Correctesa de programes iteratius

Ricard Gavaldà

Programació 2
Facultat d'Informàtica de Barcelona, UPC
Primavera 2019

Aquesta presentació no substitueix els apunts

Contingut

Correctesa de programes

Correctesa d'un programa

Definició:

Per tots els valors inicials de les variables que satisfan la Pre,
el programa acaba,
i els valors finals satisfan la Postcondició

- Com sabem que un programa és correcte?
- Només podem fer un nombre finit (i petit) de proves
- Raonament genèric sobre valors possibles de les variables

NO és una demostració de correctesa . . .

```
// Pre: x = X and y = Y >= 0
int p = 0;
while (y > 0) {
   p = p + x;
   y = y - 1;
}
// Post: p = X * Y
```

Ho he provat i . . .

. . . amb 3 i 5 dona 15

. . . amb 0 i 100 dona 0

. . . amb -4 i 5 dona -20

. . . amb 4 i -5 no cal provar (Pre)

. . . per tant, és correcte!

Nombre finit (petit) de casos
!=
Tots els casos

NO és una demostració de correctesa ...

```
// Pre: x = X and y = Y >= 0
int p = 0;
while (y > 0) {
   p = p + x;
   y = y - 1;
}
// Post: p = X * Y
```

"Inicialitzem p a 0.
Llavors, anem sumant x a p i decrementant y.
Repetim fem fins que y és 0, i llavors ja hem acabat.
Ja es veu que a p tindrem X*Y."

Llegir el programa
$\neq$
Dir per què satisfà la seva espec

Com ho fem, doncs?

- Raonament genèric sobre tots els valors
- L'eina principal és la inducció
- En programes recursius, aplicada directament
- En programes iteratius, amagada en l'invariant

Estats i assercions

Com raonar sobre programes (I)

- Estat d'un programa: Tupla de valors de totes les variables

  `(x = 10, y = -5, b = true)`

  `(x = 10, y = -15, b = false)`
- Asserció: Descripció d'un conjunt d'estats

  `P(x,y,b)` = "`b == (x + y > 0)`"
- El comentari "`// P`" o "`/* P */`" en un programa vol dir "en aquest punt es compleix `P`"
- La Pre és l'asserció que se suposa que és certa al principi
- La Post és l'asseció que volem que sigui certa al final

Com raonar sobre programes (II)

- **Mètode:** Anotarem el programa amb assercions que descriuen els estats en diferents punts, i argumentarem que cada anotació està ben feta
- Un programa és correcte si és cert que

```
/* Pre */ programa /* Post */
```

Correctesa de programes iteratius

Correctesa d'un bucle I

Esquema bàsic:

- ```
  // Pre: ...
  inicialitzacions;
  // Pre (del bucle): ...
  while (B) {
    cos
  }
  // Post (del bucle): ...
  tractament final;
  // Post: ...
  ```
  ```

L'invariant: Concepte i ús

- Invariant: Una asserció que és certa després de qualsevol nombre d'iteracions (inclós 0)
- Relació entre les variables que es compleix després de qualsevol nombre d'iteracions
- A més, quan el bucle acaba, implica la Postcondició
- Que una asserció Inv és un invariant es demostra per inducció sobre el nombre d'iteracions i

  (sovint no fem la i explícita: eliminem i)
- Finalment, cal demostrar (potser usant l'Invariant) que el bucle segur que acaba

- Bona documentació d'un bucle: explica per què funciona!

Demostració d'acabament

- Funció de fita: Variable o expressió sobre les variables que diuen quantes iteracions queden com a molt
- Ha de tenir valor enter
- Cal que decreixi (al menys en 1) a cada iteració
- Si fem una iteració més, segur que és > 0

Passos

0 Inventar un Invariant i una funció de fita

Demostrar que:

1 Les inicialitzacions del bucle estableixen l'Invariant

2 Si es compleix l'Invariant i s'entra en el bucle, al final d'una iteració torna a complirse l'Invariant

3 L'Invariant i la *negació* de la condició d'entrada al bucle impliquen la Postcondició

4 La funció de fita decreix a cada iteració

5 Si entrem un cop més al bucle, la funció de fita és >0

Observació: Quantificadors

Propietats útils per verificar recorreguts i cerques:

- $\sum_{j=0}^{i-1} v[j] + v[i] = \sum_{j=0}^{i} v[j]$
- $(\exists j\ :\ 0 \leq j < i\ :\ P(j)) \vee P(i) = (\exists j\ :\ 0 \leq j < i+1\ :\ P(j))$
- $(\forall j\ :\ 0 \leq j < i\ :\ P(j)) \wedge P(i) = (\forall j\ :\ 0 \leq j < i+1\ :\ P(j))$

Exemple: Exponenciació

```
// Pre: x = X and y = Y >= 0
int p = 1;
while (y > 0) {
   p = p * x;
   y = y - 1;
}
// Post: p = X elevat a Y
```

Invariant:

$$x = X,\ y >= 0,\ p = X \text{ elevat a } (Y\text{-}y)$$

Fita: y

Exemple: Suma d'un vector

```
// Pre: cert
double suma(const vector<double>& v) {
    int a = 0;
    double s = 0;
    while (a < v.size()) {
        s += v[a];
        ++a;
    }
    return s;
}
// Post: el resultat es la suma de tots els elements de v
```

Invariant:

$$0 <= a <= v.size() \text{ and } s = \text{suma de } v[0..a-1]$$

Fita: v.size() - a

Exemple: Cerca un element en una llista

```
// Pre: cert
bool pertany(const list<double>& l, double x) {
    list<double>::const_iterator it = l.begin();
    bool trobat = false;
    while (it != l.end() and not trobat) {
        if (*it == x) trobat = true;
        ++it;
    }
    return trobat;
}
// Post: el resultat indica si x apareix en l
```

Invariant:

it apunta a algun element de l (potser el fictici del final) i trobat = (algun element del anterior a it és x)

Funció de fita: ~~l.end()-it~~ Nombre d'elements entre l.end() i it

Exemple: variació de cerca lineal

```
// Pre: cert
// Post: retorna la posició en v d'un estudiant amb dni x,
// o bé -1 si cap estudiant de v té dni x
int posicio(int x, const vector<Estudiant>& v) {
   int i = 0;
   bool trobat = false;
   while (i < v.size() and not trobat) {
      if (v[i].consultar_dni() == x) trobat = true;
      else ++i;
   }
   if (trobat) return i;
   else return -1;
}
```

Invariant: (0 <= i <= v.size()) and (x no és a v[0..i-1]) and (trobat si i només si (i < v.size() and v[i] == x))
Funció de fita: v.size()-i-(1 si trobat)

Exemple: Suma d'una pila

Donada una pila d'enters, calcular-ne la suma dels elements:

```
// Pre: p = P
int suma(stack<int> &p) {
    int n = 0;
    // Inv: ...
    while (not p.empty()) {
        n += p.top();
        p.pop();
    }
    return n;
}
// Post: El resultat es la suma dels elements de P
```

Invariant: s es la suma dels elements de P que no son a p
Funció de fita: alçada de p

Exemple: Sumar k a una llista

Problema: donada una llista i un enter k, transformar-la en una altra resultant de sumar k a cada element de la llista original.

```
// Pre: l = L
void suma_k(list<int> &l, int k) {
    list<int>::iterator it;
    it = l.begin();
    // Inv: ...
    while (it != l.end()) {
        *it += k;
        ++it;
    }
}
// Post: Cada element de l es la suma de k i l'element
//       de L a la seva mateixa posicio
```

Invariant: ...
Funció de fita: ...

Exemple: Revessar una llista

Problema: donada una llista *l*, revessar-la. Per exemple, si *l* és [3, 8, 1, 9] ha de deixar-la contenint [9, 1, 8, 3]

```
// Pre: l = L = e1...en
void revessa(list<int> &l) {
    list<int> laux;
    while (not l.empty()) {
        laux.insert(*(l.begin());
        l.erase(l.begin());
    }
    // laux = el revessat de L
    l = laux;
}
// Post: l = en ... e1
```

Invariant: l = ei...en i laux = ...
Funció de fita: ...
Exercici: Directament sobre l, evitant l = laux. Amb i sense l.size()

Exemple: cerca dicotòmica

```
// Pre: (0 <= esq = E) and (D = dre < v.size())
//       i (v esta ordenat creixentment)
// Post: (x es a v[E..D] sii
//        (0 <= esq < v.size() and  v[esq] = x)
int posicio(double x, const vector<double>& v,
            int esq, int dre) {
   while (esq < dre) {
       int pos = (esq + dre)/2;
       if (v[pos] < x) esq = pos + 1;
       else dre = pos;
   }
   return esq;
}
```

Invariant $\simeq$ ((x es a v[E..D]) sii (x es a v[esq..dre])) Fita: dre-esq.
Millor encara, $\log_2$(dre-esq)

Exemple: comptar nombre d'elements diferents

```
// Pre: cert
// Post: el resultat es el nombre d'elements diferents a v
int diferents(const vector<elem>& v) {
   int n = 0;
   int i = 0;
   while (i < v.size()) {
       int j = i-1;
       while (j >= 0 and v[j] != v[i]) --j;
       if (j < 0) ++n;
       ++i;
   }
   return n;
}
```

Invariant 1: (0 <= i <= v.size()) and
(n es el nombre d'elements diferents en v[0..i-1])

Post del bucle intern: (Invariant 1) and
(j < 0 si i només si v[i] no apareix en v[0..i-1])

Invariant 2: Invariant 1 and (-1 <= j < i) and
(tots els elements de v[j..i-1] són diferents de v[i])

Exemple: comptar nombre d'elements diferents (2)

amb una funció separada:

```
// Pre: [a..b] inclós en [0..v.size()-1]
// Post: retorna cert sii e apareix a v[a..b]
bool apareix(elem e, const vector<elem>& v, int a, int b);

// Pre: cert
// Post: el resultat es el nombre d'elements diferents a v
int diferents(const vector<elem>& v) {
   int n = 0;
   int i = 0;
   while (i < v.size()) {
       if (not apareix(v[i],v,0,i-1)) ++n;
       ++i;
   }
   return n;
}
```

Invariant 1: (0 <= i <= v.size()) and
(n es el nombre d'elements diferents en v[0..i-1])
Es fa servir l'especificació d'`apareix` per verificar
Verificació separada de la implementació d'`apareix`

Disseny inductiu

Disseny inductiu o derivació

Invertim el procés: de la “justificació” a l’algorisme

Donades Pre i Post, proposar:

- un invariant que les generalitzi les dues
- inicialitzacions que, amb la Pre, assegurin invariant
- un cos del bucle que mantingui l’invariant
- una condició del bucle que, negada, impliqui la Post

Nombre de parelles creixents

Donat un vector v, comptar quantes parelles $(v[i], v[i+1])$ conté tals que $v[i] < v[i+1]$.

```
int parcreix(const vector<int>& v);
```

Nombre de parelles creixents

Posem

- una variable i per recorrer el vector,
- una variable p per comptar les parelles creixents trobades.

Busquem un invariant de l'estil

"p conté el nombre de parelles creixents $(v[j], v[j+1])$ amb $j < i$"

Nombre de parelles creixents

"p conté el nombre de parelles creixents $(v[j], v[j+1])$ amb $j < i$"

1. Com establim l'invariant al principi?

$p = 0$, $i = 0$ (no existeix la parella $(v[-1], v[0])$ ni cap anterior)

Nombre de parelles creixents

"p conté el nombre de parelles creixents $(v[j], v[j+1])$ amb $j < i$"

2. Quan acabem? I acabem satisfent la Post?

La darrera parella que cal comprovar és $(v[n-2], v[n-1])$, amb $n = v.size()$

Si la nostra condició de sortida és $i = n - 1$, hem provat totes les parelles $(v[j], v[j+1])$ amb $j < n - 1$, inclosa la $j = n - 2$, que és la $(v[n-2], v[n-1])$ i ja hem acabat

Podem posar de condició del bucle $i < n - 1$, que cobreix bé els casos $n = 0$ i $n = 1$, i que implica sortir quan $i = n - 1$

Nombre de parelles creixents

"p conté el nombre de parelles creixents $(v[j], v[j+1])$ amb $j < i$"

3. Com avancem mantenim l'invariant?

Volem avançar fent $i = i + 1$. Posem que ho fem com a darrera instrucció del cos

Per tant just abans d'incrementar s'hauria de complir que hem provat totes les parelles $(v[j], v[j+1])$ amb $j \leq i$

La que falta doncs és amb $j = i$, que és $(v[i], v[i+1])$

Per tant

```
if (v[i] < v[i+1]) p = p + 1;
i = i + 1;
```

Nombre de parelles creixents

Completem l'invariant amb el rang d'i:

$$(0 \leq i \leq n-1) \text{ AND } (p \text{ conté el nombre de parelles creixents } (v[j], v[j+1]) \text{ amb } j < i)$$

I l'algorisme:

```
int parcreix(const vector<int>& v) {
   int i = 0;
   int p = 0;
   int n = v.size();
   while (i < n - 1) {
     if (v[i] < v[i+1]) p = p + 1;
     i = i + 1;
   }
   return p;
}
```

Ordenació

Ordenar un vector v: Deixar-lo de manera que
"per a tot i, $0 \leq i \leq v.size() - 1$, $v[i] \leq v[i+1]$

Poso una variable j i vull mantenir que

"$v[0..j]$ està ordenat"

Fixem-nos que quan $j = v.size() - 1$ ja tenim tot el vector ordenat.

... i afegeixo

"i tots els elements de $v[0..j]$ són més petits que els de $v[j+1..v.size()-1]$"

Ordenació

"$v[0..j]$ està ordenat i tots els elements de $v[0..j]$ són més petits que els de $v[j+1..v.size()-1]$"

Incrementem $j = j + 1$.

- $v[0..j]$ segueix ordenat! Per què?
- Però no és cert que "$v[0..j]$ és més petit que $v[j+1..v.size()-1]$"
- Només és cert si $v[j+1]$ era l'element més petit de tot $v[j+1...v.size()-1]$
- Que hem de fer?
  - Buscar l'element més petit de $v[j+1...v.size()-1]$
  - Intercanviar-lo amb $v[j+1]$
  - I ara incrementem j

Ordenació per selecció. Exercici: Si no posem la segona part de l'invariant, deriveu la cerca per inserció

Exemple: Prefix de suma màxima d'un vector, I

```
// Pre: v.size() > 0
// Post: el resultat és un i tal que v[0]+...+v[i] és màxima
//       and (-1 <= i < v.size())
int psm(const vector<double>& v);
```

Exemple: Prefix de suma màxima d'un vector, II

```
// Post: el resultat és un i tal que v[0]+...+v[i] és màxima
//       and (-1 <= i < v.size())
```

Rumiem què vol dir "és màxima" ...
Suggereix l'invariant:

```
// Inv: i és tal que v[0]+...+v[i] és màxima
//      entre els valors que satisfan (-1 <= i < j)
```

i, j noves variables del programa

Invariant, segona versió

```
-1 <= i < j < v.size(),
v[0]+...+v[i] >= v[0]+...+v[k] per a tot k en [-1..j-1],
sum = v[0]+...+v[j-1],
isum = v[0]+...+v[i]
```

Exemple: Prefix de suma màxima d'un vector, III

```
// Pre: v.size() > 0
int psm(const vector<double>& v) {
   int i = -1;
   int j = 0;
   double isum = 0;
   double sum = 0;
   while (j < v.size()) {
       // Inv: ...
       sum += v[j];
       if (sum > isum) {
          isum = sum;
          i = j;
       }
       ++j;
   }
   return i;
}
// Post: el resultat és un i tal que v[0]+...+v[i] és màxima
//       and (-1 <= i < v.size())
```