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 {···}

      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. 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 como 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 = P_0}
      asegura {p.x = P_0.x + dx}
      asegura {p.y = P_0.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}
```

4. 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)}
```

4.1. 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⟩}
      ...
}
```

5. 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).

5.1. 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)	$<$, $>$, $\leq$, $\geq$, $=$, $\neq$

5.2. Tipos complejos

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

Operación	Sintaxis
secuencia por extensión	$\langle\rangle$, $\langle x, y, z\rangle$
longitud	$\|s\|$, $length(s)$, $s.length$
pertenece	$i \in s$
indexación	$s[i]$
cabeza	$head(s)$
cola	$tail(s)$
concatenación	$concat(s_1, s_2)$, $s_1 + +s_2$
subsecuencia	$subseq(s, i, j)$
cambiar elemento	$setAt(s, i, val)$

- secuencias por extensión: permite crear secuencias a partir de sus elementos
  - sintaxis: $\langle x : T, y : T, z : T, \cdots \rangle : seq\langle T \rangle$
  - ejemplos: $\langle\rangle$ (secuencia vacía). $\langle 1, 2, 25\rangle$ (secuencia de enteros con tres elementos: 1, 2 y 25)
- longitud: devuelve la cantidad de elementos de la secuencia
  - sintaxis: $length(s : seq\langle T\rangle) : \mathbb{Z}$. Alternativas: $|s|$ o o $s.length$
  - ejemplos: $|\langle\rangle| = 0$, $|\langle 1, 2, 25\rangle| = 3$
- pertenece: indica si un elemento está en la secuencia
  - sintaxis: $i : T \in s : seq\langle T\rangle$ (devuelve un valor de verdad)
  - ejemplos: $1 \in \langle\rangle$ es falso, $1 \in \langle 2, 3\rangle$ es falso, $1 \in \langle 1, 2, 3\rangle$ es verdadero
- indexación: devuelve el elemento en una determinada posición de la secuencia. El primer elemento es el 0. Se indefine si el índice es menor a cero o mayor al tamaño de la secuencia.
  - sintaxis: $seq\langle T\rangle[i : \mathbb{Z}] : T$.
  - ejemplos: $\langle 1, 2, 3\rangle[0] = 1$, $\langle 1, 2, 3\rangle[2] = 3$, $\langle 1, 2, 3\rangle[5]$ es indefinido
- cabeza: devuelve el primer elemento de la secuencia. Se indefine si la secuencia es vacía

- sintaxis: $head(s : seq\langle T\rangle) : T$
- ejemplos: $head(\langle 1, 2, 3\rangle) = 1$, $head(\langle\rangle)$ es indefinido

- cola: devuelve la secuencia sin el primer elemento. Se indefine si la secuencia es vacía
  - sintaxis: $tail(s : seq\langle T\rangle) : seq\langle T\rangle$
  - ejemplos: $tail(\langle 1, 2, 3\rangle) = \langle 2, 3\rangle$, $tail(\langle 1\rangle) = \langle\rangle$, $tail(\langle\rangle)$ es indefinido
- concatenación: concatena dos secuencias
  - sintaxis: $concat(s_1 : seq\langle T\rangle, s_2 : seq\langle T\rangle) : seq\langle T\rangle$. Alternativa: $s_1 + +s_2$
  - ejemplos: $concat(\langle 1, 2\rangle, \langle 3, 4\rangle) = \langle 1, 2, 3, 4\rangle$, $\langle\rangle +\!+ \langle 1, 2\rangle = \langle 1, 2\rangle$
- subsecuencia: devuelve una subsecuencia de la secuencia original, incluyendo el índice del principio y sin incluir el índice del final. Se indefine si los índices están fuera de rango o si el índice del final es menor al del principio. Devuelve una secuencia vacía si los índices son iguales.
  - sintaxis: $subseq(s : seq\langle T\rangle, i : \mathbb{Z}, j : \mathbb{Z}) : seq\langle T\rangle$
  - ejemplos: $subseq(\langle 1, 2, 3, 4, 5\rangle, 1, 3) = \langle 2, 3\rangle$, $subseq(\langle 1, 2, 3, 4, 5\rangle, 1, 1) = \langle\rangle$, $subseq(\langle 1, 2, 3, 4, 5\rangle, 3, 1)$ es indefinido
- cambiar elemento: devuelve una secuencia igual a la original pero con un elemento cambiado. Se indefine si el índice está fuera de rango.
  - sintaxis: $setAt(s : seq\langle T\rangle, i : \mathbb{Z}, val : T) : seq\langle T\rangle$
  - ejemplos: $setAt(\langle 1, 2, 3\rangle, 1, 5) = \langle 1, 5, 3\rangle$, $setAt(\langle 1, 2, 3\rangle, 5, 5)$ es indefinido

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

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

Operación	Sintaxis
tupla por extensión	$\langle x, y, z\rangle$
obtener un valor	s_i

- tupla por extensión: permite crear tuplas a partir de sus elementos
  - sintaxis: $\langle x : T_1, y : T_2, \cdots\rangle : tupla < T_1, T_2, \cdots >$
  - ejemplos: $\langle 1, `hola', 25{,}45\rangle$ (tupla de tipo $< \mathbb{Z}, string, \mathbb{R} >$ con los valores 1, 'hola'y 25.45
- obtener un valor: devuelve el valor en una determinada posición de la tupla. El primer elemento es el 0. Se indefine si el índice es menor a cero o mayor a la cantidad de elementos de la tupla.
  - sintaxis: $tupla < T_1, \ldots, T_n >_{i:\mathbb{Z}}: T_i$
  - ejemplos: $\langle 1, `hola', 25{,}45\rangle_0 = 1$, $\langle 1, `hola', 25{,}45\rangle_2 = 25{,}45$, $\langle 1, `hola', 25{,}45\rangle_5$ es indefinido

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

Operación	Sintaxis
struct por extensión	$\langle x : 20, y : 10\rangle$
obtener el valor de un campo	$s.x, s.y$

- struct por extensión: permite crear structs a partir de sus campos
  - sintaxis: $\langle campo_1 : T_1, campo_2 : T_2, \cdots\rangle : struct < campo_1 : T_1, campo_2 : T_2, \cdots >$
  - ejemplos: $\langle x : 20, y : 10\rangle$ (struct con dos campos de tipo $\mathbb{Z}$ denominados x e y con los valores 20 y 10 respectivamente
- obtener el valor de un campo: devuelve el valor de un campo de la struct. Se indefine si el campo no existe.

- sintaxis: $struct < campo_1 : T_1, \ldots, campo_n : T_n > .campo_i : T_i$
- ejemplos: $\langle x : 20, y : 10 \rangle.x = 20$, $\langle x : 20, y : 10 \rangle.z$ es indefinido

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

Operación	Sintaxis
conjunto por extensión	$\{\}$, $\{x, y, z\}$
tamaño	$\|c\|$
pertenece	$i \in c$
union	$c_1 \cup c_2$
intersección	$c_1 \cap c_2$
diferencia	$c_1 - c_2$

- conjunto por extensión: permite crear conjuntos a partir de sus elementos
  - sintaxis: $\{x : T, y : T, z : T, \cdots\} : conj\langle T \rangle$
  - ejemplos: $\{\}$ (conjunto vacío). $\{1, 2, 25\}$ (conjunto de enteros con tres elementos: 1, 2 y 25)
- tamaño: devuelve la cantidad de elementos del conjunto
  - sintaxis: $|c : conj\langle T \rangle|$
  - ejemplos: $|\{\}| = 0$, $|\{1, 2, 25\}| = 3$
- pertenece: indica si un elemento está en el conjunto
  - sintaxis: $i : T \in c : conj\langle T \rangle$ (devuelve un valor de verdad)
  - ejemplos: $1 \in \{\}$ es falso, $1 \in \{2, 3\}$ es falso, $1 \in \{1, 2, 3\}$ es verdadero
- unión de conjuntos
  - sintaxis: $c1 : conj\langle T \rangle \cup c2 : conj\langle T \rangle : conj\langle T \rangle$
  - ejemplos: $\{1, 2, 3\} \cup \{1, 4, 5\} = \{1, 2, 3, 4, 5\}$, $\{1, 2, 3\} \cup \{\} = \{1, 2, 3\}$
- intersección de conjuntos
  - sintaxis: $c1 : conj\langle T \rangle \cap c2 : conj\langle T \rangle : conj\langle T \rangle$
  - ejemplos: $\{1, 2, 3\} \cap \{1, 4, 5\} = \{1\}$, $\{1, 2, 3\} \cap \{\} = \{\}$
- diferencia de conjuntos
  - sintaxis: $c1 : conj\langle T \rangle - c2 : conj\langle T \rangle : conj\langle T \rangle$
  - ejemplos: $\{1, 2, 3\} - \{1, 4, 5\} = \{2, 3\}$, $\{1, 2, 3\} - \{\} = \{1, 2, 3\}$

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

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

- diccionario por extensión: permite crear diccionaros a partir de sus elementos
  - sintaxis: $\{k_1 : v_1, k_2 : v_2, \cdots\} : dict\langle K, V \rangle$
  - ejemplos: $\{$"$juan$" $: 20$, "$diego$" $: 10\}$ (diccionario de tipo K=$string$, V=$\mathbb{Z}$)
- tamaño: devuelve la cantidad de elementos del diccionario

- sintaxis: $|c : dict\langle K, V\rangle|$
- ejemplos: $|\{\}| = 0$, $|\{\text{“}juan\text{”} : 20, \text{“}diego\text{”} : 10\}| = 2$

- pertenece: indica si una clave está en el diccionario
  - sintaxis: $k : K \in d : dict\langle K, V\rangle$ (devuelve un valor de verdad)
  - ejemplos: $\text{“}juan'' \in \{\}$ es falso, $\text{“}juan'' \in \{\text{“}juan\text{”} : 20, \text{“}diego\text{”} : 10\}$ es verdadero
- valor: devuelve el valor asociado con una clave. Se indefine si la clave no está en el diccionario
  - sintaxis: $d[k : K] : V$
  - ejemplos: si d = $\{\text{“}juan\text{”} : 20, \text{“}diego\text{”} : 10\}$:d[“juan”] = 20,d[“ana”] está indefinido
- setKey: 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 $k' \in d \vee k' = k$:

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

  - sintaxis: $setKey(d : dict\langle K, V\rangle, k : K, v : V) : dict\langle K, V\rangle$
  - ejemplos: si d = $\{\text{“}juan\text{”} : 20, \text{“}diego\text{”} : 10\}$: $setKey(d, \text{“}juan'', 5) = \{\text{“}juan\text{”} : 5, \text{“}diego\text{”} : 10\}$
- delKey: La función $delKey(d, k, v)$ elimina una clave del diccionario y deja igual todo el resto de los valores.
  - sintaxis: $delKey[d : dict\langle K, V\rangle, k : K) : dict\langle K, V\rangle$
  - ejemplos: si d = $\{\text{“}juan\text{”} : 20, \text{“}diego\text{”} : 10\}$: $delKey(d, \text{“}juan'') = \{\text{“}diego\text{”} : 10\}$