Correctesa de programes iteratius

Programació 2
Facultat d'Informàtica de Barcelona, UPC

Conrado Martínez

Primavera 2019

- Apunts basats en els d'en Ricard Gavaldà
- Aquestes transparències **no** substitueixen els apunts de l'assignatura, els complementen

Correctesa de programes

Correctesa d'un programa

Definició:
L' estat d'un programa en un punt determinat de la execució vé donat pel valor de totes les variables en aquell punt.

```
// Estat = ( x = 3, y = 7, ... )
++x;
// Estat = ( x = 4, y = 7, ... )
```

Correctesa d'un programa

Definició: Correcció d'un programa

Si l'estat inicial del programa o funció satisfà la Precondició, llavors el programa acaba en un nombre finit de pasos i l'estat final satisfà la Postcondició

Correctesa d'un programa

Definició: Correcció d'un programa

Si l'estat inicial del programa o funció satisfà la Precondició, llavors el programa acaba en un nombre finit de pasos i l'estat final satisfà la Postcondició

- Com sabem que un programa és correcte?
- Només podem fer un nombre finit (i petit) de proves
- Raonament genèric sobre els estats del programa

Demostració de correctesa?

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

Demostració de correctesa?

Ho he provat i . . .

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

Demostració de correctesa?

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

Ho he provat i . . .

. . . amb 5 i 3 dona 125

Demostració de correctesa?

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

Ho he provat i ...

...amb 5 i 3 dona 125

...amb 0 i 100 dona 0

Demostració de correctesa?

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

Ho he provat i . . .

. . . amb 5 i 3 dona 125

. . . amb 0 i 100 dona 0

. . . amb -4 i 2 dona 16

Demostració de correctesa?

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

Ho he provat i . . .

. . . amb 5 i 3 dona 125

. . . amb 0 i 100 dona 0

. . . amb -4 i 2 dona 16

. . . amb 4 i -2 no cal provar (no es compleix la Pre)

Demostració de correctesa?

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

Ho he provat i . . .

. . . amb 5 i 3 dona 125

. . . amb 0 i 100 dona 0

. . . amb -4 i 2 dona 16

. . . amb 4 i -2 no cal provar (no es compleix la Pre)

. . . per tant, és correcte!

Demostració de correctesa?

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

Ho he provat i . . .

. . . amb 5 i 3 dona 125

. . . amb 0 i 100 dona 0

. . . amb -4 i 2 dona 16

. . . amb 4 i -2 no cal provar (no es compleix la Pre)

. . . per tant, és correcte!

Nombre finit (petit) de casos

$\neq$

Tots els casos

Demostració de correctesa?

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

Demostració de correctesa?

“Inicialitzem p a 1 (el producte de 0 factors).

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

Demostració de correctesa?

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

"Inicialitzem p a 1 (el producte de 0 factors).
Llavors, anem multiplicant p per x i decrementant y en cada pas.

Demostració de correctesa?

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

"Inicialitzem p a 1 (el producte de 0 factors).
Llavors, anem multiplicant p per x i decrementant y en cada pas.
Repetim fins que $y = 0$, i llavors ja hem acabat.

Demostració de correctesa?

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

"Inicialitzem p a 1 (el producte de 0 factors).
Llavors, anem multiplicant p per x i decrementant y en cada pas.
Repetim fins que $y = 0$, i llavors ja hem acabat.
Ja es veu que a p tindrem X^Y."

Demostració de correctesa?

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

"Inicialitzem p a 1 (el producte de 0 factors).
Llavors, anem multiplicant p per x i decrementant y en cada pas.
Repetim fins que $y = 0$, 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 estats possibles
- L'eina principal és la inducció
- En programes recursius, aplicada directament
- En programes iteratius, amagada en els invariants

Estats i assercions

Com raonar sobre programes

- Recordem: estat d'un programa = valors de totes les variables

```
(x = 10, y = -5, b = true)
(x = 10, y = -15, b = false)
```

Com raonar sobre programes

- Recordem: estat d'un programa = 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)`"

Com raonar sobre programes

- Recordem: estat d'un programa = 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`"

Com raonar sobre programes

- Recordem: estat d'un programa = 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 Precondició (Pre) és l'asserció que l'estat inicial ha de satisfer
- La Postcondició (Post) és l'asserció que ha de ser certa per l'estat final; altrament el programa no satisfà l'especificació

Com raonar sobre programes

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

Com raonar sobre programes

- **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 */
```

Com raonar sobre programes

- Donada una asserció P, $P(x \leftarrow E)$ és l'asserció resultant de reemplaçar les aparicions d'x en l'asserció P per l'expressió E, e.g., $P =$ "$x \geq 5$", $P(x, y+3) =$ "$y + 3 \geq 5$"
- `/*` $P(x, E)$ `*/ x = E /*` P `*/`
- Si `/*` P_1 `*/ S1 /*` Q_1 `*/` és correcte, `/*` P_2 `*/ S2 /*` Q_2 `*/` és correcte i $Q_1 \implies P_2$ llavors

  `/*` P_1 `*/ S1; S2 /*` Q_2 `*/`

  és correcte.

Com raonar sobre programes

- Si /* $P \wedge B$ */ S1 /* Q */ és correcte,
  i /* $P \wedge \neg B$ */ S2 /* Q */ és correcte llavors

  ```
  /* P */
  if (B) S1
  else S2
  /* Q */
  ```

  és correcte.

Correctesa de programes iteratius

Correctesa d'un bucle

Esquema bàsic

```
// Pre: P
inicialitzacions;
// Pre (del bucle): P'
while (B) {
  cos
}
// Post (del bucle): Q'
tractament final;
// Post: Q
```

L'invariant: Concepte i ús

- Invariant: Una asserció I que és certa després de qualsevol nombre d'iteracions (inclós 0); per tant cal que $P' \implies I$
- A més, quan el bucle acaba, implica la Post: $I \wedge \neg B \implies Q'$

L'invariant: Concepte i ús

- Invariant: Una asserció I que és certa després de qualsevol nombre d'iteracions (inclós 0); per tant cal que $P' \implies I$
- A més, quan el bucle acaba, implica la Post: $I \wedge \neg B \implies Q'$
- Que l'asserció I és un invariant es demostra per inducció sobre el nombre d'iteracions i: s'ha de cumplir

Esquema bàsic

```
// I ∧ B
cos del bucle
// I
```

L'invariant: Concepte i ús

- Invariant: Una asserció I que és certa després de qualsevol nombre d'iteracions (inclós 0); per tant cal que $P' \implies I$
- A més, quan el bucle acaba, implica la Post: $I \wedge \neg B \implies Q'$
- Que l'asserció I és un invariant es demostra per inducció sobre el nombre d'iteracions i: s'ha de cumplir

Esquema bàsic

```
// I ∧ B
cos del bucle
// I
```

- Finalment, cal demostrar (potser usant l'invariant I) que el bucle segur que acaba

- Trobar i explicitar l'invariant d'un bucle és molt bona documentació d'un bucle: explica per què funciona!

Demostració d'acabament

- Funció de fita: Una funció f sobre les variables que diuen quantes iteracions queden com a molt
- Ha de tenir valor enter no negatiu: per a qualsevol estat del programa $f \geq 0$
- Cal que decreixi (al menys en 1) a cada iteració
- Si fem una iteració més, segur que $f > 0$

Passos

0 Inventar un invariant I i una funció de fita f

Demostrar que:

1 Les inicialitzacions del bucle estableixen l'invariant: $P' \implies I$

2 Si es compleix l'invariant i s'entra en el bucle, al final d'una iteració torna a complir-se l'invariant: `/*` $I \wedge B$ `*/ cos /*` I `*/`

3 L'invariant i la *negació* de la condició d'entrada al bucle impliquen la Postcondició: $I \wedge \neg B \implies Q'$

4 La funció de fita decreix a cada iteració:
`/*` $I \wedge B \wedge f = F$ `*/ cos /*` $I \wedge f < F$ `*/`

5 Si entrem un cop més al bucle, la funció de fita és estrictament positiva: $I \wedge B \implies f > 0$

Observació: Quantificadors

Propietats útils per verificar recorreguts i cerques:

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

Exemple: Exponenciació

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

- Invariant:

$$x = X \wedge y \geq 0 \wedge p \cdot X^y = X^Y$$

- Fita: y

Exemple: Exponenciació

```
// Pre: x = X ∧ y = Y ≥ 0
int p = 1;
while (y > 0) {
   if (y % 2 == 0) { x = x*x; y = y/2; }
   else { p = p * x; y = y - 1; }
}
// Post: p = X^Y
```

- Invariant:

$$y \geq 0 \wedge p \cdot x^y = X^Y$$

- Fita: y.

Una mica de notació

- Donat un vector v de talla n i dos entres i, $j$amb $0 \leq i, j < n$, $v[i..j]$ denota el subvector entre les components i i j; si $i > j$ llavors $v[i..j]$ és un subvector buit
- Donada una llista L i dos iteradors $it1$ i $it2$ tals que $it2$ apunta a un element posterior a l'apuntat por $it1$ (o $it2 == it1$) llavors $L[it1 : it2)$ denota la subllista de L el primer element de la qual és l'apuntat per $it1$ i l'últim element és el predecessor de l'element apuntat per $it2$
- $L[: it) \equiv L[L.\texttt{begin}(), it)$
- $L[it :) \equiv L[it, L.\texttt{end}())$
- $L[:) \equiv L$

Exemple: Suma d'un vector

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

- Invariant:

$$0 \leq i \leq v.\texttt{size}() \wedge s = \text{suma de } v[0..i-1]$$

- Fita: $v.\texttt{size}() - i$

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
```

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 un element de l ó $it = l.\text{end}()$, i
  $trobat =$ "x pertany a $l[:it)$"

- Funció de fita: nombre d'elements de la subllista $l[it:)$

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:

$$0 \leq i \leq v.\texttt{size}()) \wedge x \notin v[0..i-1] \wedge trobat = \text{``}i < v.\texttt{size}() \wedge v[i] = x\text{''}$$

- Funció de fita: $v.\texttt{size}() - i - [\![trobat]\!]$, $[\![P]\!] = 1$ si P és cert, i $[\![P]\!] = 0$ si P és fals

Exemple: Suma d'una pila

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

```
// Pre: p = [a_1, ..., a_n]
int suma(stack<int>& p) {
    int s = 0;
    while (not p.empty()) {
        s += p.top();
        p.pop();
    }
    return s;
}
// Post: suma(p) = a_1 + ··· + a_n ∧ p = []
```

- Invariant:

$$p = [p_1, \ldots, p_{n-i}] \implies s = p_n + p_{n-1} + \ldots + p_{n-i-1}$$

- 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 = [a_1, ..., a_n]
void suma_k(list<int>& l, int k) {
    list<int>::iterator it;
    it = l.begin();
    while (it != l.end()) {
        *it += k;
        ++it;
    }
}
// Post: l = [a_1 + k, ..., a_n + k]
```

- Invariant:
  $l[it:) = [a_i, \ldots, a_n] \implies l[:it) = [a_1 + k, \ldots, a_{i-1} + k] \wedge i \leq n+1$
- Funció de fita: nombre d'elements de $l[it:)$

Exemple: Revessar una llista

```
// Pre: l = [a_1, ..., a_n]
void revessa(list<int>& l) {
    list<int> laux;
    while (not l.empty()) {
        laux.push_front(*(l.begin()));
        l.pop_front();
    }
    // laux = [a_n, ..., a_1]
    l = laux;
}
// Post: l = [a_n, ..., a_1]
```

- Invariant:

$$l = [a_i, \ldots, a_n] \implies laux = [a_{i-1}, \ldots, a_1] \wedge i \leq n+1$$

- Funció de fita: l.size()

Exercici: Directament sobre l, evitant la llista auxiliar

Exemple: cerca dicotòmica

```
// Pre: 0 ≤ esq = E ∧ D = dre < v.size()∧
//        v està ordenat creixentment
// Post: x és a v[E..D] si i nomes si
//         0 ≤ esq < v.size() ∧ 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: $x \in v[E..D] \Leftrightarrow x \in v[esq..dre] \wedge \ldots$
- Fita: $dre - esq$. Millor encara: $f = \log_2(dre - esq + 1)$

Exemple: comptar nombre d'elements diferents

```
// Pre: cert
// Post: retorna 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;
}
```

Exemple: comptar nombre d'elements diferents

```
// n = N ∧ i < v.size()
int j = i-1;
while (j >= 0 and v[j] != v[i]) --j;
if (j < 0) ++n;
// n = N si v[i] ∉ v[0..i-1]∧
// n = N+1 si v[i] ∈ v[0..i-1]
```

- Invariant (del bucle intern sobre j):

$$v[i] \notin v[j+1..i-1] \wedge j \geq -1$$

- Fita: $j+1$

Exemple: comptar nombre d'elements diferents

```
// Pre: cert
// Post: diferents(v) = 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; ...; if (j < 0) ++n;
     // n = nombre d'elements diferents a v[0..i]
     ++i;
   }
   return n;
}
```

- Invariant (del bucle extern sobre i):

$$0 \leq i \leq v.\texttt{size}() \wedge n = \text{nombre d'elements diferents a } v[0..i]$$

- Fita: $n - i$

Exemple: comptar nombre d'elements diferents (2)

Amb una funció separada:

```
// Pre: [a..b] ⊆ [0..v.size() − 1]
// Post: retorna cert sii x ∈ v[a..b]
template <typename T>
bool apareix(const T& x, const vector<T>& v, int a, int b);

// Pre: cert
// Post: retorna 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;
}
```

Exemple: comptar nombre d'elements diferents (2)

Invariant:

$$0 \leq i \leq v.\texttt{size}() \wedge n = \text{nombre d'elements diferents en } v[0..i-1]$$

- Es fa servir l'especificació d'`apareix` per verificar
- Podem verificar independentment la correcció de `diferents` i d'`apareix`$\Rightarrow$ Modularitat!

Invariants “gràfics”

Sovint podem donar una representació gràfica esquemàtica d’un invariant (i en general d’una asserció), molt més intuitiva i senzilla d’entendre.
Exemple: Donat un vector v d’enters, escriu un procediment que reorganitzi els seus continguts de manera que els elements parells apareguin abans que els elements senars.

```
// Pre: cert
// Post: ???
void reorganitza_parells_senars(vector<int>& v);
```

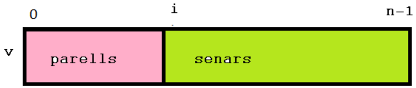
Invariants “gràfics”

```
// Pre: v = V
// Post: ???
void reorganitza_parells_senars(vector<int>& v);
```

- Postcondició formal: v és una permutació de V i

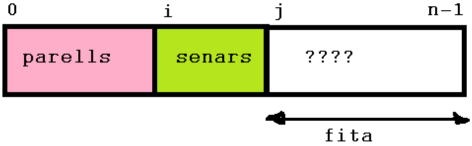
$$\exists i : 0 \le i < v.\mathsf{size}() : \Big(\forall j : 0 \le i < j : v[j] \bmod 2 = 0 \\ \wedge \forall j : i \le j < v.\mathsf{size}() : v[j] \bmod 2 = 1 \Big)$$

- Postcondició “gràfica”:



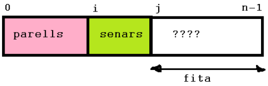
Invariants “gràfics”

```
// Pre: v = V
// Post: ???
void reorganitza_parells_senars(vector<int>& v);
```



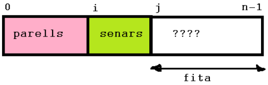
Invariants “gràfics”

```
// Pre: v = V
// Post: ...
void reorganitza_parells_senars(vector<int>& v) {
  int i = 0; int j = 0;
  while (j < v.size()) {
    if (v[j] % 2 == 0) {
      swap(v[i], v[j]); ++i;
    }
    ++j;
  }
}
```



Invariants “gràfics”

```
// Pre: v = V
// Post: \ldots
void reorganitza_parells_senars(vector<int>& v) {
  int i = 0;
  for (int j = 0; j < v.size(); ++j)
    if (v[j] % 2 == 0) {
      swap(v[i], v[j]); ++i;
    }
}
```



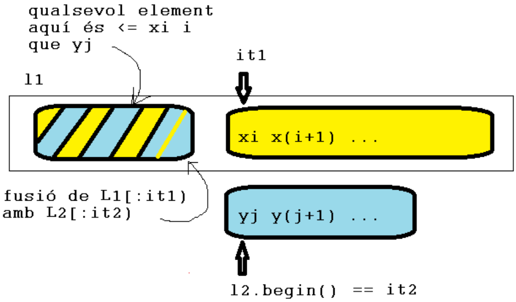
Invariants “gràfics”

```
// Pre:  $l1 = L1 = [x_1, \ldots, x_m] \wedge l2 = L2 = [y_1, \ldots, y_n] \wedge$
//       $x_1 \leq x_2 \leq \cdots \leq x_m \wedge y_1 \leq y_2 \leq \cdots \leq y_n$
// Post: $l1 = [z_1, \ldots, z_{m+n}] \wedge l2 = []$
//       $\wedge \{z_1, \ldots, z_{m+n}\} = \{x_1, \ldots, x_m\} \cup \{y_1, \ldots, y_n\}$
//       $\wedge z_1 \leq z_2 \leq \cdots \leq z_{m+n}$
template <typename T>
void fusionar(list<T>& l1, list<T>& l2);
```

N.B. Suposem que hi ha un ordre $\leq$ definit entre elements de tipus `T`

Invariants "gràfics"

- Fita: $\min(\text{mida de } l1[it1:), l2.\texttt{size}())$
- Invariant:



Invariants “gràfics”

```
template <typename T>
void fusionar(list<T>& l1, list<T>& l2) {
  list<T>::iterator it1 = l1.begin();
  list<T>::iterator it2 = l2.begin();
  while (it1 != l1.end() and it2 != l2.end()) {
    if (*it1 <= *it2) ++it1;
    else {
      l1.insert(*it2);
      it2 = l2.erase(it2);
    }
  }
  ...
}
```

Invariants "gràfics"

```
template <typename T>
void fusionar(list<T>& l1, list<T>& l2) {
  list<T>::iterator it1 = l1.begin();
  list<T>::iterator it2 = l2.begin();
  while (it1 != l1.end() and it2 != l2.end()) { ... }
  // it1 = l1.end() i l1[:it1) és la fusió de L1 amb L2[:it2)
  // o it2 = l2.end() i l1[:it1) és la fusió de L1[:it1) amb L2

  // it1 = l1.end() i l1 és la fusió de L1 amb L2[:it2)
  // o it2 = l2.end() i l1 és la fusió de L1 amb L2
  l1.splice(l1.end(), l2);
}
```