vacía.
  - (inserta x c) es la cola obtenida añadiendo x al final de c.
  - (primero c) es la primero de la cola c.
  - (resto c) es la cola obtenida eliminando el primero de c.
  - (esVacia c) se verifica si p es la pila vacía.

Las colas mediante listas

- Cabecera del módulo:

```
module ColaL
    (Cola,
     vacia,   -- Cola a
     inserta, -- a -> Cola a -> Cola a
     primero, -- Cola a -> a
     resto,   -- Cola a -> Cola a
     esVacia, -- Cola a -> Bool
    ) where
```

- Tipo de dato algebraico de las pilas.

```
data Cola a = C [a]
    deriving (Show, Eq)
```

Las colas mediante listas

- vacia es la cola vacía.

```
vacia :: Cola a
vacia = C []
```

- (inserta x c) es la cola obtenida aÃ±adiendo x al final de la cola c.

```
inserta :: a -> Cola a -> Cola a
inserta x (C c) = C (c ++ [x])
```

- (primero c) es el primer elemento de la cola c.

```
primero :: Cola a -> a
primero (C (x:_)) = x
primero (C [])  = error "primero: cola vacia"
```

Las colas mediante listas

- (resto c) es la cola obtenida eliminando el primer elemento de la cola c.

```
resto :: Cola a -> Cola a
resto (C (_:xs)) = C xs
resto (C [])     = error "resto: cola vacia"
```

- (esVacia p) se verifica si c es la cola vacía.

```
esVacia :: Cola a -> Bool
esVacia (C xs) = null xs
```

- Ejemplos

```
> c1 = foldr inserta vacia [1..10]
> c1
C [10,9,8,7,6,5,4,3,2,1]
> primero c1
10
> resto c1
[9,8,7,6,5,4,3,2,1]
```

Bibliografía

Aprende Haskell por el bien de todos.
http://aprendehaskell.es/content/Modulos.html.
Capítulo 7: Módulos