Experimentos inmunológicos: justificación de las operaciones

12 de mayo de 2011

A continuación mostramos las justificaciones de todas las operaciones (excepto las de lectura y escritura) del ejercicio “Experimentos inmunológicos” visto en la sesión 10 del laboratorio de PRO2. En una práctica real, solo os pediremos la justificación de un subconjunto reducido de operaciones recursivas e iterativas. Entendemos por justificación de una operación el proceso introducido en los capítulos 5 y 6 de los apuntes de teoría “Correctesa de programes iteratius” y “Programació recursiva” para razonar sobre la corrección de programas.

Notad que en los códigos que aparecen en este documento, tomados de los ficheros `.cpp` del ejercicio, hemos añadido las especificaciones Pre/Post de las operaciones, tomadas de los ficheros `.hpp`. Vosotros deberéis hacer lo mismo para facilitar las justificaciones.

1 La clase `Sistema`

1.1 La operación `anadir_organismo`

Esta operación dota al nuevo organismo de su identificador y organiza sus luchas. Para ello, emplea la operación privada `luchas_org_cola` que lo enfrenta a los de la cola correspondiente y obtiene la información necesaria.

1.1.1 Implementación

```
void Sistema::anadir_organismo(Organismo &o, bool &sobrevive) {
// Pre: o es un organismo sin identificador

   o.anadir_id(id);
   ++id;
   if (o.es_maligno()) {
      luchas_org_cola(def, o, sobrevive);
      if (sobrevive) mal.push(o);
   }
   else {
      luchas_org_cola(mal, o, sobrevive);
      if (sobrevive) def.push(o);
```

```
   }
}
// Post: El parámetro implícito contiene el estado del sistema después del intento
//       de entrada del organismo o; o pasa a tener identificador y a contener
//       el número de organismos que ha destruido;
//       sobrevive indica si o queda vivo en el parámetro implícito o no
```

1.1.2 Justificación

Todas las llamadas a operaciones son correctas. La única operación que no tiene cierto como precondición es `luchas_org_cola`. Notad que su precondición se cumple gracias a la definición de la clase `Sistema`. La postcondición de `luchas_org_cola` hace que las operaciones posteriores a la llamada de `luchas_org_cola` permitan alcanzar la postcondición de `anadir_organismo`.

1.2 La operación auxiliar `luchas_org_cola`

La operación `luchas_org_cola` busca el primer organismo de la cola que consigue matar al organismo nuevo y borra (y cuenta) todos los que resultan destruidos por éste. Para que el borrado sea efectivo hace falta una cola auxiliar para ir guardando los supervivientes. Al final, dicha cola permite construir la de salida mediante una nueva operación privada auxiliar: `concatenar`.

1.2.1 Implementación

```
void Sistema::luchas_org_cola(Cua<Organismo> &c, Organismo &o, bool &sobrevive) const {
// Pre: c = C; C está ordenada crecientemente según los identificadores de sus organismos
// Post: c contiene los organismos de C, ordenados crecientemente por
// identificador y con su contador de víctimas actualizado,
// excepto los que hayan muerto al enfrentarse con o;
// sobrevive indica si o queda vivo despues de sus enfrentamientos;
// o pasa a contener el número de organismos que ha destruido

   Organismo actual;
   int resultado;

   sobrevive = true;
   int long_cola = c.size();

// Inv: sobrevive = ninguno de los elementos visitados de C destruye a o;
//      c = elementos no visitados de C seguidos por los elementos visitados
//      con su contador de víctimas actualizado, excepto los que hayan muerto
//      al enfrentarse con o, en el mismo orden en que estaban en C;
//      long_cola = número de elementos no visitados de C;
//      o tiene actualizado su contador de víctimas
//
// Cota: long_cola

   while (long_cola>0 and sobrevive) {
      actual = c.front();
```

```
      resultado = o.lucha_organismos(actual);
      switch (resultado) {
         case 0: {                    // los dos mueren
            sobrevive = false;        // no actualizamos las víctimas de actual
            o.incrementar_victimas(); // porque éste ya no pertenece al sistema
            break;
         }
         case 1: {                    // o muere, actual sobrevive
            sobrevive= false;
            actual.incrementar_victimas();
            c.push(actual);
            break;
         }
         case 2: {                    // o sobrevive, actual muere
            o.incrementar_victimas();
            break;
         }
         case 3: {                    // los dos sobreviven
            c.push(actual);
            break;
         }
      }
      c.pop();
      --long_cola;
   }

// Post1: Inv1 y (long_cola == 0 or not sobrevive)

// Falta pasar al final de c los elementos no visitados de C (por Inv1, son
// los long_cola primeros de c), para mantener el orden por identificador

   recolocar(long_cola,c);
}
```

1.2.2 Justificación

Justificación del bucle:

- *Inicializaciones*. De entrada, consideramos que no hemos visitado ningún elemento. Puesto que no hay ningún elemento visitado, ninguno destruye a `o`, por lo que `sobrevive` tiene que ser `cierto` para cumplir el invariante. Por la misma razón `long_cola` (número de elementos no visitados de `C`) ha de ser `c.size()`. El resto del invariante también se cumple, ya que `c` coincide con `C` y `o` tiene actualizado su contador de víctimas, ya que no ha habido ninguna.
- *Condición de salida*. Cuando la condición del bucle es falsa tenemos que, o bien `sobrevive` es falso, o bien `long_cola=0`. Ello, junto a `Inv`, nos garantiza `Post1`.
- *Cuerpo del bucle.* La condición del bucle y el invariante garantizan que `c` no está vacía y que ninguno de los elementos visitados de `C` destruye a `o`. Puesto que la última instrucción

de la iteración es `c.pop()`, justo antes de hacerlo debemos garantizar que se cumple el invariante considerando que `c.front()`, el cual ha sido almacenado en `actual`, forma parte de los visitados. Para ello debemos saber cuál es el resultado de la lucha entre `actual` y `o`. Hay cuatro casos:

  - Si el resultado de la lucha es `0`, quiere decir que mueren los dos, por tanto `sobrevive` tiene que ser `falso`, ya que uno de los elementos visitados destruye a `o`. No añadimos `actual` al final de `c`, ya que ésta almacena los visitados que sobreviven (y, como no la tocamos, conserva el orden). Finalmente, puesto que `o` tenía su contador actualizado hasta el momento y destruye al nuevo organismo visitado, hay que incrementar su número de víctimas.
  - Si el resultado de la lucha es `1`, quiere decir que `o` muere y `actual` sobrevive, por tanto `sobrevive` tiene que ser `falso`, ya que uno de los elementos visitados destruye a `o`. Añadimos `actual` al final de `c`, después de incrementar su número de víctimas, ya que es un visitado que sobrevive. Además, `c` conserva el orden ya que el nuevo elemento tiene un identificador mayor que los previamente visitados (y añadidos). Finalmente, puesto que `o` tenía su contador actualizado y no destruye a `actual`, no hace falta modificarlo.
  - Si el resultado de la lucha es `2`, quiere decir que `o` sobrevive y `actual` muere, por tanto `sobrevive` tiene que ser `cierto` (valor que, según la condición del bucle, ya tiene), ya que ninguno de los elementos visitados destruye a `o`. No añadimos `actual` al final de `c`, ya que ésta almacena los visitados que sobreviven (y, como no la tocamos, mantiene el orden). Finalmente, puesto que `o` tenía su contador actualizado hasta el momento y destruye al nuevo organismo visitado, hay que incrementar su número de víctimas.
  - Si el resultado de la lucha es `3`, quiere decir que sobreviven los dos, por tanto `sobrevive` tiene que ser `cierto` (valor que, según la condición del bucle, ya tiene), ya que ninguno de los elementos visitados destruye a `o`. Añadimos `actual` al final de `c`, sin incrementar su número de víctimas, ya que es un visitado que sobrevive pero no destruye a `o`. Además, `c` conserva el orden ya que el nuevo elemento tiene un identificador mayor que los previamente visitados (y añadidos). Finalmente, puesto que `o` tenía su contador actualizado y no destruye a `actual`, no hace falta modificarlo.

  Todas las llamadas a operaciones son correctas. La únicas que no tienen cierto como precondición son `c.front()` y `c.pop()` pero su precondición (que `c` no es vacía) está garantizada por la condición del bucle.

- *Finalización*. El valor `long_cola` cumple los requisitos de una función de cota: (a) es un número natural y (b) decrece a cada iteración.

Justificación de las instrucciones posteriores al bucle:

- Post requiere que `c` pase a contener los organismos de `C`, ordenados crecientemente por identificador y con su contador de víctimas actualizado, excepto los que hayan muerto al enfrentarse con `o`. Pero Post1 indica que `c` ya contiene los elementos que debe contener, solo que podría haber algunos, concretamente los `long_cola` primeros, que al no haber sido visitados no cumplen el orden, ya que estarían colocados por delante de otros (los visitados) con menor identificador. Para corregir esta situación se llama a la operación `recolocar`.

- La llamada a `recolocar` es válida ya que, por definición, el valor de `long_cola` está entre C.size() y 0, con lo que se cumple su precondición.

1.3 La operación auxiliar `recolocar`

1.3.1 Implementación

```
void Sistema::recolocar(int n, queue<Organismo> &c)
{
// Pre: n = N <= c.size(), c = C;
// Post: c contiene los mismos elementos que C, pero los N primeros
elementos de C están al final de c, en el orden relativo original; los
restantes también conservan su orden relativo original

  // Inv:  c contiene los mismos elementos que C, pero los N-n primeros
  //       elementos de C están al final de c, en el orden relativo original;
  //       los restantes también conservan su orden relativo original
  // Cota: n

  while (n>0) {
    c.push(c.front());
    c.pop();
    --n;
  }
}
```

1.3.2 Justificación

- *Inicializaciones*. De entrada `n = N`, con lo que el invariante no obliga a mover ningún elemento al final de c.

- *Condición de salida*. Si `n=0`, Inv dice que ya hemos movido todos los elementos que debíamos mover y por tanto se cumple Post a la salida del bucle.

- *Cuerpo del bucle*. Puesto que la última instrucción de la iteración es `-n;`, justo antes de hacerlo tenemos que garantizar que se cumple `Inv`, es decir, el número de elementos movidos ha de ser `N-(n-1) = N-n+1`, uno más que en la vuelta anterior. El candidato será el primero de `c`.

La precondición de las llamadas a `c.front()` y `c.pop()` se cumplen seguro porque si `n>0, n<=N` y `<= c.size()` entonces `c` no es vacía.

- *Finalización*. El número `n` cumple los requisitos de una función de cota: (a) es un número natural y (b) decrece a cada iteración.

2 La clase `Organismo`

2.1 La operación `lucha_organismos`

Ésta es la operación más importante del módulo. Dados dos organismos, hay que decidir primero si van a luchar de verdad o no, es decir, si sus estructuras celulares son simétricas o no. Por eso, introducimos la operación auxiliar `simetricos` en la parte privada. Notad que en la cabecera de la operación `simetricos` decimos que los parámetros han de ser árboles de `Celula`, pero la operación es independiente del tipo de los elementos del árbol, ya que éstos nunca se llegan a consultar. Asímismo, debemos disponer de otra operación que aplique las luchas de las células de dos árboles, sabiendo que son simétricos. Aquí sí es relevante el hecho de que los árboles sean de células. Esta nueva operación privada se llama `lucha_arboles`.

2.1.1 Implementación

```
int Organismo::lucha_organismos(const Organismo &o2) const {
// Pre: El parámetro implícito (o1) y o2 están compuestos por células con el
//      mismo número de parámetros

   int n;
   Arbre<Celula> aux1 = celulas;
   Arbre<Celula> aux2 = o2.celulas;
   if (simetricos(aux1, aux2)) {
      aux1 = celulas;
      aux2 = o2.celulas;
      pair<int, int> m = lucha_arboles(aux1, aux2);
      if (m.first == m.second) n = 0;
      else if (m.first < m.second) n = 1;
      else n = 2; // m.first > m.second
   }
   else n = 3;
   return n;

// Post: Retorna el resultado de la lucha entre o1  y o2, que es
// 0 <=> o1 y o2 resultan destruidos;  1 <=> o1 resulta destruido y o2 no;
// 2 <=> o1 no resulta destruido y o2 sí; 3 <=> ni o1 ni o2 resultan destruidos
}
```

2.1.2 Justificación

Esta operación no contiene ningún bucle ni ninguna llamada recursiva a justificar. Básicamente, tenemos que preocuparnos de que las llamadas a otras operaciones sean correctas y que permitan obtener la postcondición. Por un lado, la precondición de `lucha_arboles` se cumple por la condición de la rama del `if` desde donde se realiza su llamada. Por otro lado, los resultados de `lucha_arboles` permiten obtener los de `lucha_organismos` de manera inmediata. La precondición de `simetricos` se satisface con cualquier par de árboles.

2.2 La operación auxiliar `simetricos`

2.2.1 Implementación

```
bool Organismo::simetricos(Arbre<Celula> &a1, Arbre<Celula> &a2) const {
// Pre: a1 = A1; a2 = A2

   bool b;
   if (a1.es_buit() or a2.es_buit()) b = a1.es_buit() and a2.es_buit();
   else {
    Arbre<Celula> fe1, fe2, fd1, fd2;
    a1.fills(fe1,fd1); // fe1 = hi(A1); fd1 = hd(A2); a1 = buit
    a2.fills(fe2,fd2); // fe2 = hi(A2); fd1 = hd(A2); a2 = buit
    b = simetricos(fe1, fd2);
      // HI1: b indica si hi(A1) y hd(A2) son simétricos
      if (b) { b = simetricos(fd1, fe2);
               // {HI2: b indica si hd(A1) y hi(A2) son simétricos}
      }
   }
   return b;
// Post: El resultado indica si A1 y A2 son simétricos
}
```

2.2.2 Justificación

- *Caso directo*. Si uno de los dos árboles es vacío entonces `a1` y `a2` son simétricos si y sólo si los dos son vacíos. Por tanto, el código del caso directo nos lleva a la postcondición. Las llamadas a `es_buit` son correctas porque `es_buit` tiene cierto como precondición.

- *Caso recursivo*. Si ninguno de los dos es vacío entonces `a1` y `a2` son simétricos si y sólo si `hi(a1)` y `hd(a2)` son simétricos y `hd(a1)` y `hi(a2)` son simétricos. Por HI1, `b` nos indica si `hi(a1)` y `hd(a2)` son simétricos. Por tanto, si `b` es falsa entonces `b` cumple la postcondición, ya que `a1` y `a2` no son simétricos. Por el contrario, si `b` es cierta, entonces `a1` y `a2` son simétricos si y sólo si `hd(a1)` y `hi(a2)` son simétricos. En tal caso, HI2 nos garantiza la postcondición.

  Las llamadas a `fills()` son correctas, puesto que podemos garantizar en la rama del `if` que nos encontramos que los árboles no son vacíos. Aparte, las dos llamadas recursivas cumplen la precondición de la función.

- *Finalización*. El número de elementos de `a1` cumple los requisitos de una función de cota: (a) es un número natural y (b) decrece a cada llamada recursiva (también podríamos haber escogido como cota el número de elementos de `a2`.

2.3 La operación auxiliar `lucha_arboles`

2.3.1 Implementación

```
pair<int,int> Organismo::lucha_arboles(Arbre<Celula> &a1, Arbre<Celula> &a2) const {
// Pre: a1 y a2 son simétricos; a1 = A1; a2 = A2
   pair<int,int> n;

   if (a1.es_buit()) {
      n.first = 0;
      n.second = 0;
   }
   else {
      int nraiz = a1.arrel().lucha_celulas(a2.arrel());
      n.first = 0;
      if (nraiz == 1) ++n.first;
      n.second = 0;
      if (nraiz == 2) ++n.second;
      Arbre<Celula> fe1, fe2, fd1, fd2;
      a1.fills(fe1,fd1); // fe1 = hi(A1); fd1 = hd(A2); a1 = buit
      a2.fills(fe2,fd2); // fe2 = hi(A2); fd1 = hd(A2); a2 = buit
      pair<int,int> na = lucha_arboles(fe1, fd2);
//        HI1: na.first = número de células de hi(A1) que vencen a su
//                        correspondiente en hd(A2);
//             na.second = número de células de hd(A2) que vencen a su
//                         correspondiente en hi(A1)
      pair<int, int> nb = lucha_arboles(fd1, fe2);
//        HI2: nb.first = número de células de hd(A1) que vencen a su
//                        correspondiente en hi(A2);
//             nb.second = número de células de hi(A2) que vencen a su
//                         correspondiente en hd(a1)
      n.first += na.first + nb.first;
      n.second += nb.second + na.second;
   }
   return n;
// Post: n.first = número de células de A1 que vencen a su correspondiente en A2;
//       n.second = número de células de A2 que vencen a su correspondiente en A1
}
```

2.3.2 Justificación

- *Caso directo*. Si `a1` es vacío, como `a1` y `a2` son simétricos, entonces `a2` también es vacío. En este caso, el número de células de cada árbol que vencen a las del otro es `0`.

- *Caso recursivo*. El código del caso recursivo consiste en agregar el resultado de la lucha entre las raíces al resultado de las llamadas recursivas. Primero, se calcula la contribución

de la lucha entre las raíces. Si el resultado de `lucha_celulas` es `1` entonces `a1` vence al menos una vez. Si el resultado es `2` entonces `a2` vence al menos una vez. Si no se da ninguno de los dos casos entonces nadie vence, `n.first` y `n.second` se quedan como están y pasamos a directamente a acumular el resultado de las dos llamadas recursivas. Usando `HI1` y `HI2` tenemos que para llegar hasta la postcondición sólo hacen falta las operaciones que siguen a la llamada y sólo ésas.

Las llamadas a `arrels` y `fills` son correctas, puesto que, por la rama del `if` en la que nos encontramos, los árboles no son vacíos. Sabemos que `a1` no es vacío por la rama del `if` en la que nos encontramos. Como `a1` y `a2` son simétricos, entonces `a2` tampoco es vacío. Las dos llamadas recursivas son correctas ya que como `a1` y `a2` son simétricos gracias a la precondición, entonces tenemos por un lado que `hi(a1)` y `hd(a2)` son simétricos y por otro que `hd(a1)` y `hi(a2)` son simétricos.

- *Finalización*. El número de elementos de `a1` cumple los requisitos de una función de cota: (a) es un número natural y (b) decrece a cada llamada recursiva.

3 La clase `Celula`

3.1 La operación `lucha_celulas`

Esta operación obtiene la diferencia entre el número de posiciones de la primera célula que superan a las de la segunda y viceversa. Después se compara dicha diferencia con los indicadores de tolerancia.

3.1.1 Implementación

```
int Celula::lucha_celulas (const Celula &c2) const {
// Pre: El parámetro implícito (c1) y c2 tienen el mismo número de parámetros

   int i = 0;
   int dif = 0;

// Inv: dif = diferencia entre el número de posiciones en [0..i-1] en que c1 supera a c2
//      y viceversa, 0<=i<=param.size()

   while (i < param.size()) {
     if (param[i] > c2.param[i]) ++dif;
     else if (param[i] < c2.param[i]) --dif;
     ++i;
   }
// A: dif = diferencia entre el número de posiciones totales en que c1 supera a c2 y viceversa
   int n = 3;
   if (dif > c2.i_tol) n = 1;
   else if (dif < -i_tol) n = 2;
   return n;
// Post: Retorna el resultado de la lucha entre c1 y c2, que es:
```

```
// 1 <=> c1 vence a c2; 2 <=> c2 vence a c1; 3 <=> no vence ninguna de las dos
}
```

3.1.2 Justificación

Justificación del bucle:

- *Inicializaciones*. Si ponemos `i` a `0` entonces `dif` ha de contener la diferencia de 0 posiciones y por tanto tiene que valer `0`.
- *Condición de salida*. De la negación de la condición del bucle y de `Inv` tenemos que `i=param.size()` y, por tanto, junto con el resto de `Inv` podemos garantizar `A`.
- *Cuerpo del bucle*. Puesto que la última instrucción de la iteración es `++i`, justo antes de hacerlo tenemos que satisfacer `Inv` para `i+1`, es decir, `dif` tiene que tener la diferencia entre el número de posiciones en `[0..i]`. Puesto que, según `Inv`, `dif` tiene la diferencia entre el número de posiciones en `[0..i-1]`, si el parámetro i-ésimo de `c1` es mayor que el de `c2` entonces la diferencia tiene que ser uno más y si es menor entonces la diferencia tiene que ser uno menos. Si son iguales la diferencia es la misma.
- *Finalización*. `param.size()-i` cumple los requisitos de una función de cota: (a) es un número natural (por el invariante sabemos que `i` como mucho puede valer `param.size()`) y (b) decrece a cada iteración del bucle.

Justificación de las instrucciones posteriores al bucle:

- Hemos visto que a la salida del bucle se satisface `A` gracias a `Inv` y a que la condición del bucle es falsa. Las instrucciones que siguen determinan el resultado de la lucha siguiendo el criterio definido en el enunciado hasta llegar a `Post`, la postcondición final de la operación.