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

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 de TAD:

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

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

    FuncAbs(c': ConjArr<T>): Conjunto<T> {
        ...
    }

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

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

Analicemos cada parte:

1. Nombre

```
    Modulo ConjArr<T> implementa Conjunto<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 (ver Anexo I).

- `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 TADs como tipo de una variable de estado. A dicha variable se le podrá asignar una instancia de cualquier módulo que implemente el TAD y se podrán utilizar cualquiera de las operaciones del TAD. No es posible usar tipos de especificación como `conj<T>` o `seq<T>`.

```
TAD Tablero {
    ...
    proc poner(inout t: Tablero, in x, y: int, in e: int)
}

Tad Juego {
    ...
}

Modulo TableroMatriz implementa Tablero {
    proc NuevoTablero(): TableroMatriz {
        ...
    }

    proc poner(inout t: TableroMatriz, in x, y: int, in e: int) {
        ...
    }
}

Modulo JuegoImpl implementa Juego {
    t: Tablero

    proc NuevoJuego(): JuegoImpl {
        res.t := TableroMatriz.NuevoTablero()
        res.t.poner(1, 1, 1)
    }
}
```

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 (aunque no necesariamente en el medio). Se puede considerar como un `requiere` y `asegura` implícito para todos los procedimientos.

```
InvRep(c': ConjArr<T>) {
    0 <= c'.largo <= 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 TADs, hará referencia a sus observadores. También podrá hacer referencia a predicados y funciones auxiliares definidas en el TAD, aunque no a sus operaciones.

```
TAD Tablero {
    obs filas: int
    obs columnas: int
    ...
}

Tad Juego {
    ...
}

Modulo JuegoImpl implementa Juego {
    t: Tablero

    InvRep(j: JuegoImpl) {
        1 <= j.t.filas <= 10 && 1 <= j.t.columnas <= 10
    }
}
```

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.

Contiene generalmente asociaciones entre las variables de estado del módulo y los observadores del TAD. Al igual que en el invariante de representación, como estamos escribiendo predicados y funciones auxiliares (no código), tendremos que hacer referencia a observadores y otros predicados, no procs.

```
FuncAbs(c': ConjArr<T>): Conjunto<T> {
    c: Conjunto | forall e: T :: e in c.elems <==> e in c'.arr[0..c'.largo]
}
```

Esto se debe leer como que la función `FuncAbs` devuelve un valor `c` que cumple con el predicado que está después de la barra vertical (|).

Muchas veces es conveniente escribir la función de abstracción como un predicado. Vamos a utilizar ambos de manera indistinta:

```
PredAbs(c': ConjArr<T>, c: Conjunto<T>) {
    forall e: T :: e in c.elems <==> e in c'.arr[0..c'.largo)
}
```

5. Operaciones

```
proc agregar(inout c: ConjArr<T>, in e: T)
    complejidad O(1)
    requiere c.largo < c.arr.Length
{
    c.arr[c.largo] := e
    c.largo := c.largo + 1
}
```

En los módulos, vamos a escribir los algoritmos correspondientes a cada operación. Un módulo debe implementar todos los procedimientos definidos en el TAD. No vamos a ser estrictos respecto de la sintaxis del pseudocódigo: sólo nos interesa que se entienda y que contenga las operaciones básicas de nuestro lenguaje de implementación (ver anexo II)

- declaración de variables
- asignación
- condicionales (`if`)
- ciclos (`while`)
- llamadas a procedimientos y a operaciones de otros módulos

Por otra parte, recordemos que los tipos que podemos usar para las variables son los de implementación.

En cuanto a los requiere y asegura, estos son opcionales. Cumplen una función diferente a la que cumplían en los TADs. Aquí sirven más que nada para encuadrar y ayudar a pensar el algoritmo y no para describir qué hace la operación (para esto tenemos los requiere y asegura del TAD). Recuerde que siempre que escribe un predicado (requiere o asegura) tiene que hacer referencia a la estructura y hablar de ella a partir de los observadores y no de las operaciones.

Por último, en la definición de los procedimientos debemos indicar la complejidad del mismo. Esta deberá estar en función de los parámetros de entrada, y también puede estar en función de los elementos de la estructura.

6. Anexo I: Tipos de implementación

Resumimos aquí los tipos de datos que podremos usar para implementar (en las variables de estado y en el código de los procs). Además de estos tipos, se puede usar cualquier módulo que haya sido dado por la cátedra o implementado por ustedes, con sus correspondientes operaciones.

6.1. Tipos básicos

`bool`: valor booleano.

Operación	Código
constantes	`true, false`
operaciones	`&&, ||, !`

Nota: el tipo bool de implementación no es trivaluado. La evaluación de una expresión indefinida se considera un error del programa. Los operadores son siempre operadores luego.

`int`: número entero.

Operación	Código
constantes	`1, 2, ...`
operaciones	`+, -, *, %, ...`
comparaciones	`<, >, <=, >=, ==, !=`
asignación	`x := 32`

`real` o `float`: número real.

Operación	ascii
constantes	`1.43, 22.5, ...`
operaciones	`+, -, *, sqrt, sin, ...`
comparaciones	`<, >, <=, >=, ==, !=`
asignación	`x := 24.454`

`char`: caracter.

Operación	Código
constantes	`'a', 'b', 'A', 'B'`
operaciones	`ord(c), char(c)`
comparaciones (a partir de *ord*)	`<, >, <=, >=, ==, !=`
asignación	`x := 'a'`

`string`: cadena de caracteres.

Operación	Código
crear	`c := "hello world"`
tamaño	`c.length`
ver posición	`c[5]`
asignar posición	`c[5] := 'a'`

6.2. Tipos complejos

`Array<T>`: arreglo de tamaño fijo de tipo T.

Operación	Código
crear	`new Array<T>(tamaño)`
tamaño	`s.length`
ver posición	`s[i]`
asignar posición	`s[i] := e`

`tupla<T1, ..., Tn>`: tupla de tipos $T_1, \ldots, T_n$

Operación	Código
crear	`new tupla<int,int>(1, 2)`
ver posición	`s[i]`
asignar posición	`s[0] := 35`

`struct<campo1: T1, ..., campon: Tn>`: tupla con nombres para los campos.

Operación	Código
crear	`new Punto<x:25, y:30>` (asumiento `Punto es struct<x: int, y: int>`)
ver campo	`p.x, p.y`
asignar campo	`p.x := 455`

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 operar con una variable indefinida es un error del programa. Las variables de tipos complejos deben ser inicializadas mediante el operador `new` (ver Anexo III).

7. Anexo II: 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: Conjunto<int>
```

  Al igual que las variables de estado, el tipo de las variables internas pueden ser cualquier tipo de implementación o TADs. A una variable cuyo tipo es un TAD se le puede asignar una instancia de cualquier módulo que implemente ese TAD.

- asignación

```
x := valor
c := ConjBitVector.NuevoConj<int>(10)
```

- condicional

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

- ciclo

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

- llamada a proc de un módulo
  - sin resultado

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

  - con resultado

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

8. Anexo III: Memoria dinámica

Vamos a considerar que todos los tipos complejos se manejan por referencia. Para asignarles un nuevo valor, hay que "pedir memoria" con el operador `new`. Al valor indefinido lo denominaremos `null`. Es un error de programa acceder a una variable que tiene el valor `null`.

Al inicializar un tipo complejo mediante el operador `new`, todos sus elementos se definen con el valor cero (0 para int, false para bool). De todas maneras, es deseable asignarle algún valor antes de usarlo.

```
    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 TAD se consideran también por referencia, y debern ser inicializadas con un módulo que implemente el TAD.

```
    var c: Conjunto<int>              <-- tad conjunto
    var b: bool

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

    c := ConjArr.NuevoConj<int>(10)   <-- inicializo c con el módulo ConjArr (conjunto sobre arrays)

    c.agregar(100)                    <-- ok


Modulo NuevoConjArr<T> implementa Conjunto<T> {
    var arr: Array<T>

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

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

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

    c1 := NuevoConjArr<int>(10)
    c2 := c1            <-- ahora c2 y c1 apuntan a la misma instancia

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