Exemples de demostració de correctesa d'algorismes recursius

Ricard Gavaldà

24 d'octubre de 2015

1 Nombres binomials

```
// Pre: n >= m >= 0
// Post: el resultat es el binomial (n sobre m)
int binomial (int n, int m) {
    if (m == 0) return 1;
    else return (n*binomial(n-1,m-1))/m;
}
```

1. Triem "mida"= valor de la variable m. Per la Pre sempre és ≥ 0 (si les crides es fan correctament)

2. Quan la mida (m) és 0, el programa entra en el cas base. I la crida recursiva es fa amb una mida més petita que la de m ($m-1$) Per tant el programa acaba sempre.

3. Ara anem a veure que sempre s'acaba satisfent la Post. Mirem primer el cas base. Quan $m = 0$, el programa retorna 1, que coincideix amb $n!/(0!(n-0)!) = 1$ i se satisfà la Post.

4. Suposem altrament que $m > 0$. Fem la Hipòtesi d'inducció: Per a qualsevol n' i m' que satisfan la Pre *i que a més satisfan* $m' < m$, `binomial(n',m')` retorna el binomial de n' sobre m'.

5. Sabem que $m > 0$ i a més per la Pre que $n \geq m$. La crida recursiva es amb paràmetres de mida més petita ($m - 1 \leq m$) i satisfent la Pre, ja que $n - 1 \geq m - 1 \geq 0$.

6. Així doncs podem aplicar la hipòtesi d'inducció: `binomial(n-1,m-1)` retorna $n - 1$ sobre $m - 1$, diguem-li v a aquest valor.

7. El programa retorna $(n * v)/m$. Fent una mica d'aritmètica veiem que

$$\frac{n\,v}{m} = \frac{n\binom{n-1}{m-1}}{m} = \frac{n(n-1)!}{(m-1)!((n-1)-(m-1))!m} = \frac{n!}{m!(n-m)!} = \binom{n}{m},$$

   com demana la Precondició. La divisió és entera però com que sabem que el binomial ha de donar un valor enter no hi ha arrodoniment. A més, no és una divisió per 0 que és la única operació que podria donar error (excepte *overflows*).

8. Observació: En els pas anterior ha calgut aplicar la definició del binomial de n sobre m per acabar la demostració. Això és bon senyal: un argument que no faci ús de què són realment els nombres binomials es podria aplicar igualment a un programa que calcula una altra cosa, i per tant segur que seria una demostració incorrecta o incompleta.

9. En resum, tant en els dos casos base com en el cas recursiu retornem el valor demanat per la Postcondició, que és el que es volia demostrar.

2 Producte ràpid

```
// Pre: x > 0, y >= 0
// Post: el resultat es x elevat a y
int potencia(int x, int y) {
   if (y == 0) return 1;
   else if (y%2 == 1) return x*potencia(x,y-1);
   else return potencia(x*x,y/2);
}
```

1. Triem "mida"= valor de la variable y. Per la Pre sempre és ≥ 0 (si les crides es fan correctament)

2. Les crides recursives es fan quan $y \neq 0$, que per tant vol dir quan $y > 0$. I per a tot $y > 0$, tant $y - 1$ com $y/2$ són menors que y. Això demostra que el programa sempre acaba amb qualsevol valor de y que satisfaci la Pre.

3. Suposem que entrem en el cas base, en què $y = 0$. En aquest cas el el programa retorna 1, que coincideix amb x^y (ja que la Pre exclou el cas indefinit 0^0). Per tant, es compleix la Post.

4. Suposem altrament que no entrem en el cas base, és a dir, $y \neq 0$. Com que la Pre diu que $y \geq 0$, de fet sabem que $y > 0$.

5. Fem la Hipòtesi d'Inducció. Per a qualsevol y' que satisfà la Pre ($y' \geq 0$), *i que a més satisfà* $y' < y$, `potencia(x,y')` retorna x elevat a y'.

6. Si $y > 0$ entrem o en el primer o en el segon cas recursiu. Observem primer que les crides respecten la Pre. Com que $y > 0$, tant $y - 1$ com $y/2$ són ≥ 0, que es el que demana la Precondició, i a més estrictament més petits que y. Això vol dir que podrem aplicar la Hipòtesi d'Inducció en tots dos casos recursius.

7. Si $y\%2 = 1$, entrem en el primer cas recursiu, Aplicant la hipòtesi d'inducció sabem que el valor retornat per `potencia(x,y-1)` és x^{y-1}. El valor retornat és pel programa en aquest cas és x*`potencia(x,y-1)`$= x * (x^{y-1}) = x^{1+y-1} = x^y$, com demana la Postcondició.

8. Si, pel contrari, $y\%2 \neq 1$ (que vol dir $y\%2 = 0$), entrem al segon cas recursiu. Aplicant la h.i. sabem que el valor retornat per `potencia(x*x,y/2)` és $(x * x)^{y/2} = x^{2*(y/2)}$. Com que y, és parell, $y/2$ és el mateix vist com a la divisió entera que és que com a divisió real, i per tant $2 * (y/2) = y$ i $x^{2*(y/2)} = x^y$. Veiem que el programa retorna sense fer cap més càlcul el valor de `potencia(x*x,y/2)`, que és doncs x^y tal com demana la Postcondició.

9. Observació: En els passos anteriors, on diem que $x * (x^{y-1}) = x^y$, i que $(x^2)^{y/2} = x^y$, això és cert per a la potència d'enters, no per a la suma d'enters o del logaritme en base x d'y. Una "demostració" de correctesa que no fes servir propietats definitòries de la potència com aquestes seria necessàriament incorrecta.

10. En resum, tant en el cas base com en tots dos casos recursius retornem el valor demanat per la Postcondició, que és el que es volia demostrar.

3 Mergesort de dos vectors

```
// Pre: --
// Post: els elements de v son els inicials, pero ordenats creixentment
void mergesort(vector<double>& v) {
    i_mergesort(v,0,v.size()-1);
}

// Pre: 0 <= e i d < v.size()
// Post: els elements de v[e..d] son els inicials,
//       pero ordenats creixentment
void i_mergesort(vector<double>& v, int e, int d) {
    if (e < d) {
       int m = (e + d)/2;
       mergesort(v, e, m);
       mergesort(v, m + 1, d);
       fusiona(v, e, m, d);
    }
}

// Pre: 0<=e<=m<=d<v.size() i v[e..m] i v[m+1..d], per separat,
//     son ordenats creixentment
// Post: els elements de v[e..d] son els inicials,
//     pero ordenats creixentment, i la resta de v no ha canviat
// i la resta de v no ha canviat
void fusiona(vector<double>& v, int e, int m, int d);
```

Observem que no donem la implementació de fusiona, donat que per verificar mergesort només en cal l'especificació. Caldria verificar separadament que una implementació de fusiona és correcta, és a dir, satisfà la seva especificació.

En primer lloc, veiem que la implementació de `mergesort` és correcta si la de `i_mergesort` ho és. La crida a `i_mergesort(v,0,v.size()-1)` satisfà la seva Pre ja que 0 <= 0 i v.size()-1 < v.size(). La Post assegura que v[0..v.size()-1], o sigui tot v, és ordenat, que és el que demana la Post de `mergesort`.

1. Triem "mida"$= d - e$, o la mida del bocí de vector que es demana ordenar.

2. Observem que quan la mida $d - e$ és 0 o negativa, tenim que $e < d$ no és cert, i per tant s'entra al cas base i no es fan més crides recursives. Si pel contrari $e < d$, cal comprovar que les dues crides es fan amb paràmetres de mida més petita que $d - e$ i que satisfan la Pre, és a dir, que

   - $0 <= e <= m < v.size()$
   - $m - e < d - e$
   - $0 <= m + 1 <= d < v.size()$
   - $d - (m + 1) < d - e$

   Tot surt de la Pre i del fet que si $e < d$ llavors $e \leq (e + d)/2 < d$, i ho deixem com a exercici; noteu que és important que /2 és una divisió entera que arrodoneix cap avall. Això demostra que el programa sempre acaba.

3. El cas base, implícit en aquest programa, és quan no passa que $e < d$, o sigui $e \geq d$. Si $e = d$, v[e..e] és un vector d'un element que és ordenat. Si $e > d$, [e..d] és un rang buit, i per tant v[e..d] és un vector buit que també sempre és ordenat.

4. Altrament, si $e > d$, fem la hipòtesi d'inducció següent: per a qualsevol vector v i qualsevol parell d'enters e', d' tals que $d' - e' < d - e$ i a més satisfan la Precondició, `i_mergesort(v,e',d')` retorna $v[e'..d']$ amb els elements inicials però ordenats creixentment.

5. Suposem doncs que passa $e < d$. Ja hem vist que les crides es fan amb paràmetres més petits i que satisfan la Pre. Per tant, aplicant la hipòtesi d'inducció deduïm que després de `i_mergesort(v,e,m)` tenim $v[e..m]$ ordenat, i aplicant *un altre cop* la hipòtesi d'inducció tenim que després de `i_mergesort(v,m+1,d)`, a més, $v[m + 1, d]$ està ordenat.

6. Això significa que els paràmetres e, m, d satisfan la Precondició de fusiona i per tant després de `fusiona(v,e,m,d)` tenim que se satisfà la Post de fusiona, és a dir, $v[e..d]$ està ordenat. El programa acaba en aquest estat, que satisfà la Postcondició.

4 Revessar una llista

Aquí en el fons estem servint dues definicions equivalents de què és el revessat d'una llista l:

- El revessat d'una llista $l = e_1 \dots e_n$ és $e_n \dots e_1$
- El revessat d'una llista l és 1) la mateixa l si l és buida, 2) altrament, si l té un primer element x seguit d'una llista l', el revessat de l és el revessat de l' afegint-hi x al final

```
// Pre: l es una llista amb elements e1...en, n>=0
// Post: l conte la llista amb elements en...e1
void revessar(list<double>& l) {
    if (not l.empty()) {
        double x = *(l.begin());
        l.erase(l.begin());
        revessar(l);
        l.insert(l.end(),x);
    }
}
```

- Triem com a mida $l.size()$, que mai no és negativa.
- La crida recursiva es fa amb la mateixa l però després de fer $l.erase()$, per tant amb una llista de mida menor que la donada. Quan la llista té mida 0, som en el cas base. Això implica que el procediment sempre acaba.
- En el cas base, l és la llista buida, que és igual al seu revessat, i per tant se satisfà la Post.
- Altrament, fem la hipòtesi d'inducció següent: si l té mida n, llavors per a qualsevol llista $l' = a_1 \dots a_m$ de mida $m < n$, $revessa(l')$ deixa en l' la llista $a_m \dots a_1$
- Si la l inicial és $e_1 \dots e_n$, en fer la crida a $revessar(l)$ tenim $x = e_1$ i $l = e_2 \dots e_n$ (per definició de $l.begin()$ i $l.erase()$). Per hipòtesi inducció després de la crida tenim $l = e_n \dots e_2$. Després de fer $l.insert(l.end(), x)$ tenim que $l = (e_n \dots e_2)x = e_n \dots e_2 e_1$, que és el revessat de $e_1 \dots e_n$, com demana la Post.

- Cal observar que les operacions $*(l.begin())$ i $l.erase(l.begin(), x)$ són vàlides perquè satisfan les Pre's de les operacions: les dues demanen que l'iterador que tenen com a paràmetre referenciïn un element de la llista diferent del fictici del final, cosa que és certa perquè la llista no és buida i per tant $l.begin()$ sempre satisfà això.

5 Cerca d'un element en una cua

En el fons estem fent servir la definició recursiva següent de cerca:

- x no és a una cua buida.
- Si una cua c no és buida, x és a c si i només si $x = c.front()$ o bé x és a la cua que resulta de fer $c.pop()$.

```
// Pre:  c = C
// Post: el resultat indica si x apareix en C
bool cerca(queue<double> &c, double x) {
    if (c.empty()) return false;
    else if (c.front() == x) return true;
    else {
        c.pop();
        return cerca(c, x);
    }
}
```

- Triem com a mida $c.size()$, que mai no és negativa.
- La crida recursiva es fa amb la mateixa c però després de fer $c.pop()$, per tant amb una cua de mida menor que la donada. Quan la cua té mida 0, som en el cas base. Això implica que el procediment sempre acaba.
- En el cas base, c és la llista buida, que no conté x, i per tant retornant $false$ satisfem la Post.
- Altrament, si entrem en el primer if, sabem que x és a c i retornem $true$, satisfent la Post.

- Altrament, si c no és buida, fem la hipòtesi d'inducció següent: per a qualsevol c' de mida menor que la de c, `cerca(c',x)` retorna un booleà que diu si x és a c.

- La crida recursiva $cerca(c, x)$ es fa havent fet $c.pop()$. Diguem-li C' al valor de c en aquest moment. Clarament la mida de C' és menor que la del valor inicial C. Aplicant la H.I., sabem que aquesta crida aquesta crida retorna cert si i només si x és a C'.

- Com que la crida es fa quan C no és buida i $C.front()$ no és x, sabem que x és a C si i només si és a C', que és el valor retornat per la crida recursiva. Per tant, el valor retornat pel programa és el requerit per la Postcondició, o sigui, si x és a C, també en aquest cas.

- Observem que les crides a funcions de cua sempre es fan satisfent les seves Pre: empty() té precondició cert, i front() i pop() demanen que el seu p.i. no sigui buit, que se satisfà en ambdós casos.

6 Sumar k a un arbre

```
/* Pre: a = A */
/* Post: a te la mateixa forma que A, el valor de
         cada node d'a es la suma del valor del node
         corresponent d'A mes k */
void suma(Arbre<int> &a, int k) {
    if (not a.es_buit()) {
        int n = a.arrel() + k;
        Arbre<int> a1, a2;
        a.fills(a1,a2);
        suma(a1,k);
        suma(a2,k);
        a.plantar(n,a1,a2);
    }
}
```

Tota la qüestió està en definir recursivament quin és l'arbre resultat de sumar k a un arbre:

- El resultat de sumar k a un arbre buit és l'arbre buit.
- El resultat de sumar k a un arbre amb arrel x i fills $f1$ i $f2$ és plantar $x + k$ damunt dels dos arbres $f1'$ i $f2'$ que resulten recursivament de sumar k a $f1$ i $f2$.

Notem que la definició no recursiva, "sumar k a cada node de l'arbre", no es presta directament a una implementació recursiva.

Amb aquest definició demostrem la correctesa així:

- Triem com a mida $a.size()$.
- Si a és buit, el programa retorna l'arbre buit tal com demana la nostra definició recursiva.
- Altrament, fem la H.I. que "per a tot arbre a' de mida menor que la d'a, la crida `suma(a',k)` amb $a' = A'$ deixa en a' el resultat de sumar k a tots els nodes d'A'".
- Ara apliquem dos cops la H.I. als subarbres $a1$ i $a2$ d'a, més la nostra definició recursiva i ja està.

- Cal comprovar que totes les crides a operacions satisfan les seves precondicions: arrel() es crida amb un p.i. no buit, a.fills(a1,a2) es criden amb a no buit i a1, a2 buits i són objectes diferents, i a.plantar(n,a1,a2) es crida amb a buit, i a, a1, a2 són objectes diferents.