Algoritmos y Estructuras de Datos

Guía Práctica x
Diseño de un TAD

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

En esta parte de la materia vamos a desarrollar implementaciones de TADs. El primer paso es realizar el *diseño del TAD*. Al diseñar un TAD vamos a elegir una estructura concreta para almacenar la información y vamos a escribir los algoritmos correspondientes a todas las operaciones, de manera de respetar la especificación. Un mismo TAD puede tener múltiples implementaciones. Todas ellas deberían ser "intercambiables", ya que todas ellas tienen las mismas operaciones y respetan la misma especificación.

Veamos un ejemplo simplificado de diseño del TAD conjunto acotado. Vamos a implementarlo con un arreglo de tamaño fijo (que se inicialice con la cantidad máxima de elementos que puede tener) y un entero que indique cuántos elementos hay efectivamente guardados:

```
Modulo ConjArr<T> implementa ConjuntoAcotado<T> {
      arr: Array<T>
      largo: int

      pred InvRep(c': ConjArr<T>)
         {···}

      pred Abs(c': ConjArr<T>, c: ConjuntoAcotado<T>)
         {···}

      proc nuevoConjArr(tamMax: int): ConjArr<T>
         {... pseudocódigo ...}

      proc agregar(inout c: ConjArr<T>, in e: T)
         {... pseudocódigo ...}
}
```

Analicemos cada parte:

1. Nombre

```
Modulo ConjArr<T> implementa ConjuntoAcotado<T> {
...
}
```

La primera línea contiene la palabra `Modulo` seguida del nombre de la implementación del TAD. Al igual que los TADs puede tener parámetros de tipo. Luego de la palabra `implementa` se indica el nombre del TAD que este módulo implementa. Luego del nombre tenemos la definición del módulo entre corchetes.

2. Estructura / variables de estado

```
arr: Array<T>
largo: int
```

Las variables de estado van a representar la *estructura* del módulo. Almacenarán información que van a permitir guardar el estado del módulo. Estas variables de estado serán manipuladas por las operaciones mediante el código de los algoritmos.

Para definirlas, se utilizan tipos de *implementación*, que son los tipos que usamos en nuestro pseudocódigo:

- `int, real, bool, char, string`

- `tupla<T1, T2, T3>`, `struct<campo1: val1, campo2: val2>`
- `Array<T>` (arrays de tamaño fijo)

También es posible usar otros módulos como tipo de una variable de estado. Por ejemplo, luego de definir nuestro módulo `ConjArr`, vamos a poder definir un módulo que lo use, de la siguiente manera:

```
Modulo EscuelaImpl implementa Escuela {
    alumnos: ConjArr<Alumno>
    profesores: ConjArr<Profesor>
}
```

IMPORTANTE: No es posible usar tipos de especificación como `conj<T>` o `seq<T>` (aquellos que utilizabamos para definir los observadores de un TAD).

3. Invariante de representación

El invariante de representación define una restricción sobre el conjunto de valores que pueden tomar las variables de estado para que se considere una instancia válida. Es un predicado sobre el módulo y se debe cumplir siempre al entrar y al salir de todos los procedimientos. Se puede considerar como un `requiere` y `asegura` implícito para todos los procedimientos. En nuestro ejemplo de `ConjArr`, el invariante de representación podría ser:

`pred InvRep(`c'`: ConjArr<T>)`
$\{0 \leq c'.largo \leq c'.arr.length\}$

El invariante de representación hace referencia a las variables de estado del módulo. En el caso de que el tipo de éstas sean otros módulos, hará referencia a los observadores del TAD que dicho módulo implementa. También podrá utilizar predicados y funciones auxiliares definidas en ese TAD, aunque no sus operaciones.

En nuestro ejemoplo de la escuela, la variable `alumnos` es de tipo `ConjArr<Alumno>`. Este módulo implementa el TAD `ConjuntoAcotado`, y sus observadores son *elems*, de tipo `conj<T>`, y *cap*, de tipo $\mathbb{Z}$. Por lo tanto, en el invariante de representación, para decir cosas sobre `alumnos` vamos a hablar de *elems*. Si quisieramos decir que no hay más de un docente cada 10 alumnos, podríamos escribir:

`pred InvRep(`e'`: EscuelaImpl<T>)`
$\{10.|e.docentes.elems| \leq |e.alumnos.elems|\}$

4. Función de Abstracción

La función de abstracción es una función que indica con qué instancia del TAD se corresponde una instancia del módulo. Nos va a servir para poder verificar que nuestro módulo cumple con la especificación del TAD.

La escribiremos en general como un predicado que toma una instancia del módulo y una instancia del TAD y es verdadero cuando la instancia del módulo se corresponde con la instancia del TAD. Contiene generalmente asociaciones entre las variables de estado del módulo y los observadores del TAD. Al igual que en el invariante de representación, tendremos que hacer referencia a observadores y otros predicados, no procs.

En el ejemplo de conjunto, se podría escribir de la siguiente manera:

`pred Abs(`c'`: ConjArr<T>,` c`: ConjuntoAcotado<T>)`
$\{c'.arr.length = c.cap \land (\forall e : T)(e \in c.elems \leftrightarrow e \in subseq(c'.arr, 0, c'.largo))\}$

En palabras, la instancia c' del módulo `ConjArr` se corresponde con la instancia c del TAD `ConjuntoAcotado` sí y sólo sí todos los elementos que están en el TAD están en *arr* entre las posiciones 0 y *largo*.

Algunas veces vamos a escribir la función de abstracción efectivamente como una función, con la siguiente notación:

$$\texttt{Abs}(c'\texttt{: ConjArr<T>}) = \{c\ \texttt{:ConjuntoAcotado<T>} | c'.arr.length = c.cap \land (\forall e : T)(e \in c.elems \leftrightarrow e \in subseq(c'.arr, 0, c'.largo))\}$$

Esta expresión se lee de la siguiente manera: la función *Abs* devuelve un conjunto *c tal que* se cumple el predicado que está a la derecha.

5. Operaciones

Un módulo debe implementar todos los procedimientos definidos en el TAD. Vamos a escribir los algoritmos correspondientes a cada operación de la siguiente manera:

```
proc pertenece(in c: ConjArr<T>, in e: T): bool

{
    var i: int
    i := 0
    while i < c.largo do
        if c.arr[i] == e then
            return true
        endif
        i := i + 1
    end
    return false
}
```

No vamos a ser estrictos respecto de la sintaxis del pseudocódigo: sólo nos interesa que se entienda y que contenga las operaciones de nuestro lenguaje de implementación y no más (ver Anexo I).

Si tenemos variables de estado que refieren a otros módulos, en nuestros algoritmos utilizaremos las operaciones propias del TAD para manipular la variable de estado. En nuestro caso de la escuela, como la variable alumnos es de tipo `ConjArr` que implementa el TAD `ConjuntoAcotado`, podemos usar `agregar`, `sacar`, etc, sobre esa variable:

```
proc agregarAlumno(inout e: EscuelaImpl, in a: Alumno)

{
    e.alumnos.agregar(a)
}
```

6. Anexo I: Operaciones de pseudocódigo

Para escribir algoritmos utilizaremos pseudocódigo. Seremos flexibles en cuanto a la notación, siempre y cuando sea clara la semántica. Lo que sí vamos a restringir es el conjunto de operaciones que podemos usar a la siguiente lista:

- declaración de variables

  ```
  var x: int
  var c: ConjuntoArr<int>
  ```

  Al igual que las variables de estado, el tipo de las variables internas pueden ser cualquier tipo de implementación o módulo que implemente un TAD.

- asignación

```
x := valor
```

- condicional

```
if condición then
    ... código ...
else
    ... código ...
endif
```

- ciclo

```
while condición do
    ... código ...
endwhile
```

- llamada a proc de un módulo
  - sin resultado

```
var c: ConjArr<int>
var i: int
c.agregar(i)
```

  - con resultado

```
var c: ConjArr<int>
var b: bool
b := c.vacio()
```

7. Anexo II: Memoria dinámica

Los tipos complejos son usados siempre por referencia. Es decir que existe el valor "indefinido", identificado con la palabra `null`. Si a una variable no se le asigna ningún valor, se considera que tiene el valor `null`. Intentar acceder al contenido de una variable indefinida es un error del programa.

Las variables de tipos complejos deben ser inicializadas mediante el operador `new`.

```
var a: Array<int>
var b: int

b := a[0]                  <-- error de programa

if a == null then          <-- esto está bien
    ... código ...
endif

a := new Array<int>(10)    <-- pido memoria
b := a[0]                  <-- ok. b vale 0
```

Las variables cuyo tipo es un módulo se consideran también por referencia, y deberán ser inicializadas utilizando el operador `new` seguido de una llamada al procedimiento de creación de instancias.

```
    var c: ConjArr<int>         <-- implementación de conjunto
    var b: bool

    b := c.vacío()              <-- error de programa

    c := new nuevoConj(10)      <-- inicializo c

    c.agregar(100)              <-- ok


Modulo ConjArr<T> implementa ConjuntoAcotado<T> {
    var arr: Array<T>

    proc nuevoConj(in tam: int): ConjArr<T> {
        res.arr := new Array<T>(tam)    <-- pido memoria para el array
        res.largo := 0
        return res
    }
}
```

Al ser por referencia, si asignamos una variable a otra, ambas van a apuntar a la misma instancia.

```
    var c1, c2: ConjArr<int>
    var b: bool
    var i: int

    c1 := new nuevoConj(10)
    c2 := c1            <-- ahora c2 y c1 apuntan a la misma instancia

    c1.agregar(100)     <-- ok
    i := c2.tamaño()    <-- i vale 1
```