Algoritmos y Estructuras de Datos

Guía Práctica
Anatomía de un TAD

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

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: ℝ
      obs y: ℝ

      proc crearPunto(in x: ℝ, in y: ℝ): Punto
            asegura {···}

      proc mover(inout p: Punto, in dx: ℝ, in dy: ℝ)
            requiere {···}
            asegura {···}

      pred igualdad(p1: Punto, p2: Punto)
         {···}

      aux theta(p: Punto): ℝ
         {···}

      pred estaEnElOrigen(p: Punto)
         {···}
}
```

Analicemos cada parte:

1. Nombre

La primera línea contiene la palabra TAD seguida del nombre del TAD. Luego del nombre tenemos la definición del TAD entre corchetes:

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

2. Observadores

```
      obs x: ℝ
      obs y: ℝ
```

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). 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.

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 observadores pueden ser también funciones que tomen parámetros. Por ejemplo, imaginemos que queremos registrar la historia de nuestro punto (por donde se fue moviendo) y nos interesa saber cuántas veces pasó por una determinada posición. Podemos tener un observador funcional de la siguiente forma:

```
obs cuantasVecesPaso(x: ℝ, y: ℝ): ℤ
```

Los observadores funcionales son funciones del lenguaje de especificación y sirven para explicar operaciones, indicando cuál es su valor al ingresar a los procs (en el requiere) y cuál al salir de los procs (en el asegura).

3. Igualdad

En general, para determinar si dos instancias de un TAD son iguales, alcanza con que todos sus observadores sean iguales. Es el caso de nuestro ejemplo: si dos puntos tienen las mismas coordenadas x y y, entonces son el mismo punto.

Pero muchas veces no es así. Por ejemplo si queremos describir un número racional a partir de dos números reales (numerador y denominador) tendremos muchas instancias que en realidad representan el mismo número. En ese caso, escribiremos un predicado especial denominado *igualdad* que tomará dos instancias del tipo y será verdadero si las dos instancias son iguales. En nuestro ejemplo de número racional, el predicado *igualdad* debería ser algo así:

```
pred igualdad(r1: Racional, r2: Racional)
  {r1.numerador/r1.denominador = r2.numerador/r2.denominador}
```

4. Operaciones

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. Los parámetros pueden ser de cualquier tipo, incluídos otros TADs.

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. 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.

```
proc crearPunto(in x: ℝ, in y: ℝ): Punto
     asegura {···}

proc mover(inout p: Punto, in dx: ℝ, in dy: ℝ)
     requiere {···}
     asegura {···}
```

Cada función debe indicar la precondición y la postcondición (`requiere` y `asegura`) 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 metavariables en el requiere, al igual que con veníamos haciendo con la especificación de procs. Como alternativa, se puede usar la expresión $old(p)$ para referirse al valor inicial de p (al ingresar a la función).

Para los predicados podemos usar cualquier construcción de nuestro lenguaje de especificación ($\forall, \exists, \sum, if/then/else, etc.$).

```
proc crearPunto(in x: ℝ, in y: ℝ): Punto
     asegura {res.x = x}
     asegura {res.y = y}

proc mover(inout p: Punto, in dx: ℝ, in dy: ℝ)
     requiere {p = P0}
     asegura {p.x = P0.x + dx}
     asegura {p.y = P0.y + dy}
```

Los observadores de tipo función se comportan de la misma manera. Por ejemplo, así deberíamos especificar el comportamiento de nuestro observador `cuantasVecesPaso` para el la operación `mover`:

```
proc mover(inout p: Punto, in dx: ℝ, in dy: ℝ)
      requiere {p = P_0}
      asegura {p.cuantasVecesPaso(p.x, p.y) = P_0.cuantasVecesPaso(p.x, p.y) + 1}
      asegura {(∀x, y : ℝ)((x ≠ p.x ∨ y ≠ p.y) → p.cuantasVecesPaso(x, y) = P_0.cuantasVecesPaso(x, y)}
```

En caso de tener parámetros de tipo TAD, para hablar de ellos en los *requiere* y *asegura* tenemos que referirnos a sus observadores y no a sus operaciones.

Una observación final: salvo excepciones, para que la especificación sea completa, tenemos que describir cómo quedan todos los observadores al salir de la operación. Por ejemplo, si tenemos un proc para mover sólo en el eje horizontal, tenemos que decir que el observador y queda igual:

```
proc moverHoriz(inout p: Punto, in dx: ℝ)
      requiere {p = P_0}
      asegura {p.x = P_0.x + dx}
      asegura {p.y = P_0.y}
```

5. Funciones y predicados auxiliares

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

```
aux theta(p: Punto): ℝ
   {arctan(p.y/p.x)}

aux rho(p: Punto): ℝ
   {√(p.x^2 + p.y^2)}

proc moverAngulo(inout p: Punto, in a: ℝ)
      requiere {p = P_0}
      asegura {p.x = rho(P_0) * cos(theta(P_0) + a)}
      asegura {p.y = rho(P_0) * sin(theta(P_0) + a)}
```

5.1. TADs paramétricos

Muchas veces vamos a querer especificar un TAD a partir de un tipo genérico. Por ejemplo, podemos definir un conjunto que guarde elementos que sean todos de un mismo tipo, cualquiera sea éste. Definiremos entonces un TAD `Conjunto<T>`, donde T representa el tipo de los elementos. Luego, al utilizar el TAD reemplazaremos el tipo T por el tipo concreto que queramos (`Conjunto<int>`, `Conjunto<Punto>`, etc.). A modo de ejemplo, vemos aquí algunas partes del `TAD Cola<T>`.

```
TAD Cola<T> {
      obs elems: seq<T>
      ...
      proc encolar(inout c: Cola<T>, in e: T)
            requiere {c = C_0}
            asegura {c.elems = C_0.elems + ⟨e⟩}
      ...
}
```

6. Anexo I: Tipos de especificación

Hemos expandido el lenguaje de especificación que veníamos usando con nuevos tipos complejos que nos facilitarán la especificación de TADs. Resumimos aquí los tipos de datos que podremos usar para especificar (en los observadores, los predicados y los requiere/asegura).

6.1. Tipos básicos

Las constantes devuelven un valor del tipo. Las operaciones operan con elementos de los tipos y retornan algun elemento de algun tipo. Las comparaciones generan fórmulas a partir de elementos de tipo.

`bool`: valor booleano.

Operación	Sintaxis
constantes	$True$, $False$
operaciones	$\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$
comparaciones	$=$, $\neq$

`int`: número entero.

Operación	Sintaxis
constantes	$1, 2, \cdots$
operaciones	$+$, $-$, ., / (div. entera), % (módulo), $\cdots$
comparaciones	$<$, $>$, $\leq$, $\geq$, $=$, $\neq$

`real` o `float`: número real.

Operación	Sintaxis
constantes	$1, 2, \cdots$
operaciones	$+$, $-$, ., /, $\sqrt{x}$, $\sin(x)$, $\cdots$
comparaciones	$<$, $>$, $\leq$, $\geq$, $=$, $\neq$

`char`: caracter.

Operación	Sintaxis
constantes	'a', 'b', 'A', 'B'
operaciones	$ord(c)$, $char(c)$
comparaciones (a partir de ord)	$<$, $>$, $\leq$, $\geq$, $=$, $\neq$

6.2. Tipos complejos

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

Operación	Sintaxis
crear	$\langle\rangle$, $\langle x, y, z\rangle$
tamaño	$\|s\|$, $length(s)$
pertenece	$i \in s$
ver posición	$s[i]$
cabeza	$head(s)$
cola	$tail(s)$
concatenar	$concat(s_1, s_2)$, $s_1 + s_2$
subsecuencia	$subseq(s, i, j)$, $s[i..j]$
setear posición	$setAt(s, i, val)$
suma	$\sum_{i=0}^{\|s\|} s[i]$
producto	$\prod_{i=0}^{\|s\|} s[i]$

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

Operación	Sintaxis
crear	$\{\}$, $\{x, y, z\}$
tamaño	$\|c\|$, $length(c)$
pertenece	$i \in c$
union	$c_1 \cup c_2$
intersección	$c_1 \cap c_2$
diferencia	$c_1 - c_2$

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

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

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	Sintaxis
crear	$\langle x, y, z \rangle$
campo	s_i

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

Operación	Sintaxis
crear	$\langle x : 20, y : 10 \rangle$
campo	s_x, s_y o $s.x, s.y$

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