Algoritmos y Estructuras de Datos

Apunte
Anatomía de un TAD
Primer Cuatrimestre 2025

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

1. Especificación de un TAD

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:

1.1. Nombre

La primera línea contiene la palabra TAD seguida del nombre del TAD, que debería sea declarativo, es decir, describir el problema a resolver. Luego del nombre tenemos la definición del TAD entre llaves:

```
TAD Punto {
   ···
}
```

1.2. Observadores

```
obs x: ℝ
obs y: ℝ
```

Los observadores permiten describir el estado de una instancia del TAD en un determinado momento. La especificación de las operaciones del TAD se escribe en función de los observadores antes y después de su ejecución. Es decir, los *requiere* y *asegura* deben especificar el estado de todos los observadores. Los tipos de datos que se pueden usar son los de especificación descriptos en el apunte del tema, más dos nuevos tipos complejos $\mathsf{conj}\langle T\rangle$ y $\mathsf{dict}\langle K, V\rangle$. Todos estos tipos y sus operaciones están descriptos en el Anexo I de este apunte.

Importante: Los nombres que elegimos al definir un TAD, como siempre, deben ser declarativos, es decir, deben ayudar a entender qué representa y qué hace un TAD. Es importante no sólo elegir bien los nombres de los TADs y de los observadores sino también usar renombres de tipos y, preferentemente, usar `structs` con nombres declarativos en lugar de `tuplas`. Todo esto está también descripto en el Anexo I de este apunte

1.3. 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 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 *interfaz* 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. Para referirnos al observador x de la instancia t usamos la notación $t.x$. Para hablar del valor inicial de un parámetro de tipo inout usamos metavariables, refiriendo al tipo entero, no al observador, es decir, $T_0.x$, no $t.X_0$. Las metavariables deben declararse en el `requiere`.

```
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}
}
```

Importante: Para que la especificación sea completa, tenemos que describir el estado de todos los observadores al salir de la operación. Recordemos que la implementación final del TAD sólo debe respetar la especificación y puede modificar cualquier cosas que no se defina. Por ejemplo, si tenemos un proc para mover sólo en el eje horizontal x, 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}
}
```

1.4. 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.x² + p.y²)

proc moverÁngulo (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)}
}
```

2. TADs paramétricos

Muchas veces vamos a querer especificar un TAD que tome algún de un tipo genérico como parámetro. Por ejemplo, podemos definir un conjunto que guarde elementos que sean todos de un mismo tipo, cualquiera sea éste. Definiremos entonces un TAD Conjunto$\langle T \rangle$, donde T representa el tipo de los elementos. Luego, al utilizar el TAD reemplazaremos el tipo T por el tipo concreto que queramos (Conjunto$\langle \mathbb{Z} \rangle$, Conjunto$\langle Punto \rangle$, etc.). A modo de ejemplo, vemos aquí algunas partes del TAD Cola$\langle T \rangle$.

```
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⟩}
    }
    ...
}
```

3. 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).

Para mayor declaratividad en los TADs recomendamos hacer *renombres de tipos básicos*. Por ejemplo, si queremos representar alumnos de una escuela por su número de documento o algún otro identificador numérico podemos escribir.

```
Alumno ES ℤ
```

y luego usarlos como cualquier tipo

```
proc calificarAlumno (inout e : Escuela, in a : Alumno, in nota : ℤ) {
   ...
}
```

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

3.2. Tipos complejos

`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

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

Operación	Sintaxis
secuencia por extensión	$\langle\rangle$, $\langle x, y, z\rangle$
longitud	$\lvert s\rvert$, $length(s)$
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: $\langle x : T, y : T, z : T, \cdots\rangle : \mathsf{seq}\langle T\rangle$
  - 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: $|s : \mathsf{seq}\langle T\rangle| : \mathbb{Z}$. Alternativa: $length(s)$ o $longitud(s)$
- ejemplos: $|\langle\rangle| = 0$, $|\langle 1, 2, 25\rangle| = 3$

- pertenece: indica si un elemento está en la secuencia
  - sintaxis: $i : T \in s : \mathsf{seq}\langle T\rangle$ (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: $\mathsf{seq}\langle T\rangle[i : \mathbb{Z}] : T$.
  - 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: $head(s : \mathsf{seq}\langle T\rangle) : T$
  - 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: $tail(s : \mathsf{seq}\langle T\rangle) : \mathsf{seq}\langle T\rangle$
  - 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: $concat(s_1 : \mathsf{seq}\langle T\rangle, s_2 : \mathsf{seq}\langle T\rangle) : \mathsf{seq}\langle T\rangle$. Alternativa: $s_1 \mathbin{+\!\!+} s_2$
  - ejemplos: $concat(\langle 1, 2\rangle, \langle 3, 4\rangle) = \langle 1, 2, 3, 4\rangle$, $\langle\rangle \mathbin{+\!\!+} \langle 1, 2\rangle = \langle 1, 2\rangle$
- 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: $subseq(s : \mathsf{seq}\langle T\rangle, i : \mathbb{Z}, j : \mathbb{Z}) : \mathsf{seq}\langle T\rangle$
  - 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: $setAt(s : \mathsf{seq}\langle T\rangle, i : \mathbb{Z}, val : T) : \mathsf{seq}\langle T\rangle$
  - 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>`.

`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: $\{x : T, y : T, z : T, \cdots\} : \mathsf{conj}\langle T\rangle$
  - 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: $|c : \mathsf{conj}\langle T\rangle| : \mathbb{Z}$. Alternativa: $size(s)$ o $tamaño(s)$
- ejemplos: $|\{\}| = 0$, $|\{1, 2, 25\}| = 3$

- pertenece: indica si un elemento está en el conjunto
  - sintaxis: $i : T \in c : \mathsf{conj}\langle T\rangle$ (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: $c1 : \mathsf{conj}\langle T\rangle \cup c2 : \mathsf{conj}\langle T\rangle : \mathsf{conj}\langle T\rangle$
  - ejemplos: $\{1, 2, 3\} \cup \{1, 4, 5\} = \{1, 2, 3, 4, 5\}$, $\{1, 2, 3\} \cup \{\} = \{1, 2, 3\}$
- intersección de conjuntos
  - sintaxis: $c1 : \mathsf{conj}\langle T\rangle \cap c2 : \mathsf{conj}\langle T\rangle : \mathsf{conj}\langle T\rangle$
  - ejemplos: $\{1, 2, 3\} \cap \{1, 4, 5\} = \{1\}$, $\{1, 2, 3\} \cap \{\} = \{\}$
- diferencia de conjuntos
  - sintaxis: $c1 : \mathsf{conj}\langle T\rangle - c2 : \mathsf{conj}\langle T\rangle : \mathsf{conj}\langle T\rangle$
  - 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	$d[k]$
claves	$claves(d)$
setear valor	$setKey(d, k, v)$
eliminar valor	$delKey(d, k)$

- diccionario por extensión: permite crear diccionaros a partir de sus elementos
  - sintaxis: $\{k_1 : v_1, k_2 : v_2, \cdots\} : \mathsf{dict}\langle K, V\rangle$
  - ejemplos: $\{$"$juan$" : 20, "$diego$" : 10$\}$ (diccionario de tipo K=$string$, V=$\mathbb{Z}$)
- tamaño: devuelve la cantidad de elementos (claves) del diccionario
  - sintaxis: $|c : \mathsf{dict}\langle K, V\rangle| : \mathbb{Z}$. Alternativa: $size(s)$ o $tamaño(s)$
  - ejemplos: $|\{\}| = 0$, $|\{$"$juan$" : 20, "$diego$" : 10$\}| = 2$
- pertenece: indica si una clave está en el diccionario
  - sintaxis: $k : K \in d : \mathsf{dict}\langle K, V\rangle$ (devuelve un valor de verdad)
  - ejemplos: "$juan$" $\in \{\}$ es falso, "$juan$" $\in \{$"$juan$" : 20, "$diego$" : 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 = $\{$"$juan$" : 20, "$diego$" : 10$\}$: d["juan"] = 20, d["ana"] está indefinido
- claves: devuelve el conjunto de claves de un diccionario
  - sintaxis: $claves(d : TDiccKV) : \mathbb{Z}$
  - ejemplos: si d = $\{$"$juan$" : 20, "$diego$" : 10$\}$: claves(d) = $\{$"$juan$", "$diego$"$\}$

- 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: $setKey(d : \mathsf{dict}\langle K, V\rangle, k : K, v : V) : \mathsf{dict}\langle K, V\rangle$
  - ejemplos: si d = {"$juan$" : 20, "$diego$" : 10}: $setKey(d, \text{"}juan\text{"}, 5) = \{\text{"}juan\text{"} : 5, \text{"}diego\text{"} : 10\}$

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

  - sintaxis: $delKey[d : \mathsf{dict}\langle K, V\rangle, k : K) : \mathsf{dict}\langle K, V\rangle$
  - ejemplos: si d = {"$juan$" : 20, "$diego$" : 10}: $delKey(d, \text{"}juan\text{"}) = \{\text{"}diego\text{"} : 10\}$

3.2.1. Sumatorias y productorias sobre conjuntos y diccionarios

Se puede escribir sumatorias y productorias sobre conjuntos usando la siguiente sintáxis

$$elementosPositivos = \sum_{e \in c} \big(\mathsf{IfThenElse}(e \geq 0, 1, 0)\big)$$

Para hacer algo similar sobre un diccionario hay que usar el diccionario de claves

$$significadosPositivos = \sum_{c \in claves(d)} \big(\mathsf{IfThenElse}(d[c] \geq 0, 1, 0)\big)$$