Correctesa de programes iteratius

R. Ferrer i Cancho

Universitat Politècnica de Catalunya

PRO2 (curs 2017-2018)
Versió 0.5

Avís: aquesta presentació no pretén ser un substitut dels apunts oficials de l'assignatura.

On som?

- Tema 4: Correctesa de programes iteratius
- 5a sessió

Avui

- Què és un programa correcte/incorrecte?
- Com es demostra la correctesa d'un programa?
- Correctesa d'algorismes (més que no pas de programes) iteratius.

El principi d'inducció: utilitat

- Motivació: com demostrar que un programa es totalment correcte amb un nombre finit de proves?
- Com demostrar que una propietat la satisfan un nombre infinit d'elements?
- PRO2: demostrar la correctesa de programes iteratius i recursius.
- En general: demostrar propietats dels nombres naturals.
- Dues versions del principi d'inducció.

El principi d'inducció: 1a versió

- $P(x)$: x satisfà la propietat P.
- Definició 1:
  $[P(0) \wedge \forall x \in \mathbb{N}\ (P(x) \implies P(x+1))] \implies \forall y \in \mathbb{N}\ P(y)$.
- Per demostrar $\forall y \in \mathbb{N}\ P(y)$ cal verificar primer
  1. Cas base: $P(0)$.
  2. Hipòtesi d'inducció: $x \in \mathbb{N}, P(x)$.
  3. Pas d'inducció: $P(x) \implies P(x+1)$.

El principi d'inducció: 2a versió

- Definició 2:
  $[P(0) \wedge \forall x \in \mathbb{N}\ (\forall y < x\ P(y) \implies P(x))] \implies \forall z \in \mathbb{N}\ P(z)$
- Per demostar $\forall z \in \mathbb{N}\ P(z)$ cal demostrar:
  1. Cas base: $P(0)$.
  2. Hipòtesi d'inducció: $\forall y < x\ P(y)$.
  3. Pas d'inducció: $\forall y < x\ P(y) \implies P(x)$.

Exemple aplicació

Demostrar: $\forall n \in \mathbb{N}, 1 + 2 + \cdots + (n-1) + n = n(n+1)/2$.

- Propietat: ?
- Cas base: ?
- Pas d'inducció: ?

- Com usar el principi d'inducció per demostrar correctesa d'algorismes iteratius i recursius?
- Quines serien la propietat P a demostrar en aquest cas?

Correctesa d'un bucle I

Esquema bàsic:

- ```
  // Pre: ...
  inicialitzacions
  while (B) {
    S
  }
  ...
  // Post: ...
  ```
  ```

Correctesa d'un bucle II

Demostrar la correctesa d'un bucle

a) Demostrar que si a l'inici de l'execució les variables satisfan la precondició, en acabar l'execució satisfan la postcondició.

b) Demostrar que l'execució acabarà.

Clau a): trobar l'invariant.

- Invariant: una propietat que es preserva a cada interació.
- Exemple: càlcul de l'alçada d'una pila:

```
/* Inv: n="nombre d'elements de la part tractada de P" y
p="elements de la pila inicial P que queda per tractar". */
```

- Exemple: invariant d'un bucle de cerca d'un element en un vector.
- Si es compleix la condició B aleshores és cert a l'inici i al final de cada iteració.
- Les inicialitzacions han de fer-lo cert.
- A la darrera iteració (quan B ja no sigui cert) ha d'implicar la postcondició.

Usar l'invariant per demostrar a)

Part a): si a l'inici de l'execució les variables satisfan la precondició, en acabar l'execució satisfan la postcondició.

a.1) Demostrar que les variables del programa en qualsevol iteració del bucle satisfan l'invariant.

- Per inducció sobre el número natural que indica el número de la iteració.
- Invariant cert a la iteració i-èssima: hipòtesi d'inducció.
- $\{Pre\} Inicialitzacions \implies \{Invariant\}$: base de la inducció (invariant cert a la iteració 0).
- $\{Invariant \wedge B\} S \implies \{Invariant\}$: pas d'inducció (invariant cert la iteració $i+1$).

a.2) $\{Invariant \wedge \neg B\} \implies \{Post\}$

b) acabament

Per què es penjen els programes?

- Evitar bucles infinits.
- Estratègia: trobar paràmetre o funció fita.
- Paràmetre o funció fita:
  - natural que decreix a cada iteració del bucle (distància a l'acabament).
  - ha d'acostar-nos a la condició d'acabament ($\neg B$).

Es important per garantir acabament que la funció fita sigui un natural: no creuarà mai el zero (garantia d'acabament)

Derivació de programes 1

Dues estratègies:

- Proposar un programa (estil PRO1/FP) i després demostrar-ne la correctesa.
- Derivar (generar un programa) a partir de l'especificació pre/post:
  Mètode en dues fases:
  1. Obtenir l'invariant
  2. Obtenir instruccions del bucle.

Derivació de programes 2

Passos detallats:

- Proposar invariant a partir de la postcondició
  Trucs:
  - L'invariant ha de descriure un punt intermig (qualsevol) de l'execució del bucle.
  - L'invariant com a afebliment de la postcondició (ex. post sobre la part tractada d'un seqüència)
  - Per expresar l'invariant cal introduir noves variables (variables locals en el codi.

Derivació de programes 3

Passos detallats (continuació):

- A partir de l'invariant hem de deduir l'estat B' en el que la execució ha acabat. $B = \neg B'$.
- Veure quines instruccions del cos de bucle son necessaries per
  - Preservar l'invariant.
  - Acostar-nos a l'acabament del bucle.
- Inferir instruccions o inicialitzacions que faran que es compleixi l'invariant abans d'entrar al bucle.
- Pot caldre afegir instruccions per obtenir la postcondició en sortir del bucle.

Comentaris generals

Restriccions al codi en C/C++ per tal de poder fer les justificacions de correctesa:

1. Nomes podrem fer iteracions amb la instrucció `while` (no `for`).
2. Les expresions booleanes no s'avaluen amb prioritat.
   - Si tenim "p and q", si p és fals, q també s'avalua.
   - Si tenim "p or q", si p és cert, q també s'avalua.
3. En el cas de funcions, només podem utilitzar un return, i aquest es col·locarà al final de tot.

Qualsevol altre codi que no compleixi alguna de les restriccions, pot ser transformat fàcilment en un de equivalent que si que les compleixi.

Esquema bàsic d'una justificació

- *Inicialitzacions*. Demostrar que les inicialitzacions fan cert l'invariant abans d'entrar al bucle.
- *Condició de sortida*.
  - Demostrar que dóna la Post a partir de l'invariant.
  - Demostrar que dóna la Post a partir de l'invariant + instruccions addicionals a la sortida del bucle.
- *Cos del bucle*. Demostrar que les instruccions garanteixen que l'invariant és cert després d'executar-ne les instruccions.
- *Acabament*. Funció fita (funció que decreix a cada interació).
- *Instruccions finals (sortida del bucle)*. Demostrar que porten a la post un cop el bucle ha acabat.
- *A més: precondicions*. Raonar que la *Pre* de totes les crides a operacions se satisfà en el punt de la crida.

Calcular l'alçada d'una pila

- Donada una pila d'enters, calcular el nombre d'elements.
- Especificació:

```
int altura_pila_int(stack<int> &p)
// Pre: p = P
// Post: El resultat és el nombre d'elements de P
```

Solució

```
int altura_pila_int(stack<int> &p)
// Pre: p = P
{
  int n = 0;

  // Inv: n="nombre d'elements de la part tractada de P" y
  //   p="elements de la pila inicial P que queda per tractar".

  while (not p.empty()){
    ++n;
    p.pop();
  }
  // A: n="nombre d'elements de P"
  return n;
}
// Post: El resultat és el nombre d'elements de P
```

Justificació informal

- *Inicialitzacions*. Inicialment considerem que no hi ha cap element de `P` tractat (`p` conté tots els elements). Amb `n = 0` satisfem l'invariant.
- *Condició de sortida*. Quan la pila es buida `n` equival al nombre d'elements de la pila i per tant la Post s'obté retornant `n`
- *Cos del bucle*. Amb `++n` i `p.pop()` recuperem l'invariant després d'una interació del bucle.
  La Prec de `p.pop()`) és certa gràcies a la condició del bucle.
- *Acabament*. Funció fita: mida de `p`.
- *Precondicions*. La de `p.pop()` està garantida per la condició del bucle.

Cerca en una cua

- Donada una cua d'enters i un enter, determinar si aquest apareix a la cua.
-
```
bool cercar_iter_cua_int(queue<int> &c, int i)
// Pre: c = C
{
  bool ret = false;

  // Inv: ret ="el element i esta a la part tractada de C" y
  //      c es la part de C que queda per tractar.

  while (not c.empty() and not ret){
    ret = (c.front() == i);
    c.pop();
  }
  return ret;
}
// Post: El resultat ens diu si i és un element de C o no
```

Justificació

- *Inicialitzacions*. Inicialment no hi ha cap element de `C` tractat (`c` conté tots els elements). Per tant `ret = false` fa cert l'invariant abans d'executar el bucle.
- *Condició de sortida*. Si `c` és buida o bé hem trobat l'element (`ret` és cert) aleshores ja tenim la Post.
- *Cos del bucle*. Amb les dues instruccions obtenim l'invariant al final de cada iteració (...).
  Gràcies a la condició del bucle, la Prec de `c.front()` i `c.pop()` se continuen satisfent.
- *Acabament*. Funció fita: mida de `c`.
- *Precondicions*. La de `c.front()` i `p.pop()` estan garantida per la condició del bucle.

Exemple amb llistes

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

```
void suma_llista_k(list<int> &l, int k)
// Pre: l = L
{
...
}
// Post: Cada element de l es la suma de k i l'element de L
// a la seva mateixa posició
```

Exemple amb llistes: solució proposada

```
void suma_llista_k(list<int> &l, int k)
// Pre: l = L
{
  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 posició
```

Quin és l'invariant?

Proposta d'invariant

```
// Inv:
// - it va entre l.begin() i l.end() (ambdos inclosos).
// - Els elements de l de l'inici fins a la posicio anterior
// a la que marca el it son la suma de k mes el element corresponent
// de L.
// - A partir de it fins al final de l, els elements son els
// mateixos que a L.
```

Exemple amb llistes: justificació

- *Inicialitzacions*. Després d'executar it = l.begin() se satisfà l'invariant.
- *Condició de sortida*. Un cop hem arribat al sentinella de la llista (`it ==l.end()`), l'invariant indica que ja tenim la postcondició.
- *Cos del bucle*.
  - Suposarem que a l'inici d'una iteració qualsevol del bucle es compleixen l'invariant i la condició d'entrada en el bucle.
  - Després d'executar les instruccions del bucle es torna a fer cert l'invariant.
- *Acabament*. Fita: mida de la subllista entre l'iterador i `l.end()`.
- *Precondicions*. Compte amb `*it += k;`

Percentatge d'estudiants presentats (amb nota)

- Donat un vector d'estudiants retornem el percentatge d'estudiants presentats (amb nota) del vector.
- Especificació:

```
double presentats(const vector<Estudiant> &vest)
// Pre: vest conté almenys un element
...
// A: n és el nombre d'estudiants presentants de vest
...
// Post: el resultat és el percentatge de presentats de vest
```

Derivació de la implementació. Pas 1: l'invariant

- Percentatge de presentats necessitem implica saber el nombre d'estudiants amb nota.
- Recorrer el vector comptant els estudiants amb nota.
- Invariant:

```
I: 0 <= i <= vest.size(), n="nombre d'estudiants amb nota a vest[0..i-1]"
```

Esquema bàsic d'una justificació/derivació

- *Inicialitzacions*. Quines instruccions afegir perquè l'invariant sigui cert abans d'entrar al bucle.
- *Condició de sortida*.
  - Quina condició ens dóna la Post a partir de l'invariant?
  - Quina condició ens dóna la Post a partir de l'invariant + instruccions addicionals?
- *Cos del bucle*. Com avançar? Quines instruccions cal per garantir que l'invariant es preserva després d'avançar?
- *Acabament*. Què decreix a mesura que anem iterant?
- *Instruccions finals*. Instruccions addicionals per acabar satisfent la post un cop el bucle ha acabat.

Derivació de la implementació. Pas 2: instruccions del bucle I

- *Inicialitzacions*.
  - Inicialment: cap estudiant tractat, és a dir, `i = 0`.
  - Conseqüència: `n = 0`, ja que no hi ha cap estudiant fins a `i-1` amb nota.
- *Condició de sortida*.
  - Esquema de recorregut: sortirem del bucle quan `i = vest.size()` (romandre mentre `i < vest.size()`)
  - A l'invariant `i <= vest.size()`.
- *Cos del bucle*.
  - Cal incrementar `i` per avançar en el recorregut.
  - Cal afegir instruccions per preservar l'invariant després d'haver incrementat `i`.
  - La condició del bucle ens garanteix `i + 1 <= vest.size()`.

Derivació de la implementació. Pas 2: instruccions del bucle II

- *Acabament*.
  - Funció fita: `vest.size() - i`. (distància a l'acabament): `vest.size() - i`.
- *Instruccions finals*.
  - Quan sortim del bucle tenim a `n` el "nombre d'estudiants amb nota en `vest`".
  - Conseqüencia: només cal calcular el percentatge a partir de n.

El programa anotat

```
double presentats(const vector<Estudiant> &vest)
// Pre: vest conté almenys un element
{
  int numEst = vest.size();
  int n = i = 0;

  // Inv: 0 <= i <= numEst, n = nombre d'estudiants amb nota a vest[0..i-1]

  while (i < numEst){
    if (vest[i].te_nota()) ++n;
    // n és el nombre d'estudiants amb nota de vest
    ++i;
  }
  // A: n és el nombre d'estudiants presentants de vest
  double pres = n*100./numEst;
  return pres;
}
// Post: el resultat és el percentatge de presentats de vest
```

Arrodoniment de la nota

- Donat un vector d'estudiants, modificar-lo arrodonint-ne les notes a la dècima més propera (es pot fer com a acció o com a funció).
- Especificació

```
void arrodonir_notes(vector<Estudiant> &vest);
// Pre: cert
// Post: vest té les notes dels estudiants arrodonides respecte
   al seu valor inicial
```

Derivació de la implementació. Pas 1: l'invariant

- Abstracció funcional: funció que arrodoneix un real (disseny descendent).
- Exercici: invariant.

Invariant

- Pista: postcondició aplicada a la part tractada del vector.
- `vest[0..i-1]` té les notes dels estudiants arrodonides respecte al seu valor inicial, `0 <= i <= vest.size()`

Derivació de la implementació. Pas 2: instruccions del bucle

- *Inicialitzacions*.
  Inicialment no hi ha cap estudiant del vector tractat: `i = 0`.
- *Condició de sortida*.
  Esquema de recorregut: sortirem del bucle quan `i = vest.size()` (romandre mentre `i < vest.size()`).
- *Cos del bucle*.
  - Avançar en el recorregut incrementant la `i`.
  - Abans d'incrementar la `i`, hem d'assegurar-nos que l'invariant se satisfaci despres d'incrementar-la.
- *Acabament*. A cada volta decreix la distància entre `vest.size()` i `i`.

Implementació: acció principal

```
void arrodonir_notes(vector<Estudiant> &vest)
// Pre: cert
{
  int numEst = vest.size();
  int i = 0;
  // Inv: vest[0..i-1] té les notes dels estudiants arrodonides
  //      respecte al seu valor inicial, 0 <= i <= numEst
  while (i < numEst) {
    if (vest[i].te_nota()) { // mirem si vest[i] té nota
      // obtenim la nota de vest[i] arrodonida
      double aux = arrodonir(vest[i].consultar_nota());
      // modifiquem la nota de vest[i] amb la nota arrodonida
      vest[i].modificar_nota(aux);
    }
    ++i;
  }
}
// Post: vest té les notes dels estudiants arrodonides respecte
// al seu valor inicial
```

Implementació de la funció `arrodonir`

Assumim que hi ha una funció `int(...)` que retorna la part entera d'un real.

```
double arrodonir(double r)
// Pre: cert
{
  return int(10.*(r + 0.05))/10.0;
}
// Post: el resultat és el valor original de r arrodonit a
// la dècima més propera (i més gran si hi ha empat)
```