Especificación de TADs parte 2

Algoritmos y Estructuras de Datos

18 de septiembre

Valentino Murga

Funciones observadores

Nuestros observadores pueden recibir parámetros, lo que nos deja especificar sobre elementos específicos. Esto puede ser muy util para armar especificaciones más cortas y legibles, pero hay que tener ciertos cuidados.

```
obs cosa(num: Z): R
```

Si la función no recibe ningún parámetro, es lo mismo que un observador normal.

`obs cosa(): R` es lo mismo que `obs cosa: R`

"Función"

En realidad se llaman así porque les pasamos parámetros y especificamos sobre el resultado para esos parámetros. Sin embargo, la función no tiene una fórmula o programa o nada por el estilo.

Ejemplo: Diccionario<K, V>

¿Cómo podemos especificar un diccionario con las siguientes operaciones?

- nuevoDiccionario
- definir
- obtener
- esta
- borrar

Opción 1: Observador conjunto de tuplas

`obs tuplas: conj<Tupla<K, V>>`

Veamos como se especificarían algunas operaciones.

Opción 1: Observador conjunto de tuplas

`obs tuplas: conj<Tupla<K, V>>`

```
proc nuevoDiccionario(): Diccionario<K,V>
    requiere {True}
    asegura {res.tuplas = {}}
```

En general los generadores son bastante sencillos.

Opción 1: Observador conjunto de tuplas

`obs tuplas: conj<Tupla<K, V>>`

```
proc obtener(in d: Diccionario<K, V>, in k: K): V
    requiere {
        (∃ t: Tupla<K, V>)(t₀ = k ∧ t ∈ d.tuplas)
    }
    asegura {
        (∃ t: Tupla<K, V>)(t₀ = k ∧ t ∈ d.tuplas ∧ res = t₁)
    }
```

Trabajar con tuplas y subíndices suele ser molesto.

Opción 1: Observador conjunto de tuplas

`obs tuplas: conj<Tupla<K, V>>`

```
proc definir(inout d: Diccionario<K, V>, in k: K, in v: V)
    requiere {d = D₀}
    asegura {
        todosLosMismosMenosUno(d, D₀, k) ∧
        unaSolaConClave(d, k) ∧ <k, v> ∈ d.tuplas
    }
```

Ni tan mal. ¿Cómo se ven todosLosMismosMenosUno y unaSolaConClave?

Opción 1: Observador conjunto de tuplas

```
obs tuplas: conj<Tupla<K, V>>
```

```
pred todosLosMismosMenosUno(d1: Diccionario<K, V>, d2: Diccionario<K, V>, k: K) {
    (∀ t: Tupla<K, V>)(t₀ ≠ k ⟹ (t ∈ d1.tuplas ⟺ t ∈ d2.tuplas))
}

pred unaSolaConClave(d: Diccionario<K, V>, k: K) {
    (∃! t: Tupla<K, V>)(t₀ = k ∧ t ∈ d.tuplas)
}
```

Corrección: No se puede usar "∃!". Queda de tarea especificarlo a mano.

No es terrible, pero cuando hagamos el resto de procs se nos va a alargar un poco el TAD. De todas formas es una solución completamente válida. Veamos que pasa con funciones.

Opción 2: Observador función

```
obs claves: conj<K>
obs valor(k: K): V
```

¿Dos observadores? ¿No es peor tener más? Uno ni siquiera es una función.

Si bien significa que vamos a tener que asegurarnos de que ambos se mantengan correctos, dependiendo de como los usemos nos puede quedar algo más sencillo. Veamos como se ven las mismas operaciones ahora.

Opción 2: Observador función

`obs claves: conj<K>`

`obs valor(k: K): V`

```
proc nuevoDiccionario(): Diccionario<K, V>
    requiere {True}
    asegura {res.claves = {}}
```

Por ahora es lo mismo. Hace falta decir algo sobre `valor`?

Opción 2: Observador función

`obs claves: conj<K>`

`obs valor(k: K): V`

```
proc obtener(in d: Diccionario<K, V>, in k: K): V
    requiere {k ∈ d.claves}
    asegura {res = d.valor(k)}
```

Cortito y sencillo, a mi me gusta más este. Vuelvan a comparar con el anterior.

Opción 2: Observador función

`obs claves: conj<K>`

`obs valor(k: K): V`

```
proc definir(inout d: Diccionario<K, V>, in k: K, in v: V)
    requiere {d = D₀}
    asegura {
        d.claves = D₀.claves ∪ {k} ∧ d.valor(k) = v ∧ valoresIgualesMenosUno(d, D₀, k)
    }
```

Para este proc no hay tanta diferencia, en ambos casos tenemos que pedir que las claves que no tocamos se mantengan igual. Veamos como se ve valoresIgualesMenosUno.

Opción 2: Observador función

```
obs claves: conj<K>
```

```
obs valor(k: K): V
```

```
pred valoresIgualesMenosUno(d1: Diccionario<K, V>, d2: Diccionario<K, V>, k: K) {
    (∀ l: K)(l ≠ k ⟹ d1.valor(l) = d2.valor(l))
}
```

Este es un predicado que va a aparecer casi siempre que especifiquen con funciones observadores porque no existe una operación para cambiar el resultado para solo un parámetro. Ahí ganan los tipos complejos con sus "setAt", "setKey", etc.

Opción 3: Observador diccionario

Justo para este caso tenemos un tipo complejo que ya tiene operaciones que encajan perfectamente con lo que queremos especificar y seguramente quede todo sencillísimo. Queda de tarea.

Ejemplo: Conjunto

¿Qué tipos de observadores podríamos usar para especificar un conjunto?

- Una secuencia, teniendo cuidado con elementos repetidos.
- Un conjunto, que nos soluciona todo trivialmente.
- Un diccionario, que tiene olor a implementación.
- Una función que nos diga que elementos están.

Veamos un poco como se vería la última.

Ejemplo: Conjunto

`obs está(elem: T): Bool`

```
proc nuevoConjunto(): Conjunto<T>
    requiere {True}
    asegura {(∀ elem: T)(¬res.está(elem))}
```

En este caso si que es muy importante asegurarnos de que el conjunto empiece sin ningún elemento.

Ejemplo: Conjunto

`obs está(elem: T): Bool`

```
proc pertenece(in c: Conjunto<T>, in elem: T): Bool
    requiere {True}
    asegura {res ⟺ c.está(elem)}
```

Esta operación la tenemos gratis con nuestro observador.

Ejemplo: Conjunto

`obs está(elem: T): Bool`

```
proc agregar(inout c: Conjunto<T>, in elem: T)
    requiere {c = C₀}
    asegura {
        mismosElementosMenosUno(c, C₀, elem) ∧ c.está(elem)
    }
```

Ahí vuelve a aparecer ese predicado que se asegura de que solo toquemos el elemento que nos importa. Si no lo ponemos, nuestro conjunto podría quedar con cualquier cosa. No pedimos nada en el requiere sobre `elem`, ¿tiene sentido?

Ejemplo: Conjunto

```
obs está(elem: T): Bool
```

```
proc intersección(in c1: Conjunto<T>, in c2: Conjunto<T>): Conjunto<T>
    requiere {True}
    asegura {
        (∀ elem: T)((c1.está(elem) ∧ c2.está(elem)) ⟺ res.está(elem))
    }
```

Este tipo de operaciones es particularmente fácil de escribir con funciones observadoras.

Ejemplo: Conjunto

```
obs está(elem: T): Bool
```

```
proc tamaño(in c: Conjunto<T>): Z
    requiere {True}
    asegura {esTamañoDeConjunto(res)}
```

```
pred esTamañoDeConjunto(c: Conjunto<T>, t: Z) {
    (∃ d: conj<T>)((∀ e: T)(e ∈ d ⟺ e ∈ c) ∧ t = |d|)
}
```

Esta es la parte en la que sería muy útil tener un tipo complejo que ya nos regale el tamaño. De todas formas existe solución. El truco de crear un conjunto / secuencia / etc. que cumpla ciertas condiciones es muy útil, seguramente ya se hayan dado cuenta con el TP.

Ejercicio 8. Un `caché` es una capa de almacenamiento de datos de alta velocidad que almacena un subconjunto de datos, normalmente transitorios, de modo que las solicitudes futuras de dichos datos se atienden con mayor rapidez que si se debe acceder a los datos desde la ubicación de almacenamiento principal. El almacenamiento en caché permite reutilizar de forma eficaz los datos recuperados o procesados anteriormente.

Esta estructura comunmente tiene una interface de diccionario: guarda valores asociados a claves, con la diferencia de que 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 operaciones indicadas y las siguientes políticas de borrado automático. Si necesita conocer la fecha y hora actual, puede pasarla como parámetro a los procedimientos o bien puede asumir que existe una función externa $horaActual() : \mathbb{Z}$ que retorna la fecha y hora actual. Asuma que las fechas son números enteros (por ejemplo, la cantidad de segundos desde el 1 de enero de 1970).

```
TAD Cache⟨K, V⟩ {
  proc esta(in c: Cache⟨K, V⟩, in k: K)  : Bool
  proc obtener(in c: Cache⟨K, V⟩, in k: K)  : V
  proc definir(inout c: Cache⟨K, V⟩, in k)  : K
}
```

a) `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 definida por primera vez hace más tiempo.

b) `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.

c) `TTL` o `time-to-live`:
El cache tiene asociado un máximo tiempo de vida para sus elementos. Los elementos se borran automáticamente cuando

Ejercicio 8 de la guía 4: Cache<K, V>

Nos piden modelar algo que se parece mucho al diccionario de antes. La diferencia es que, con cierto critero, hay que invalidar algunas claves automáticamente. Eso ya suena complejo, pero además ese criterio puede implicar al tiempo. Pero, ¿como especificamos sobre cosas que pasan a lo largo del tiempo?

Ejercicio 8 de la guía 4: Cache<K, V>

Nos piden modelar algo que se parece mucho al diccionario de antes. La diferencia es que, con cierto critero, hay que invalidar algunas claves automáticamente. Eso ya suena complejo, pero además ese criterio puede implicar al tiempo. Pero, ¿como especificamos sobre cosas que pasan a lo largo del tiempo?
Por suerte la consigna nos da una pista. Nos dicen que podemos recibir el momento actual como parámetro en los procs o usar una "función externa" llamada `horaActual(): Z`. A mi me gusta la idea del parámetro.

Ejercicio 8 de la guía 4: Cache<K, V>

Vamos a empezar con la versión FIFO, el primero que entra es el primero que sale. Nosotros lo vamos a especificar usando fechas, ¿se puede hacer sin usarlas?

A pensar.

Ejercicio 8 de la guía 4: Cache<K, V>

Observadores

```
obs cache: dict<K, struct<fecha: Z, valor: V>>
```

```
obs capacidad: Z
```

Ejercicio 8 de la guía 4: Cache<K, V>

```
proc nuevaCacheFIFO(in c: Z): CacheFIFO<K, V>
    requiere {c > 0}
    asegura {
        res.capacidad = c ∧ res.cache = {}
    }
```

```
proc está(in c: CacheFIFO<K, V>, in k: K): Bool
    requiere {True}
    asegura {
        res ⟺ k ∈ c.cache
    }
```

Ejercicio 8 de la guía 4: Cache<K, V>

```
proc obtener(in c: CacheFIFO<K, V>, in k: K): V
    requiere {k ∈ c.cache}
    asegura {
        res = c.cache[k].valor
    }
```

Hasta acá no pasó nada raro, nos estamos manejando exactamente igual que con un diccionario normal. A la hora de definir vamos a tener que ocuparnos un poco más.

Ejercicio 8 de la guía 4: Cache<K, V>

```
proc definir(inout c: CacheFIFO<K, V>, in k: K, in v: V, in fAct: Z)
    requiere {c = C₀}
    asegura {
        (((k ∈ C₀.cache ∨ C₀.capacidad > |C₀.cache|) ∧
            c.cache = setKey(C₀.cache, k, <fecha: fAct, valor: v>)) ∨
        ((k ∉ C₀.cache ∧ C₀.capacidad = |C₀.cache|) ∧
            (∃ l: K)(l ∈ C₀.cache ∧ₗ
                (∀ m: K)(l ≠ m ∧ m ∈ C₀.cache ⟹ₗ
                    C₀.cache[l].fecha < C₀.cache[m].fecha) ∧
                c.cache = setKey(delKey(C₀.cache, l), k, <fecha: fAct, valor: v>)
            )
        )) ∧ c.capacidad = C₀.capacidad
    }
```

Es bastante difícil de leer y habría que agregar predicados, pero es importante ver bien lo que está pasando con las fechas.

Ejercicio 8 de la guía 4: Cache<K, V>

```
((k ∈ C₀.cache ∨ C₀.capacidad > |C₀.cache|) ∧
    c.cache = setKey(C₀.cache, k, <fecha: fAct, valor: v>))
```

Cada vez que agregamos un elemento, lo guardamos con la fecha de ese momento. Si todavía hay espacio libre no hace falta hacer nada.

Ejercicio 8 de la guía 4: Cache<K, V>

```
(k ∉ C₀.cache ∧ C₀.capacidad = |C₀.cache|) ∧
    (∃ l: K)(l ∈ C₀.cache ∧ₗ
        (∀ m: K)(l ≠ m ∧ m ∈ C₀.cache ⟹ₗ
            C₀.cache[l].fecha < C₀.cache[m].fecha) ∧
        c.cache = setKey(delKey(C₀.cache, l), k, <fecha: fAct, valor: v>)
    )
```

En el caso en el que ya esté lleno, tenemos que borrar el elemento más viejo para guardar el nuevo.

Ejercicio 8 de la guía 4: Cache<K, V>

¿En algún punto de nuestra especificación le pusimos algún requerimiento a la fecha actual? No. Si bien lo recibimos como un parámetro, en realidad funciona más como un valor conocido que simplemente existe. Por eso la consigna también nos permite usar una "función" en vez de un parámetro.

Ejercicio 8 de la guía 4: Cache<K, V>

Para la versión FIFO quedaba bastante claro cuando teníamos que eliminar a los elementos viejos. ¿Qué pasa con la versión TTL? Nos piden que los elementos se borren automáticamente cuando pasa cierta cantidad de tiempo. ¿En qué parte de la especificación nos ocupamos de eso? ¿En que proc? Pensemos.

Ejercicio 8 de la guía 4: Cache<K, V>

`obs cache: dict<K, struct<fecha: Z, valor: V>>`

`obs duración: Z`

```
proc está(in c: CacheTTL<K, V>, in k: K, in fAct: Z): Bool
    requiere {True}
    asegura {
        res ⟺ (k ∈ c.cache ∧ₗ c.cache[k].fecha + c.duración > fAct)
    }
```

Si bien no tenemos una forma de que nuestro observador borre una clave en un momento particular, no hace falta. Lo único que necesitamos es que las claves que son demasiado viejas no sean válidas.

Ejercicio 8 de la guía 4: Cache<K, V>

```
proc definir(inout c: CacheFIFO<K, V>, in k: K, in v: V, in fAct: Z)
    requiere {c = C₀}
    asegura {
        c.cache = setKey(C₀.cache, k, <fecha: fAct, valor: v>) ∧ c.duración = C₀.duración
    }
```

```
proc obtener(in c: CacheTTL<K, V>, in k: K, in fAct: Z): V
    requiere {k ∈ c.cache ∧ₗ c.cache[k].fecha + c.duración > fAct}
    asegura {
        res = c.cache[k].valor
    }
```

Si la clave se pasó de su duración, vamos a hacer como si no existiera.

Ejercicio 10 de la guía 4: Vivero

Ejercicio 10. Queremos modelar el funcionamiento de un vivero. El vivero arranca su actividad sin ninguna planta y con un monto inicial de dinero.

Las plantas las compramos en un mayorista que nos vende la cantidad que deseemos pero solamente de a una especie por vez. Como vivimos en un país con inflación, cada vez que vamos a comprar tenemos un precio diferente para la misma planta. Para poder comprar plantas tenemos que tener suficiente dinero disponible, ya que el mayorista no acepta fiarnos.

A cada planta le ponemos un precio de venta por unidad. Ese precio tenemos que poder cambiarlo todas las veces que necesitemos. Para simplificar el problema, asumimos que las plantas las vendemos de a un ejemplar (cada venta involucra un solo ejemplar de una única especie). Por supuesto que para poder hacer una venta tenemos que tener *stock* de esa planta y tenemos que haberle fijado un precio previamente. Además, se quiere saber en todo momento cuál es el balance de caja, es decir, el dinero que tiene disponible el vivero.

a) Indique las operaciones (procs) del TAD con todos sus parámetros.

b) Describa el TAD en forma completa, indicando sus observadores, los requiere y asegura de las operaciones. Puede agregar los predicados y funciones auxiliares que necesite, con su correspondiente definición

---

c) ¿qué cambiaría si supiéramos a priori que cada vez que compramos en el mayorista pagamos exactamente el 10 % más que la vez anterior? Describa los cambios en palabras.

Ejercicio 10 de la guía 4: Vivero

Este ya es un ejercicio mucho más completo. Lo que tenemos que modelar ya no es algo tan concreto, es un escenario escrito en lenguaje humano. Va a tener ambigüedades y vamos a tener que tomar decisiones. Empecemos por el punto a, cuales son los procs?

Ejercicio 10 de la guía 4: Vivero

```
Especie es string
```

```
nuevoVivero(in monto: Z): nuevoVivero
```

```
comprarPlantas(inout v: Vivero, in especie: Especie, in cantidad: Z, in precioCompraTotal: Z)
```

```
cambiarPrecio(inout v: Vivero, in especie: Especie, in nuevoPrecio: Z)
```

```
venderPlanta(inout v: Vivero, in especie: Especie)
```

```
balance(in v: Vivero): Z
```

Ejercicio 10 de la guía 4: Vivero

Ahora toca realmente escribir el TAD. ¿Qué observadores vamos a usar? ¿Qué tipos nos convienen? ¿Funciones observadoras?

Ejercicio 10 de la guía 4: Vivero

```
obs fondos: Z
```

```
obs stock(e: Especie): Z
```

```
obs precio(e: Especie): Z
```

También tendría sentido que `stock` y `precio` sean valores en un diccionario donde la especie es la clave. Yo soy fan de las funciones. Pasemos a especificar cada proc de forma completa.

Ejercicio 10 de la guía 4: Vivero

```
proc nuevoVivero(in m: Z): nuevoVivero
    requiere {m > 0}
    asegura {
        res.fondos = m ∧ (∀ e: Especie)(res.stock(e) = 0 ∧ res.precio(e) = -1)
    }
```

```
proc comprarPlantas(inout v: Vivero, in e: Especie, in c: Z, in p: Z)
    requiere {v = V₀ ∧ v.fondos ≥ p ∧ c > 0}
    asegura {
        v.fondos = V₀.fondos - p ∧ v.stock(e) = V₀.stock(e) + c ∧
        mismosPrecios(v, V₀) ∧ mismoStockMenosUno(v, V₀, e)
    }
```

Vuelven a aparecer esos predicados comunes cuando trabajamos con funciones.

Ejercicio 10 de la guía 4: Vivero

```
proc cambiarPrecio(inout v: Vivero, in e: Especie, in p: Z)
    requiere {v = V₀ ∧ p > 0}
    asegura {
        v.fondos = V₀.fondos ∧ v.precio(e) = p ∧
        mismoStock(v, V₀) ∧ mismosPreciosMenosUno(v, V₀, e)
    }
```

Acá tomamos una decisión, podríamos haber pedido que tuviera que haber stock de la especie para cambiarle el precio.

Ejercicio 10 de la guía 4: Vivero

```
proc venderPlanta(inout v: Vivero, in e: Especie)
    requiere {v = v₀ ∧ v.stock(e) > 0 ∧ v.precio(e) > 0}
    asegura {
        v.fondos = V₀.fondos + v.precio(e) ∧ v.stock(e) = V₀.stock(e) - 1 ∧
        mismoStockMenosUno(v, V₀, e) ∧ mismosPrecios(v, V₀)
    }
```

```
proc balance(in v: Vivero): Z
    requiere {True}
    asegura {res = v.fondos}
```

Y con eso el TAD queda completo. Habiendo usado funciones observadoras la cosa quedo bastante agradable a la vista.

Ejercicio 10 de la guía 4: Vivero

Como siempre, esta solo es una solución válida. Hay más y esta puede no ser la mejor. Acuérdense que lo más importante es que la especificación sea correcta, después ya se ocupan de que sea legible (pero por favor traten de no escribir choclos enormes :)). Además, en este tipo de ejercicios, la correctitud de la especificación va a depender de la interpretación del problema. Es muy importante que toda desición que tomen tenga sentido y que no hagan cualquier cosa cuando aparece una ambigüedad.

Suerte!