Algoritmos y Estructuras de Datos

Segundo cuatrimestre de 2024

Departamento de Computación - FCEyN - UBA

Tipos Abstractos de Datos - wp invocación

¿Qué es un TAD?

- TAD quiere decir Tipo Abstracto de Datos
- ¿Qué es un Tipo Abstracto de Datos?
  - Es un tipo de datos porque define un conjunto de valores y las operaciones que se pueden realizar sobre ellos
  - Es abstracto ya que para utilizarlos, no se necesita conocer los detalles de la representación interna ni cómo están implementadas sus operaciones.
  - No conocemos "la forma" de los valores
  - Describe el "qué" y no el "cómo"
  - Son una forma de modularizar a nivel de los datos
- ¿Qué TADs recuerdan de IP?
  - Diccionario
  - Pila
  - Cola

¿Qué es un TAD?

Ejemplo - Pila

- Una pila es una lista de elementos de la cual se puede extraer el último elemento insertado.
- Operaciones básicas
  - **apilar:** ingresa un elemento a la pila
  - **desapilar:** saca el último elemento insertado
  - **tope:** devuelve (sin sacar) el último elemento insertado
  - **vacía:** retorna verdadero si está vacía
- Todo esto es lo que ya sabemos (de IP), pero ...
  - ¿**Qué** hacen **exactamente** estas operaciones.
  - ¿Y si tuvieramos que **demostrar** un programa que usa una Pila?
  - Vamos a tener que especificar estas operaciones.

¿Qué es un TAD?
Ejemplo - Conjunto

- El TAD conjunto es una abstracción de un conjunto matemático, que "contiene" "cosas" (todas del mismo tipo), sus "elementos".
- Hay operaciones para **agregar** y **sacar** elementos y para ver si algo está o no (**pertenece**). Se puede saber cuántos elementos tiene.
- El conjunto no tiene en cuenta repetidos: si en un conjunto de números agregamos el 1, el 5 y otra vez el 1, la cantidad de elementos será 2.

¿Qué es un TAD?
Ejemplo - Punto 2D

- El TAD punto 2D es una abstracción de un punto en el plano cartesiano.
- Se puede describir a partir de sus coordenadas cartesianas (x, y) o polares (ρ,θ).
- Tiene operaciones para moverlo, rotarlo sobre el eje o alejarlo del centro, etc

¿Qué caracteriza a un TAD?

- **Instancias:** que pertenecen a su conjunto de valores
- **Operaciones:**
  - para crear una nueva instancia
  - para calcular valores a partir de una instancia
  - para modificar
- **Observadores:**
  - El estado de una instancia de un TAD lo describimos a través de variables o funciones de estado llamadas observadores
  - Podemos usar todos los tipos de datos del lenguaje de especificación ($\mathbb{Z}$, $\mathbb{R}$, $seq < T >$, $conj < T >$, etc.)
  - En un instante de tiempo, el estado de una instancia del TAD estará dado por el estado de todos sus observadores

Tipos abstractos de datos

- ¿Qué son los TADs?
- Anatomía de un TAD
- Observadores
- Operaciones
- Ejemplos de especificaciones

Demostración de la invocación

- Ejemplo 1 - Sumatoria
- Ejemplo 2 - Desapilar en una Pila

Observadores

Ejemplo: Pila

- ¿Qué tipo básico del lenguaje de especificación nos puede ayudar para especificar las operaciones de una pila?
- Una secuencia:
  `obs s: seq<T>`
  - La pila vacía es una secuencia vacía
  - Cuando apilamos le agregamos un elemento a la secuencia (¿Cuál?)
  - Cuando desapilamos le sacamos un elemento a la secuencia (¿Cuál?)
  - Cuando querramos ver el elemento de arriba, miramos un elemento de la secuencia (¿Cuál?)

Observadores

Ejemplo: TAD punto 2D

- El estado del TAD punto 2D puede ser dado por:
  - variables de estado para las coordenadas cartesianas
    `obs x:` $\mathbb{R}$
    `obs y:` $\mathbb{R}$
  - o, variables de estado para las coordenadas polares
    `obs rho:` $\mathbb{R}$
    `obs theta:` $\mathbb{R}$
  - ¡Pero no ambas!
- ¿Podríamos tener un solo observador real (por ejemplo, una sola coordenada)?
  - No nos serviría, porque no se puede describir un punto del plano mediante una sola coordenada. No nos alcanza.

Observadores

Ejemplo: TAD conjunto

- El estado del TAD conjunto puede ser:
  - una variable de tipo $conj < T >$ (el conjunto de nuestro lenguaje de especificación)

    `obs elems: conj<T>`

    Nota: Formalmente, las variables de estado pueden considerarse también funciones como

    `obs elems(c: Conjunto<T>): conj<T>`
  - O, una función que, dado un elemento, indique si está o no está presente en el conjunto y otra que nos indique la cantidad de elementos

    `obs esta(e: T): Bool`

Observadores

- El conjunto de observadores tiene que ser completo. Tenemos que poder observar todas las características que nos interesan de las instancias.
- A partir de los observadores se tiene que poder distinguir si dos instancias son distintas
- Todas las operaciones tienen que poder ser descriptas a partir de los observadores
- OJO Si usamos funciones como observadores, estas son funciones auxiliares de nuestro lenguaje de especificación, y por lo tanto:
  - no pueden tener efectos colaterales ni modificar los parámetros
  - pueden usar tipos de nuestro lenguaje de especificación
  - NO pueden usar otros TADs

Operaciones de un TAD

- Las operaciones del TAD indican qué se puede hacer con una instancia de un TAD
- Las especificamos con nuestro lenguaje de especificación
- Para indicar qué hacen, usamos precondiciones y postcondiciones (requiere y asegura)

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

- Para eso hablaremos del estado del TAD (o sea, del valor de sus observadores) antes y después de aplicar la operación

Especifiquemos un TAD

Ejemplo - Pila

```
TAD Pila<T> {
    obs s: seq<T>

    proc pilaVacía(): Pila<T>
        asegura {res.s = ⟨⟩}

    proc vacía(in p: Pila<T>): bool
        asegura {res = true ↔ p.s = ⟨⟩}

    proc apilar(inout p: Pila<T>, in e: T)
        requiere {p = P_0}
        asegura {p.s = concat(P_0.s, ⟨e⟩)}

    proc desapilar(inout p: Pila<T>): T
        requiere {p = P_0}
        requiere {p.s ≠ ⟨⟩}
        asegura {p.s = subseq(P_0.s, 0, |P_0.s| − 1)}
        asegura {res = P_0.s[|P_0.s| − 1]}

    proc tope(in p: Pila<T>): T
        requiere {p = P_0}
        requiere {p.s ≠ ⟨⟩}
        asegura {res = P_0.s[|P_0.s| − 1]}
}
```

Tipos abstractos de datos
¿Qué son los TADs?
Anatomía de un TAD
Observadores
Operaciones
Ejemplos de especificaciones

Demostración de la invocación
Ejemplo 1 - Sumatoria
Ejemplo 2 - Desapilar en una Pila

Especifiquemos un TAD

Ejemplo - Conjunto

```
TAD Conjunto<T> {
      obs elems: conj<T>

      proc conjVacío(): Conjunto<T>
            asegura {res.elems = ⟨⟩}

      proc pertenece(in c: Conjunto<T>, in e: T): bool
            asegura {res = true ↔ e ∈ c.elems}

      proc agregar(inout c: Conjunto<T>, in e: T)
            requiere {c = C0}
            asegura {c.elems = C0.elems ∪ ⟨e⟩}

      proc sacar(inout c: Conjunto<T>, in e: T)
            requiere {c = C0}
            asegura {c.elems = C0.elems − ⟨e⟩}

      proc unir(inout c: Conjunto<T>, in c': Conjunto<T>):TAREA
      proc restar(inout c: Conjunto<T>, in c': Conjunto<T>):TAREA
      proc intersecar(inout c: Conjunto<T>, in c': Conjunto<T>):TAREA
      proc agregarRápido(inout c: Conjunto<T>, in e: T):TAREA
      proc tamaño(in c: Conjunto<T>): ℤ:TAREA
}
```


Especifiquemos un TAD

Ejemplo - Punto 2D

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

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

      proc coordX(in p: Punto): ℝ
            asegura {res = p.x}

      proc coordY(in p: Punto): ℝ
            asegura {res = p.y}

      proc coordTheta(in p: Punto): ℝ
            asegura {res = safearctan(p.x, p.y)}

      proc coordRho(in p: Punto): ℝ
            asegura {res = sqrt(p.x ** 2 + p.y ** 2)}

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

      aux safearctan(x: ℝ, y: ℝ) = ifThenElseFi(x == 0,
π/2*signo(y),arctan(y/x))
```


Tipos abstractos de datos
¿Qué son los TADs?
Anatomía de un TAD
Observadores
Operaciones
Ejemplos de especificaciones

Demostración de la invocación
Ejemplo 1 - Sumatoria
Ejemplo 2 - Desapilar en una Pila

Procedimientos y funciones: por qué?

- Reuso de código
- Razonamiento más compacto y efectivo
- Evolución (correcta) de código

Ejemplo Proc y Uso y Re-uso

Notar que el lenguaje SmallLang no tenía ni definiciones ni invocaciones a procedimientos. Agregamos la definición de procedmientos (ya vista de alguna manera) y la invocación `x := Call P (E)` al lenguaje. Lo mantenemos simple para ilustrar el concepto

```
PROC Sumatoria (in hasta:ℤ):ℤ
    s:=0;
    i:=1;
    While i ≤ hasta
        s:=s+i;
        i:=i+1
    EndWhile;
    Result:=s;
    Return
```

```
x:= Sumatoria(n);
y:= Sumatoria(m-1);
z:= x - y
```


Procedimientos y funciones: por qué?

- Reuso de código: Ok, es más o menos obvio (abstracción procedimental)
- Razonamiento más compacto/abstracto: Usar la abstracción procedimental para no pensar en cómo hace lo que hace. ¿O sea?...

Ejemplo Proc y Uso con Contratos

```
PROC Sumatoria (in hasta:ℤ):ℤ
    s:=0;
    i:=1;
    While i ≤ hasta
        s:=s+i;
        i:=i+1
    EndWhile;
    Result:=s;
    Return
```

```
{true}
x:= Sumatoria(n);
y:= Sumatoria(m-1);
z:= x - y
{z = Σ_{k=m}^{n} k }
```

¿Cómo demostramos esta tripla de hoare?

Razonamiento con Proc: Inlining

```
{true}
s:=0;
i:=1;
While )i ≤ n)
 s:=s+i;
 i:=i+1
EndWhile;
x:=s;
s:=0;
i:=1;
While (i ≤ m-1)
 s:=s+i;
 i:=i+1
EndWhile;
y:=s;
z:= x - y
{z = Σ_{k=m}^{n} k }
```

¿Qué tenemos que hacer para probar que $\{Pre\}$`S1;while...;S3`$\{Post\}$ es válida?

1. $Pre \Rightarrow_L wp(S1, P_C)$
2. $P_C \Rightarrow_L wp(while..., Q_C)$
3. $Q_C \Rightarrow_L wp(S3, Post)$

Por monotonía, esto nos permite demostrar que

$Pre \Rightarrow_L wp($`S1;while...;S3`$, Post)$ es verdadera.

Nota: Este ejemplo, con dos ciclos es muy complejo de resolver de esta forma.

Razonamiento modular basado en procedimientos

Inlining no es problemático. PERO qué pasa si sabemos que es cierta tupla de Hoare: $\{\text{Pre}\}C_{Proc}\{\text{Post}\}$ (por ejemplo porque lo dice el `requiere` y el `asegura` y lo hemos probado). Ejemplo:

```
proc Sumatoria (in hasta: ℤ):ℤ
    requiere {true}
    asegura {result = Σ_{k=1}^{hasta} k }
```

Queremos usar esa información para probar el código que invoca al procedimiento. Ejemplo:

```
{true}
x:= Sumatoria(n);
y:= Sumatoria(m-1);
z:= x - y
Q≡{z = Σ_{k=m}^{n} k }
```

Surge la pregunta: ¿Cuál es la wp(`x := Call Proc (E)`, Q)?

Wp (`x:=Call P(E)`, Q) sabiendo $\{\text{Pre}\}C_P\{\text{Post}\}$

- Qué quiero lograr?: Razonamiento Modular! O sea:
  - Reusar de alguna manera lo que sé del procedimiento y no reproducir los pasos de la prueba $\{\text{Pre}\}C_{Proc}\{\text{Post}\}$ cada vez que me encuentro con una invocación del procedimiento
  - Veamos en concreto esto del razonamiento modular con Wp

Wp (`x := Call P(E)`, Q) sabiendo {Pre} C_P {Post}

Asumamos que

- P tiene un parámetro formal pf que es in
- el resultado va a parar antes del retorno a la variable distinguida result
- Pre (del proc) predica sobre pf
- Post (del proc) sobre pf y result (i.e, Pre(pf) y Post(pf,result))
- Asumamos que además probamos que pf = pf_0 en el retorno (Wp ó analizando en el código) ya que lo pide el hecho de ser un parámetro in

Entonces:

$$wp(x := Call\ P(E),\ Q) =_{def}$$
$$def(E) \wedge_L Pre^{pf}_{E} \wedge_L (\forall\ r)(Post^{pf|res}_{E|r} \Rightarrow Q^{x}_{r}\ )$$

Nota: Pre y Post son del Proc, Q es del programa donde se usa el Proc.

Ejemplo

$$wp(x := Call\ P(E),\ Q) =_{def}$$
$$def(E) \wedge_L Pre^{pf}_{E} \wedge_L (\forall\ r)(Post^{pf|res}_{E|r} \Rightarrow Q^{x}_{r}\ )$$

{true} $\not\Rightarrow$
wp(S,Q)≡wp(S1,wp(S2,wp(S3,Q)≡
$\{(\Sigma^{n}_{k=1}\ k) - (\Sigma^{m-1}_{k=1}\ k) = \Sigma^{n}_{k=m}\ k\} \equiv \{n \geq m\}$
S1 x:= Sumatoria(n);
$wp(S2,wp(S3,Q) \equiv \{x-(\Sigma^{m-1}_{k=1}\ k) = \Sigma^{n}_{k=m}\ k\}$
S2 y:= Sumatoria(m-1);
$wp(S3,Q) \equiv \{x-y = \Sigma^{n}_{k=m}\ k\}$
S3 z:= x - y
$Q \equiv \{z = \Sigma^{n}_{k=m}\ k\}$

Nuestro programa que usa el proc Sumatoria no es correcto

Algunas Conclusiones

- Usamos el **qué** del procedimiento para probar el **cómo** del código que lo usa (código "cliente"). **Abstracción procedimental acompañada de razonamiento modular!**
- Cualquier cambio del procedimiento que deje igual o debilite su precondición y deje igual o fortalezca la postcondición NO impacta en la corrección del código "cliente" (Design by Contracts (Meyer)/ **Principio de Sustitución** de Liskov). Evolución disciplinada del software
- Lo que viemos es una pieza central en el camino hacia mecanismos que ponen -de manera abstracta- a disposición procedimientos (y estructuras de datos) que el código cliente puede invocar (ej. liberías, proc de TADs) o ser invocado (ej. framework)

Usando el TAD Pila

Recordemos

```
proc desapilar(inout p: Pila<T>): T
    requiere {p = P0}
    requiere {p.s ≠ ⟨⟩}
    asegura {p.s = subseq(P0.s, 0, |P0.s| − 1)}
    asegura {res = P0.s[|P0.s| − 1]}
```

Probemos la siguiente tupla de Hoare:

Requiere $\equiv \{mazo.s \neq \langle\rangle \wedge_L |mazo.s| + i = tamorig\}$
i:=i+1;
wp(descarte := Call desapilar(mazo), Q) = ??
descarte:= Call desapilar(mazo)
Asegura $\equiv$ Q $\equiv \{|mazo.s| + i = tamorig\}$

Wp (x := Call P(E,v), Q) sabiendo $\{\texttt{Pre(pf}^1\texttt{,pf}^2\texttt{,Pf}^2_0\texttt{)}\}C_P\{\texttt{Pos(res,pf}^1\texttt{,pf}^2\texttt{,Pf}^2_0\texttt{)}\}$

Generalizando lo que vimos para Sumatoria, asumamos que:

- Parámetros formales: $\texttt{pf}^1$ (in); $\texttt{pf}^2$ (inout)
- El resultado va a parar a la variable distinguida res
- Pre predica sobre pfs y la metavble $\texttt{Pf}^2_0$ ($\texttt{Pf}^2_0 = \texttt{pf}^2$)
- Pos predica sobre pfs, res y $\texttt{Pf}^2_0$
- $\texttt{pf}^1$ sólo se lee: no aparece en el lado izquierdo de una asignación (para simplificar)
- Para simplificar el predicado, que hay un sólo tipo

Entonces:

$$\texttt{wp(x := Call P(E,v), Q)} =_{def} \texttt{def(E)} \wedge_L$$
$$(\exists \texttt{Pf}^2_0)(\texttt{Pre[pf}^1\texttt{/E,pf}^2\texttt{/v]} \wedge_L$$
$$(\forall \texttt{r,m})(\texttt{Post[pf}^1\texttt{/E,pf}^2\texttt{/m,res/r]} \Rightarrow \texttt{Q[x/r,v/m]}))$$

Donde / es sustitución de variable libre a la izquierda por expresión a la derecha.
Nota, Q[x/r] es lo mismo que Q^x_r

Usando el TAD Pila

$$\texttt{wp(x := Call P(E,v), Q)} =_{def} \texttt{def(E)} \wedge_L$$
$$(\exists \texttt{Pf}^2_0)(\texttt{Pre[pf}^1\texttt{/E,pf}^2\texttt{/v]} \wedge_L$$
$$(\forall \texttt{r,m})(\texttt{Post[pf}^1\texttt{/E,pf}^2\texttt{/m,res/r]} \Rightarrow \texttt{Q[x/r,v/m]}))$$

wp(descarte := Call desapilar(mazo), Q) = ??
descarte:= Call desapilar(mazo)
Q $\equiv \{|mazo.s| + i = tamorig\}$

$\texttt{wp(descarte := Call desapilar(\_,mazo), Q)} =_{def}$
def$(mazo) \wedge_L (\exists P_0\text{:pila})(mazo = P_0 \wedge_L mazo.s \neq \langle\rangle \wedge_L$
$(\forall\ \text{r:carta, m:pila})(m.s = subseq(P_0.s,\ 0,\ |P_0.s| - 1) \wedge_L$
$r = P_0.s[|P_0.s| - 1]) \Rightarrow |m.s| + i = tamorig\ ) \equiv$

$(\exists P_0\text{:pila})(mazo = P_0 \wedge_L mazo.s \neq \langle\rangle \wedge_L$
$(|subseq(P_0.s,\ 0,\ |P_0.s| - 1)| + i = tamorig)) \equiv$

$mazo.s \neq \langle\rangle \wedge_L (|subseq(mazo.s,\ 0,\ |mazo.s| - 1)| + i = tamorig) \equiv$

$mazo.s \neq \langle\rangle \wedge_L (|mazo.s| - 1 + i = tamorig)$

Usando el TAD Pila

Probemos la siguiente tupla de Hoare:

Requiere $\equiv \{mazo.s \neq \langle\rangle \wedge_L |mazo.s| + i = tamorig\}$
i:=i+1;
$\{mazo.s \neq \langle\rangle \wedge_L |mazo.s| - 1 + i = tamorig\}$
descarte := Call desapilar(mazo)
Asegura $\equiv Q \equiv \{|mazo.s| + i = tamorig\}$

Requiere $\Rightarrow$ wp(i:=i+1, $\{mazo.s \neq \langle\rangle \wedge_L |mazo.s| - 1 + i = tamorig\}$)

Requiere $\Rightarrow \{mazo.s \neq \langle\rangle \wedge_L |mazo.s| + i = tamorig\}$

$\{mazo.s \neq \langle\rangle \wedge_L |mazo.s| + i = tamorig\} \Rightarrow \{mazo.s \neq \langle\rangle \wedge_L |mazo.s| + i = tamorig\}$

Q.E.D.