Algoritmos y Estructuras de Datos

Guía Práctica
Anatomía de un TAD

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

Versión 2: 26/9/2023
Los cambios con respecto a la versión anterior están marcados en rojo.

Un TAD, tipo abstracto de datos, define un conjunto de valores y las operaciones que se pueden realizar sobre ellos. Es abstracto ya que se enfoca en las operaciones (en cómo interactuar con los datos) y no necesita conocer los detalles de la representación interna o bien el cómo están implementadas sus operaciones. La especificación de un TAD indica qué hacen las operaciones y no cómo lo hacen.

Empecemos con un ejemplo sencillo:

```
TAD Punto {
    obs x: real
    obs y: real

    proc coordX(in p: Punto): real
        asegura ...

    proc coordY(in p: Punto): real
        asegura ...

    proc crearPunto(in x: real, in y: real): Punto
        asegura ...

    proc moverPunto(inout p: Punto, in dx: real, in dy: real)
        requiere ...
        asegura ...

    aux theta(px: real, py: real): real
        requiere ...
    {
        ...
    }

    pred estaEnElOrigen(in p: Punto) {
        ...
    }
}
```

Analicemos cada parte:

1. Nombre

```
    TAD Punto
```

La primera línea contiene la palabra TAD seguida del nombre del TAD. Un TAD puede ser genérico (parametrizado con un tipo), en cuyo caso la lista de parámetros de tipo siguen al nombre:

```
TAD Conjunto<T>
```

Luego del nombre tenemos la definición del TAD entre corchetes:

```
TAD Punto {
    ...
}
```

2. Observadores

```
obs x: real
obs y: real
```

Los observadores permiten describir el estado de una instancia del TAD en un determinado momento. Sirven para describir qué hacen las operaciones: esto se logra mediante predicados que indiquen el valor de los observadores antes y después de su ejecución (en los requiere y asegura).

Muy importante: los observadores utilizan el lenguaje de especificación y se utilizan en predicados. No son operaciones ni parte de la interface de un TAD. No pueden ser utilizadas en los programas. Sólo son utilizadas para explicar el TAD.

Los tipos de datos que se pueden usar son los de especifcación. En el anexo I de este apunte se describen los tipos de especificación y sus operaciones. También es posible usar otro TAD como tipo de un observador:

```
TAD Tablero {
    ...
}

Tad Juego {
    obs t: Tablero
    ...
}
```

Los observadores pueden ser también funciones que tomen parámetros:

```
obs puntajeDe(p: Persona): int
```

MUY IMPORTANTE: los observadores funcionales son funciones del lenguaje de especificación. No son operaciones del TAD (`proc`). No se definen (no hay que escribir una expresión indicando "qué hace"): sólo sirven para describir el funcionamiento de las operaciones.

3. Igualdad observacional

Para determinar si dos instancias de un TAD son iguales, alcanza con que todos sus observadores sean iguales. En este ejemplo, si dos puntos tienen las mismas coordenadas x y y, entonces son el mismo punto.

```
TAD Punto {
    obs x: real
    obs y: real
}
```

4. Operaciones

```
proc crearPunto(in x: real, in y: real): Punto
    asegura ...

proc moverPunto(inout p: Punto, in dx: real, in dy: real)
    requiere ...
    asegura ...
```

Las operaciones de un TAD se especifican mediante nuestro lenguaje de especificación. Se debe indicar el nombre del procedimiento, la lista de parámetros con su nombre, tipo e indicando si son `in`, `out` o `inout`. Opcionalmente las operaciones pueden devolver un valor.

La lista de operaciones que indiquemos en el TAD representan el *contrato* o *interface* del TAD, y será lo que luego implementemos y lo que usen los *clientes* del TAD. Los parámetros tienen que ser de tipos de implementación u otros TADs (no pueden ser `seq<T>` o `conj<T>` pero sí `Array<T>`, `int` o `Conjunto<T>`).

Por convención, salvo las funciones que crean nuevas instancias del TAD, vamos a tratar de usar como primer parámetro una variable de tipo del TAD.

Cada función debe indicar la precondición y la postcondición (`requiere` y `asegura`)

```
proc moverPunto(inout p: Punto, in dx: real, in dy: real)
    asegura p.x == old(p).x + dx
    asegura p.y == old(p).y + dy
```

Los predicados pueden escribirse en símbolos de LaTex como hasta ahora o, para simplificar, permitiremos escribirlos en *ascii* con una sintaxis simple, que está descripta en el anexo I.

Como ya indicamos, los requiere y asegura van a hacer referencia a los observadores del TAD. Usamos una notación estilo python (p.x para referirnos al observador x de la instancia p).

Para hablar del valor inicial de un parámetro de tipo inount (el que tenía al inicio de la función) usamos la función `old(p)`. Al nuevo valor lo identificamos con el nombre del mismo y al valor de retorno de la función lo referimos con la palabra `res` o `result` (ambas sólo en el asegura). Para los predicados podemos usar cualquier construcción de nuestro lenguaje de especificación (`forall, exists, sum, if/then/else, etc`).

```
proc nuevoPunto(in x: real, in y: real): Punto
    asegura res.x == x
    asegura res.y == y
```

Si tenemos parámetros de tipo TAD, para hablar de ellos tenemos que referirnos a sus observadores:

```
TAD Persona {
```

```
    obs nombre: string
    obs dirección: string
    obs edad: int
}

TAD Personal {
    proc agregarPersonas(inout pp: Personal, in p: Persona)
        requiere p.edad >= 18
        asegura ...
}
```

Lo mismo si tenemos observadores cuyo tipo es otro TAD:

```
    TAD Tablero {
        obs casilleros: seq<bool>
    }

    TAD Juego {
        obs tab: Tablero

        proc jugar(inout j: Juego, p: Jugador, cas: int)
            requiere 0 <= cas < |j.tab.casilleros| &&L j.tab.casilleros[cas] == false
            asegura ...
    }
```

Una observación final: salvo excepciones, para que la especificación sea completa, tenemos que describir cómo quedan todos los observadores al salir de la operación.

```
TAD Juego {
    obs jugadores: conj<Persona>
    obs jugadas: seq<Jugada>
    obs puntajes: dict<Persona, int>

    proc agregarJugador(input j: Juego, p: Persona)
        requiere ...
        // describo lo que cambia
        asegura j.jugadores == old(j).jugadores + {p}

        // pero también lo que queda igual
        asegura j.jugadas == old(j).jugadas
        asegura j.puntajes == old(j).puntajes
}
```

5. Funciones y predicados auxiliares

```
TAD Juego {
    ...
```

```
    pred seTermino(j: juego) {
        ...
    }

    aux ganador(j: Juego): Persona
        requiere seTermino(j)
    {
        ...
    }
}
```

Para facilitar la especificación, es posible definir predicados y funciones auxiliares, usando nuestro lenguaje de especificación. Estos predicados pueden usarse en los requiere y asegura de las operaciones.

También pueden ser usadas en las especificaciones de otros TADs que refieran a este:

```
TAD Juego {
    ...

    pred seTermino(j: juego) {
        ...
    }

    aux ganador(j: Juego): Persona
        requiere seTermino(j)
    {
        ...
    }
}

TAD Campeonato {
    obs juegos: seq<Juego>

    proc algunoTerminó(in c: Campeonato): bool
        asegura res <==> exists j: Juego :: j in c.juegos && j.seTermino()
}
```

6. Anexo I: Tipos de especificación

Resumimos aquí los tipos de datos que podremos usar para especificar (en los observadores, los predicados y los requiere/asegura). Asimismo, indicamos sus operaciones y su notacióon en *LaTeX* y *ascii*.

6.1. Tipos básicos

`bool`: valor booleano.

Operación	LaTeX	ascii
constantes	T, F	`true, false`
operaciones	$\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$	`&&, ||, !, ==>, <==>`
operaciones luego	$\wedge_L$, $\vee_L$, $\rightarrow_L$	`&&L, ||L, ==>L`

`int`: número entero.

Operación	LaTeX	ascii
constantes	1, 2, ...	`1, 2, ...`
operaciones	$+, -, ., /$ (div. entera), % (módulo), ...	`+, -, *, %, ...`
comparaciones	$<, >, \leq, \geq, =, \neq$	`<, >, <=, >=, ==, !=`

`real` o `float`: número real.

Operación	LaTeX	ascii
constantes	1, 2, ...	`1, 2, ...`
operaciones	$+, -, ., /, \sqrt{x}, \sin(x)$, ...	`+, -, *, sqrt, sin, ...`
comparaciones	$<, >, \leq, \geq, =, \neq$	`<, >, <=, >=, ==, !=`

`char`: caracter.

Operación	LaTeX	ascii
constantes	'a', 'b', 'A', 'B'	`á', 'b', Á', 'B'`
operaciones	$ord(c)$, $char(c)$	`ord(c), char(c)`
comparaciones (a partir de *ord*)	$<, >, \leq, \geq, =, \neq$	`<, >, <=, >=, ==, !=`

6.2. Funciones genéricas y operadores de primer orden

Operación	LaTeX	ascii	ejemplo
condicional	`if` P `then` f `else` g `fi` $IfThenElse(P, f, g)$	`if P then f else g fi` `ifThenElse(P, f, g)`	`if x > 0 then x else -x fi`

Operación	LaTeX	ascii	ejemplo
universal	$(\forall i : T)(P(i))$	`forall i: T :: P(i)`	`forall i: int :: i% 2 == 0 ==>(i-1) >0`
existencial	$(\exists i : T)(P(i))$	`exists i: T :: P(i)`	`exists i: int :: i + 2 == 2 * i`
sumatoria (con conjunto)	$(\sum_{P(i)})(f(i))$	`sum i: T :: P(i) :: f(i)`	`sum i: int :: esPrimo(i) :: 2 * i`
sumatoria (con rango)	$(\sum_{i=0}^{n})(f(i))$	`sum i: T :: lo <= i <hi :: f(i)`	`sum i: int :: 0 <= i <1000 :: i`

6.3. Tipos complejos

Estos tipos de datos son dados por la cátedra y los podemos utilizar a la hora de especificar procedimientos, sobre todo dentro de los TADs.

`seq<T>`: secuencia de tipo T.

Operación	LaTeX	ascii
crear	$\langle\rangle$, $\langle x, y, z\rangle$	`[], [x, y, z]`
tamaño	$\|s\|$, $length(s)$	`|s|, s.length`
pertenece	$i \in s$	`i in s`
ver posición	$s[i]$	`s[i]`
cabeza	$head(s)$	`head(s), s[0]`
cola	$tail(s)$	`tail(s), s[s.length-1]`
concatenar	$concat(s_1, s_2)$, $s_1 + s_2$	`concat(s1, s2), s1 + s2`
subsecuencia	$subseq(s, i, j)$, $s[i..j]$	`subseq(s, i, j), s[i..j]`
setear posición	$setAt(s, i, val)$	`setAt(s, i, val)`

`conj<T>`: conjunto de tipo T.

Operación	LaTeX	ascii
crear	$\{\}$, $\{x, y, z\}$	{}, {x, y, z}
tamaño	$\|c\|$, $length(c)$	|c|, c.length
pertenece	$i \in c$	i in c
union	$c_1 \cup c_2$	c1 + c2
intersección	$c_1 \cap c_2$	c1 * c2
diferencia	$c_1 - c_2$	c1 - c2

`dict<K, V>`: diccionario que asocia claves de tipo K con valores de tipo V.

Operación	LaTeX	ascii
crear	$\{\}$, $\{$"$juan$" $: 20,$ "$diego$" $: 10\}$	{}, {"juan": 20, "diego": 10}
tamaño	$\|d\|$, $length(d)$	|d|, d.length
pertenece (hay clave)	$k \in d$	k in d
valor	$d[k]$	d[k]
setear valor	$setKey(d, k, v)$	setKey(d, k, v)
eliminar valor	$delKey(d, k)$	delKey(d, k)

Al igual que $setAt$, la función $setKey(d, k, v)$ devuelve un diccionario igual al ingresado pero con el valor de la clave k seteado en v. Es decir, para toda clave k' tal que $k' \in d \vee k' = k$:

$$setKey(d, k, v)[k'] = \begin{cases} v & \texttt{si } k' = k \\ d[k'] & \texttt{si } k' \neq k \end{cases}$$

La función $delKey(d, k, v)$ elimina una clave del diccionario y deja igual todo el resto de los valores.

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

Operación	LaTeX	ascii
crear	$\langle x, y, z \rangle$	<x,y,z>
campo	s_i	s[i]

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

Operación	LaTeX	ascii
crear	$\langle x : 20, y : 10 \rangle$	<x: 20, y: 10>
campo	s_x, s_y	s.x, s.y

`string`: renombre de `seq<char>`.