Introducción a diseño

Invariante de Representación y Función de Abstracción

Diego Bendersky, 27/9/2023

Introducción

TADs

- Definimos las operaciones
- Describimos qué hace cada una (con observadores y lógica)
- Dijimos que se pueden implementar de varias maneras, en función de los requerimientos

Introducción

Diseño

- Ahora queremos *implementar* un TAD. Qué tenemos que hacer?
  - Elegir una *estructura* de nuestro lenguaje de programación
  - Escribir los *algoritmos*
- Queremos que la combinación estructura/algoritmos
    - Sea eficiente
      - Se pueda ejecutar rápido
      - Ocupe poca memoria
    - Sea copada
      - Elegante, linda, fácil de entender, fácil de cambiar

Nombre del módulo

```
Modulo ConjArr<T> implementa Conjunto<T> {
```

Nomnbre del módulo y qué TAD implementa

- Puede ser genérico (como los TADs)

```
}
```

Estructura

```
Modulo ConjArr<T> implementa Conjunto<T> {
    arr: Array<T>
    largo: int
}
```

- Qué tipos de datos podemos usar?
  - `int, real, bool, char, string`
  - `tupla, struct`
  - `Array`
  - `TADs`
    - NO tipos de especificación
    - `seq, conj`

Estructura

```
Modulo ConjArr<T> implementa Conjunto<T> {
    arr: Array<T>
    largo: int
```

- Qué tipos de datos podemos usar?
  - `int, real, bool, char, string`
  - `tupla, struct`
  - `Array`
  - `TADs`

esto nos va a permitir componer y modularizar

```
}
```

Estructura (modularización)

```
Modulo ConjArr<T> implementa Conjunto<T> {
    clientes: Conjunto<Cliente>
    ventas: Diccionario<Cliente, Venta>
}
```

- En la estructura usamos los tipos *abstractos*
- Nos definen qué operaciones vamos a poder usar
- Dependen de los requerimientos del problema
    - Hay orden? Se permiten repetidos? …

Estructura (modularización)

```
Modulo StockImpl implementa Stock {
    clientes: Conjunto<Cliente>
    ventas: Diccionario<Cliente, Venta>
}
```

- Más adelante, cuando escribamos los algoritmos, vamos a decidir qué módulo usar para cada uno de ellos
    - para clientes, un Conjunto implementado con arrays porque quiero agregar fácil
    - para ventas, un Diccionario implementado con hash tables porque quiero buscar rápido

Módulos

invariante de representación

Función de abstracción

```
Modulo ConjArr<T> implementa Conjunto<T> {
    arr: Array<T>
    largo: int

    InvRep(c': ConjArr<T>) {
        ...
    }

    FuncAbs(c': ConjArr<T>): Conjunto<T> {
        ...
    }

}
```

Invariante de representación

- Es un predicado (de nuestro lenguaje de especificación)
- Toma un sólo parámetro: el módulo
- Nos va a indicar cuáles instancias son válidas y cuales no

```
Modulo ConjArr<T> implementa Conjunto<T> {
    arr: Array<T>
    largo: int

    InvRep(c': ConjArr<T>) {
        ...
    }
```

```
arr = Array<int>(10), largo=24
```


```
arr = Array<int>(10), largo=3
```


```
InvRep(c': ConjArr<T>) {

    0 <= c'.largo <= c'.arr.length

}
```

Invariante de representación

- Tiene que valer *siempre* al entrar y al salir de cualquier procedimiento
- Pero no necesariamente en el medio

Como el invariante del ciclo!

```
Modulo ConjArr<T> implementa Conjunto<T> {
    arr: Array<T>
    largo: int

    InvRep(c': ConjArr<T>) {
        ...
    }
```

Invariante de representación

- Si tenemos TADs, vamos a usar sus observadores, no sus procs

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

```
TAD Diccionario<K, V> {
    obs data: dict<K, V>
```

```
Modulo StockImpl implementa Stock {
    clientes: Conjunto<Cliente>
    ventas: Diccionario<Cliente, Venta>

    InvRep(s': StockImpl) {
      |s'.clientes.elems| > 0 &&
      forall c: Cliente :: c in s'.ventas.data ==> c in s'.clientes.elems
}
```

Función de abstracción

- Es una función (de nuestro lenguaje de especificación) un poco especial
- Toma un parámetro de tipo del módulo concreto y devuelve una instancia del TAD
- Nos permite relacionar ambas cosas: dada una instancia del módulo, a qué instancia del TAD se corresponde

```
Modulo ConjArr<T> implementa Conjunto<T> {
    arr: Array<T>
    largo: int

    FuncAbs(c': ConjArr<T>): Conjunto<T> {
        ...
    }
```

Se lee: la función devuelve un conjunto que cumple el predicado que aparece después de la barra

```
FuncAbs(c': ConjArr<T>): Conjunto<T> {
    c: Conjunto |
    c'.largo == |c.elems| &&
    forall i: 0 <= i < |c.elems| ==> c'.arr[i] == c.elems[i]
}
```

Función de abstracción

```
Modulo ConjArr<T> implementa Conjunto<T> {
    arr: Array<T>
    largo: int


    FuncAbs(c': ConjArr<T>): Conjunto<T> {
        ...
    }
```

- También lo vamos a escribir como un predicado (es lo mismo)

```
PredAbs(c': ConjArr<T>, c:
Conjunto<T>) {
    c'.largo == |c.elems| &&
    forall i: 0 <= i < |c.elems| ==>
    c'.arr[i] == c.elems[i]
}
```

```
FuncAbs(c': ConjArr<T>): Conjunto<T> {
    c: Conjunto |
    c'.largo == |c.elems| &&
    forall i: 0 <= i < |c.elems| ==>
    c'.arr[i] == c.elems[i]
}
```

Función de abstracción

- Si tenemos TADs, vamos a usar sus observadores, no sus procs

```
Modulo StockImpl implementa Stock {
    clientes: Conjunto<Cliente>
    ventas: Diccionario<Cliente, Venta>
}
```

```
TAD Stock {
    obs ventas: dict<Venta, Cliente>
}
```

```
PredAbs(s': StockImpl, s: Stock) {
  (forall c: Cliente :: c in s'.ventas.data ==>
    s'.ventas.data[c] in s.ventas && s.ventas[s'.ventas.data[c]] == c) &&
  |s'.ventas.data| == |s.ventas|
```

Operaciones

```
Modulo ConjArr<T> implementa Conjunto<T> {
    arr: Array<T>
    largo: int

    InvRep(c': ConjArr<T>) {
        ...
    }

    FuncAbs(c': ConjArr<T>): Conjunto<T> {
        ...
    }

    proc nuevoConjArr(tamMax: int): ConjArr<T>
        requiere ...
        asegura ...
        complejidad ...
    {
        ... pseudocódigo ...
    }

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

Operaciones

- La pre y postcondición de las operaciones refiere a la estructura de implementación

```
proc agregar(inout c': ConjArr<T>, in e: T)
  requiere c’.largo < c’.arr.length
  asegura e in c.arr
  asegura c’.largo ==
    if e in old(c’).arr then
      old(c’).largo
    else
      old(c’).largo+1
```

Operaciones

- Ahora sí escribimos el algoritmo

```
proc agregar(inout c': ConjArr<T>, in e: T) {
    c’.arr[c’.largo] := e
    c’.largo := c’.largo + 1
}
```

Operaciones

- Cómo escribimos el algoritmo? Pseudocódigo (con que se entienda está bien)

  - Declaración de variables

```
var x: int
var c: Conjunto<real>
```

  - Asignación

```
x := 54
```

  - Condicional

```
if condicion then
  ...
else
  ...
fi
```

  - Llamadas a otros procs

```
c.agregar(1)
b := c.vacio()
```

  - Ciclo

```
while condicion do
  …
end
```

Operaciones

- Recuerden que si una variable es de tipo abstracto (TAD) hay que asignarle un valor de un tipo concreto (módulo)

```
var c: Conjunto<real>

c := ConjuntoArrOrdenado.NuevoConjunto(10)
```

Memoria dinámica

- Para tipos complejos vamos a tener que pedir memoria con el operador *new*

```
var c: tupla<int, int, int>
c := new tupla<int, int, int>(10, 10, 10)
c[0] := 35
```

- Si no pedimos, vale *null*

```
var c: tupla<int, int, int>
if c == null {
  ...
}
```

Memoria dinámica

- Si intentamos usar una variable que vale *null,* es error de programa (EXPLOTA)

```
var c: tupla<int, int, int>

c[0] := 35
```

explota!

```
var c: Conjunto<int>

if c.vacio() then
  ...
}
```

explota!

Memoria dinámica

```
Modulo ConjArr<T> implementa Conjunto<T> {
    arr: Array<T>
    largo: int
```

- Cómo inicializamos una estructura

```
proc NuevoConjArr(in tam: int): ConjArr<T>

    res := new ConjArr<T>()
    res.largo := 0
    res.arr := new Array<T>(tam)
    return res
```

Ejercicio de TAD

Diego Bendersky, 27/9/2023

Ejercicio 3. Un `cache` es una estructura de acceso rápido (tradicionalmente almacenada en memoria) que sirve para guardar temporariamente resultados que son consultados con frecuencia. Tiene la misma interface que un diccionario: guarda valores asociados a claves. Los datos almacenados en un cache pueden *desaparecer* en cualquier momento, en función de diferentes criterios.

Especificar caches genéricos (con claves de tipo K y valores de tipo V) que respeten las siguientes políticas de borrado automático. En todos los casos puede asumir que existe una función `now()` que devuelve la hora actual (asuma que es de tipo real)

a) `TTL` o `time-to-live`:
El cache tiene asociado un máximo tiempo de vida para sus elementos. Los elementos se borran automáticamente cuando se alcanza el tiempo de vida (contando desde que fueron agregados por última vez).

b) `FIFO` o `first-in-first-out`:
El cache tiene una capacidad máxima (máximo número de claves). Si se alcanza esa capacidad máxima se borra automáticamente la clave que fue seteada por última vez hace más tiempo.

c) `LRU` o `last-recently-used`:
El cache tiene una capacidad máxima (máximo número de claves). Si se alcanza esa capacidad máxima se borra automáticamente la clave que fue accedida por última vez hace más tiempo. Si no fue accedida nunca, se considera el momento en que fue agregada.

```
TAD CacheTTL<K, V> {
        obs data: dict<K, V>            // los datos
        obs agregado: dict<K, real>     // el momento en que fue agregado
        obs ttl: real                   // máximo tiempo de vida

        proc nuevoCache(in ttl: real): CacheTTL<K, V>
            asegura res.data == {}
            asegura res.agregado == {}
            asegura res.ttl == ttl

        proc esta(in c: CacheTTL<K, V>, in k: K): bool
            // devuelve true si la clave está en la data y no se venció
            asegura res == true <==> k in c.data &&L now() - c.agregado[k] < c.ttl

        proc definir(inout c: CacheTTL<K, V>, in k: K k, in v: V)
            // agrega la nueva asociación clave/valor
            asegura c.data == setKey(old(c).data, k, v)

            // agrega el tiempo en que fue agregado
            asegura c.agregado == setKey(old(c).agregado, k, now())

            // ttl no cambia
            asegura c.ttl == old(c).ttl

        proc obtener(in c: CacheTTL<K, V>, in k: K): V
            // requiere que la clave esté en la data y que no haya vencido
            requiere k in c.data &&L now() - c.agregado[k] < c.ttl

            // devuelve el dato asociado a la clave
            asegura res == c.data[k]

        proc borrar(inout c: CacheTTL<K, V>, in k: K)
            // borra la clave de la data
            asegura c.data == delKey(old(c).data, k)

            // borra el tiempo en que fue agregado
            asegura c.agregado == delKey(old(c).agregado, k)

            // ttl no cambia
            asegura c.ttl == old(c).ttl
    }
```

```
TAD CacheLRU<K, V> {
    obs data: dict<K, V>              // los datos
    obs accedido: dict<K, real>       // el momento en que fue accedido por última vez
    obs cap: int                      // capacidad máxima

    proc nuevoCache(in cap: int): CacheLRU<K, V>
        asegura res.data == {}
        asegura res.accedido == {}
        asegura res.cap == cap

    proc esta(in c: CacheLRU<K, V>, in k: K): bool
        // devuelve true si la clave está en la data
        asegura res == true <==> k in c.data

    proc obtener(inout c: CacheLRU<K, V>, in k: K): V
        // requiere que la clave esté en la data
        requiere k in c.data

        // devuelve el dato asociado
        asegura res == c.data[k]

        // pero también cambia el momento en que fue accedido
        asegura c.accedido == setKey(old(c).accedido, k, now())

        // y no cambia ni cap ni data
        asegura c.cap == old(c).cap && c.data == old(c).data

    proc borrar(inout c: CacheLRU<K, V>, in k: K)
        // borra la clave de la data
        asegura c.data == delKey(old(c).data, k)

        // borra el tiempo en que fue accedido
        asegura c.accedido == delKey(old(c).accedido, k)

        // cap no cambia
        asegura c.cap == old(c).cap

    proc definir(inout c: CacheLRU<K, V>, in k: K k, in v: V)
```

```
proc definir(inout c: CacheLRU<K, V>, in k: K k, in v: V)
    // hay dos casos: si hay lugar o si se llenó la capacidad

    // 1: si hay lugar, seteo la clave y todo el resto de la data queda igual
    asegura |c.data| < c.cap ==> c.data == setKey(old(c).data, k, v)

    // 2: si no hay lugar, seteo la clave pero no queda todo el resto igual:
    // la clave accedida hace más tiempo ya no va a estar más
    asegura |c.data| == c.cap ==> (
        // la nueva clave va a estar
        k in c.data &&L c.data[k] == v

        &&

        // el que fue accedido hace más tiempo no va a estar
        exists k': K :: k' in old(c).data &&L esMin(old(c).accedido, k') && !(k in c.data)

        &&

        // todo el resto sí va a estar
        forall k': K :: k' in old(c).data &&L !esMin(old(c).accedido, k') ==>
            k' in c.data && c.data[k'] == old(c).data[k']

        &&

        // la vuelta, para asegurarnos que no se agregaron cosas de más: lo que está ahora estaba
        // antes o es la clave nueva
        forall k': K :: k' in c.data ==> k' in old(c).data || k' == k
    )

    // por último, guardamos el último acceso del elemento nuevo
    asegura c.accedido == setKey(old(c).accedido, k, now())

    // y mencionamos lo que no cambia
    asegura c.cap == old(c).cap

// predicado para facilitar la construcción de los asegura
pred esMin(d: dict<K, real>, k: K) {
    forall k': K :: k' in d ==>L d[k] <= d[k']
}
```