1. Resolución ejercicio 3

1.1. Enunciado

Se quiere implementar el TAD cola acotada, que es una cola que puede contener a lo sumo n elementos.

```
TAD ColaAcotada<T> {
    obs s: seq<T>
    obs cap: int

    proc colaVacía(in c: int): Cola<T>
        asegura res.cap == c && res.s == []

    proc vacia(in c: Cola<T>): bool
        asegura res == true <==> c.s == []

    proc encolar(inout c: Cola<T>, in e: T)
        requiere |c.s| < c.cap
        asegura c.s == old(c).s + [e]

    proc desencolar(inout c: Cola<T>): T
        requiere c.s != []
        asegura c.s == old(c).s[1..|old(c).s|]
        asegura res == old(c)[0]
}
```

Como estructura de representación se propone utilizar un *buffer circular*. Esta estructura almacena los k elementos de la cola en posiciones contiguas de un arreglo, aunque no necesariamente en las primeras k posiciones. Para ello, se utilizan dos índices que indican en qué posición empieza y en qué posición termina la cola. Los nuevos elementos se encolan "a continuación" de los actuales tomando módulo n, es decir, si el último elemento de la cola se encuentra en la última posición del arreglo, el próximo elemento a encolar se ubicará en la primera posición del arreglo. Notar que para desencolar el primer elemento de la cola, simplemente se avanza el índice que indica dónde empieza la cola (eventualmente volviendo éste a la primera posición del arreglo). En nuestra implementación, además de esto, se pone en cero la posición que se acaba de liberar. Se propone la siguiente estructura de representación.

```
modulo ColaAcotadaImp<T> implementa ColaAcotada<T> {
    var inicio: int
    var fin: int
    var elems: array<T>
}
```

Se pide:

a) Definir el invariante de representación y la función de abstracción.

b) Por qué tiene sentido utilizar un buffer circular para una cola y no para una pila?

1.2. Resolución

a) Antes de comenzar a responder lo que nos pide el inciso recordemos cómo es nuestra implementación. Tenemos un buffer circular que se implementa mediante un arreglo y dos índices indicando la posición donde comienza la cola (dónde está el primer elemento) y donde termina. Si bien no vamos a implementar los procs del TAD, quisiéramos saber cómo determinar si la cola está vacía y cuándo está completa (ya que no podríamos agregar más elementos). Para poder diferenciar estos dos casos, se representa a la cola vacía cuando los dos índices apuntan a la misma dirección del buffer y a la cola completa cuando el índice fin apunta a una dirección previa (en módulo) al de *inicio*. Estas decisiones provocan que el tamaño del buffer deba ser uno más que la capacidad de la cola. De esta manera, el índice *inicio* apunta al primer elemento de la cola

y el índice fin apunta al elemento siguiente al último. Es decir, siempre tenemos al menos una posición del arreglo que no forma parte de la cola en sí.

Ahora sí podemos pensar el invariante de representación y la función de abstracción. Para ambos nos conviene primero definir en palabras que es lo que debe valer, dada la implementación sugerida.

Comencemos con el invariante. Este nos va a indicar las condiciones que debe cumplir la estructura para que sea válida.

Dado que tenemos dos variables que representan los índices del arreglo *elems* queremos que estos no se salgan de los límites del mismo. Esta será la primera condición que deberá cumplir el invariante. La segunda está relacionada a los elementos que hay en el buffer. Para simplificar, podemos asumir que el buffer se encuentra vacío al principio, que tiene todos ceros. El enunciado nos dice que, cuando se desencola un elemento de la cola, la posición que se acaba de liberar se pone en cero. Esto también hay que tenerlo en cuenta ya que si no, no sería una representación válida. Es decir, no puede haber algún elemento distinto de cero fuera de las posiciones que están entre *inicio* y fin. Como el buffer es circular, los índices pueden tomar cualquier par de valores. Para simplificar, dividimos en tres casos: el caso en el que fin es mayor que *inicio*, el caso en el que *inicio* es mayor que fin y el caso en el que son iguales. A partir de ello, obtenemos el siguiente invariante:

```
InvRep(c': ColaAcotadaImpl<T>) {
    0 <= c'.inicio <= c'.elems.length &&
    0 <= c'.fin <= c'.elems.length &&L
    (c'.inicio < c'.fin ==>L (forall i: int :: 0 <= i < c'.inicio ||
    c'.fin <= i < c'.elems.length ==>L c'.elems[i] == 0)) &&
    (c'.inicio > c'.fin ==>L (forall i: int :: c'.fin <= i < c'.inicio
    ==>L c'.elems[i] == 0)) &&
    (c'.inicio == c'.fin ==>L (forall i: int :: 0 <= i <
    c'.elems.length ==>L c'.elems[i] == 0))
}
```

Para la función de abstracción lo que queremos es poder indicar qué instancias del tipo de implementación se corresponden con qué instancias del tipo abstracto. Es decir, dada una instancia del tipo de implementación, la función devolverá la instancia del tipo abstracto a la que representa. Como el TAD de cola tiene dos observadores, eso es lo que tenemos que relacionar. Primero, vemos que la capacidad sea igual a la longitud del arreglo menos uno, que se corresponde con lo que mencionamos antes sobre lo que sucedía con la cola vacía y completa. Después, sabemos que la en el TAD la cola la representamos mediante una secuencia que va modificando su tamaño al agregar y quitar elementos. Queremos que entonces la longitud de la secuencia sea igual a la cantidad de elementos que hay en el buffer. Para determinar este valor, lo que hacemos es ver la diferencia entre el índice fin e *inicio* y tomarle módulo, pues recordemos que estamos trabajando con un buffer circular y los índices pueden estar invertidos. Por último, queremos que los elementos que están en el buffer, sean los mismos que en la secuencia y que estén en el mismo orden. Acá también tomaremos módulo para poder indexar en el arreglo a partir del índice de inicio. A partir de lo visto antes obtenemos la siguiente función de abstracción:

```
FuncAbs(c': ColaAcotadaImpl<T>): ColaAcotada<T> {
    c: ColaAcotada<T> |
    c.cap == c'.elems.length-1 &&
    (c'.fin-c'.inicio)%c'.elems.length == |c.s| &&
    forall i: int :: 0 <= i < |c.s| ==>L
    c.s[i] == c'.elems[(c'.inicio+i)%c'.elems.length]
}
```

b) Veamos primero la diferencia entre ambos tipos de datos. En el caso de la cola tenemos las operaciones encolar y desencolar que nos permiten agregar un elemento al final de la cola y quitar el primer elemento respectivamente. En el caso de la pila, las operaciones apilar y desapilar también permiten agregar y quitar elementos respectivamente, con la diferencia de que en este caso siempre lo lo hacemos del mismo lugar, del comienzo.

Implementar una cola con un buffer circular nos da la libertad de movernos por la estructura sin necesidad de mover elementos. Es decir, si implementáramos la cola con un arreglo fijo provocaría lo siguiente: comenzando en el ínidice cero, encolamos todos los elementos posibles hasta llegar al final. Ahora queremos agregar un nuevo elemento. Como no podemos encolar más; comenzamos a desencolar desde el principio, sacando el elemento de la posición cero. ¿A dónde agregamos el nuevo elemento? Al principio no podemos porque nuestro buffer no es circular. Pero el final del buffer también está ocupado. Por lo que una solución sería correr todos los elementos una posición hacia adelante y encolar el elemento al final. Esto sería

costoso, ya que implica hacer varios cambios en la estructura. La otra solución es implementar la cola con un buffer circular, pudiendo así agregar el elemento al comienzo de la misma y solo mantener las dos variables que nos indican el inicio y el fin.

Para el caso de la pila, implementarla con un buffer circular no nos aportaría ninguna cualidad que podamos aprovechar. En un buffer común lo que haríamos es ir agregando los elementos desde el final hasta el principio simulando una pila y al quitarlos lo haríamos desde el principio teniendo en cuenta el último apilado. Para ello podríamos mantener un índice que nos inidique el inicio de la pila para saber dónde poner el próximo elemento o de dónde quitar el primero. Observar que, en el caso de que la pila se llene, basta con quitar el primer elemento para poder agregar uno nuevo.

Si quisierámos implementar la pila con un buffer circular podríamos hacer lo mismo, con la diferencia que ahora el fin de la pila podría no ser necesariamente el último elemento del buffer. En ese caso, deberíamos mantener dos índices, uno para el comienzo y otro para el final. Lo importante de esto es que el ínidice del final nunca se va a modificar, ya que los elementos se agregan y se sacan del principio de la pila. Pero entonces, no es necesario tener dos índices, únicamente con el primero nos alcanza y es por eso que no tiene sentido el buffer circular para la pila. Ya que usar un buffer circular requiere tener variables adicionales en lugar de una e interpretar el buffer de una forma distinta, lo cual es más costoso.

2. Resolución ejercicio 5

2.1. Enunciado

(Planilla de actividades) Un consultor independiente desea mantener una planilla con las actividades que realiza cada mes en cada uno de los proyectos en los que participa. La planilla que desea mantener se describe con el siguiente TAD.

```
TAD Planilla {
    obs actividades: conj<Actividad>
    obs proyectoDe: dict<Actividad, Proyecto>
    obs mesDe: dict<Actividad, int>
    obs horaDe: dict<Actividad, int>

    proc nuevaPlanilla(): Planilla

    proc totProyxMes(in p: Planilla, in m: Mes, in r: Proyecto): int

    proc agregar(
        inout p: Planilla,
        in a: Actividad,
        in r: Proyecto,
        in mes: int,
        in horas: int
    )
}

Actividad es string
Proyecto es string
```

Se propone la siguiente estructura para representar dicho TAD

```
modulo PlanillaImpl implementa Planilla {
    var detalle: Diccionario<
        Actividad, struct<proy: Proyecto, mes: int, horas: int>
    >
    var horasPorMes: Diccionario<proyecto, Array<int>>
}

Actividad es string
Proyecto es string
```

Se pide:

a) Escribir formalmente y en castellano el invariante de representación.

b) Escribir la función de abstracción.

2.2. Resolución

Antes de responder a las consignas analicemos un poco el enunciado. El TAD planilla tiene cuatro observadores: un conjunto de actividades que realiza el consultor independiente, para cada actividad a qué proyecto corresponde, en qué mes se realiza cada actividad y cuántas horas tiene asignada esa actividad dentro del mes correspondiente.

Por otro lado, el módulo que implementa el TAD tiene la variable *detalle* y $horasPorMes$. La primera es un diccionario que dada una actividad nos da un struct que tiene el proyecto al que pertenece la actividad, el mes en el que se realiza y la cantidad de horas de esa actividad. La segunda, como su nombre lo indica, es un diccionario que dado un proyecto nos da un arreglo de doce elementos donde cada posición se corresponde con un mes del año y guarda la cantidad de horas dedicadas a ese proyecto en ese mes. Es valor se corresponde con la suma de la cantidad de horas de cada actividad que pertenece al proyecto.

Dicho esto, podemos plantear el invariante de representación. Debemos ver que las estructuras satisfacen ciertas condiciones. Empezando por *detalle* se tiene que, para toda actividad, el mes en el que se realiza la actividad debe ser un número entre 0 y 11 (no lo tomamos entre 1 y 12 para poder indexar en el arreglo). Con la cantidad de horas pasa algo similar: para simplificar la implementación podemos decir que las horas hacen referencia a la cantidad de horas dentro de un día y que por lo tanto tiene que ser un valor entre 0 y 24 (sin incluir). Por último, decimos que para toda actividad, el proyecto al que pertenece está en el diccionario $horasPorMes$. Una interpretación posible, sería que también todos los proyectos que están en este último diccionario, tienen alguna hora asignada, por lo que también debemos revisar que para todos los proyectos haya alguna actividad que tienen asignada, lo cual lo buscamos en el diccionario *detalle*.

Esta primera parte del invariante quedaría así:

```
InvRep(p': PlanillaImpl) {
    forall a: Actividad :: a in p'.detalle ==>L
        0 <= p'.detalle[a].mes < 12 &&
        p'.detalle[a].proy in p'.horasPorMes
    &&
    forall r: Proyecto :: r in p'.horasPorMes ==>L
    (exists a: Actividad :: a in p'.detalle &&L p'.detalle[a].proy == r)
    &&
    ...
```

Pasemos ahora a analizar qué condiciones debe satisdacer el diccionario $horasPorMes$. Debemos tene en cuenta que el arreglo se corresponde con la cantidad de horas por mes de cada proyecto, por lo que su tamaño debería ser doce, indicando los doce meses. Para saber qué valor debe haber en cada posición de esta arreglo recurrimos a lo que definimos antes. Es decir, en cada posición debemos tener la suma de la cantidad de horas de cada actividad que pertenece al proyecto en ese mes. Para ello, recurrimos a la información que tenemos en el otro diccionario y calculamos para cada proyecto y cada mes, qué actividades lo forman, cuál de ellas está en el mes correspondiente, y la cantidad de horas que suman.

Uniendo lo anterior con esto último, obtenemos el siguiente invariante de representación:

```
InvRep(p': PlanillaImpl) {
    forall a: Actividad :: a in p'.detalle ==>L
        0 <= p'.detalle[a].mes < 12 &&
        p'.detalle[a].proy in p'.horasPorMes
    &&
    forall r: Proyecto :: r in p'.horasPorMes ==>L
    (exists a: Actividad :: a in p'.detalle &&L p'.detalle[a].proy == r)
    &&
    forall r: Proyecto :: r in p'.horasPorMes ==>L p'.horasPorMes[r].length == 12
    &&
    forall r: Proyecto, m: int :: r in p'.horasPorMes &&
    0 <= m < 12 ==>L p'.horasPorMes[r][m] ==
```

```
        sum a: Actividad ::
        if a in p'.detalle &&L p'.detalle[a].proy == r && p'.detalle[a].mes == m then
        p'.detalle[a].horas else 0 fi
}
```

La función de abstracción en este caso es más directa. Dada una instancia de implementación, lo primero que debemos decir es que toda actividad que está en *detalle* pertenece al conjunto de actividades del TAD y que no hay otras. Y lo segundo es que si la actividad está en las actividades del TAD entonces valen tres cosas: que el proyecto que le corresponde a esa actividad es el mismo en ambos casos, que el mes es el mismo y que la cantidad de horas es la misma. Esto lo hacemos relacionando el struct del diccionario *detalle* con cada uno de los observadores de la planilla: *proyectoDe*, *mesDe* y *horaDe*. Con todo esto en mente, la función queda así:

```
FuncAbs(p': PlanillaImpl): Planilla {
p: Planilla |
    forall a: Actividad :: (a in p'.detalle <==>
        a in p.actividades) &&L
        p.proyectoDe[a] == p'.detalle[a].proy &&
        p.mesDe[a] == p'.detalle[a].mes &&
        p.horaDe[a] == p'.detalle[a].horas
}
```