Verifica – parte IIIB

Rif. Ghezzi et al.
6.4.2

università di ferrara
DA SEICENTO ANNI GUARDIAMO AVANTI.

Prove formali di correttezza

- Presuppongono specifica delle postcondizioni e precondizioni di un (frammento di) programma (ad esempio con asserzioni)
- Dimostrazione matematica che la postcondizione discende dalla precondizione e dal programma

Esempio

```
{true}
1 begin
2     read (a); read (b);
3     x := a + b;
4     write (x);
5 end
{output = input1 + input2}
```

- Postcondizione vera se, prima di 4,
- x = input1 + input2, vero se, prima di 3,
- a + b = input1 + input 2, vera per la semantica di read.

Backward substitution

- Procedimento che costruisce una catena di implicazioni inverse dalla postcondizione alla precondizione
- Data un'istruzione

  x := exp,

  una sua postcondizione Post vale dopo l'esecuzione se, prima dell'esecuzione, valeva l'espressione ottenuta sostituendo in Post exp a ogni occorrenza di x.

Esempio

{x = 5}
x:= x + 1
{x = 6}

- La postcondizione vale dopo l'esecuzione se, prima dell'esecuzione, valeva x+1=6, cioè x =5 (precondizione)

Esempio

{a + b - 43 > y + 7}
z := a + b; x := z - 43
{x > y + 7}
Con backward substitution

- {z – 43 > y + 7} (postcond. di z:=a+b)
- {a + b - 43 > y + 7} (precondizione)

Input-output

- input e output come file (sequenze di valori)
- read(x) trattata come assegnamento da input ad x
- write(x) trattata come assegnamento da x ad output

Esempio

{true}
read(x); x := x + 1; write(x)
{output = input + 1}
Con backward substitution:

- x = input + 1
- x + 1 = input + 1
- input + 1 = input + 1

Notazione di Hoare

- Dalla verità di Claim1 e Claim2 si può dedurre Claim3

$$\frac{\text{Claim1, Claim2}}{\text{Claim3}}$$

- Utile per esprimere le relazioni fra precondizioni e postcondizioni delle strutture di controllo.
- Permette di costruire prove complesse a partire da prove più semplici

Sequenza

$$\frac{\{F1\}S1\{F2\},\ \{F2\}S2\{F3\}}{\{F1\}S1;S2\{F3\}}$$

Se F1 e l'esecuzione di S1 implicano F2, e se F2 e l'esecuzione di S2 implicano F3, F1 e l'esecuzione successiva di S1 e S2 implicheranno S3.

- Esempio: da
  - {a+b-43>y+7} z:=a+b {z-43>y+7}, e
  - {z-43>y+7} x:=z-43 {x>y+7}, si deduce
  - {a+b-43>y+7} z:=a+b; x:=z–43 {x>y+7}

Istruzione condizionale

$$\frac{\{Pre \text{ and } cond\}\ S1\ \{Post\},\{Pre \text{ and not } cond\}\ S2\ \{Post\}}{\{Pre\}\ \text{if cond then S1 ; else S2 ; end if;}\ \{Post\}}$$

- Se la verità di Post è garantita da:
  - Pre, la verità di cond e l'esecuzione di S1 (ramo then)
  - Pre, la falsità di cond e l'esecuzione di S2 (ramo else)

  Post è garantita da Pre e dall'esecuzione dell'istruzione condizionale

Esempio

```
{true}
if x >= y then
   max := x
else
   max := y
end if
{(max=x or max=y) and (max>=x and
   max>=y)}
```

Analisi con backward substitution

- Ramo then:
  - (x=x or x=y) and (x>=x and x>=y)
  - equivalente a (x>=y), implicato da Pre and cond
- Ramo else:
  - (y=x or y=y) and (y>=x and y>=y)
  - equivalente a y>x, implicato da Pre and not cond

Ciclo while

$$\frac{\{I \text{ and } cond\}\ S\ \{I\}}{\{I\}\ \text{while cond loop S; end loop;}\ \{I \text{ and not } cond\}}$$

- S corpo del ciclo
- I è invariante se I, la verità di cond e l'esecuzione di S mantengono vero I.
- Se I è un invariante per il ciclo,

  allora la verità di I all'ingresso nel ciclo while e l'esecuzione del ciclo implicano I e la falsità di cond all'(eventuale) uscita dal ciclo

Esempio

{x >= 0}
while (x>0) loop
  x:=x-1;
end loop
{x = 0}

I: {x >= 0} è invariante di ciclo: per backward substitution, x >=0 se x-1>=0, implicata da I and cond essendo x intero.

All'uscita dal ciclo, valgono I and not cond, cioè x >=0 and x <=0, cioè x = 0

Esempio

{x>y}
while x != 0 loop
x:=x-2
y:=y-2
end loop
{x>y and x=0}

x > y è un invariante, ma il ciclo non termina se x è inizialmente dispari o minore di 0.

Terminazione

- Una prova di correttezza totale richiede anche la prova della terminazione del (frammento del) programma.
- La sola dimostrazione della postcondizione a partire dalla precondizione e dall'esecuzione del programma è una prova di correttezza parziale

Scelta dell'invariante di ciclo

- Non esistono regole per individuarlo; bisogna esaminare le istruzioni del ciclo
- L'invariante deve essere abbastanza forte da implicare la postcondizione del ciclo
- Deve però essere abbastanza debole da valere all'ingresso del ciclo

Esempio: divisione intera

```
{input1 > 0 and input2 > 0}
begin
    read (x); read (y);
    div := 0;
    while x>=y loop
        div := div + 1;
        x := x - y;
    end loop;
    write (div); write (x);
end;
{input1 = output1 * input2 + output2 and
0 <= output2 < input2 }
```


Backward substitution

```
{input1 > 0 and input2 > 0}
begin
    read (x); read (y); div := 0;
    while x>=y loop
        div := div + 1;
        x := x - y;
    end loop;
end;
{input1 = div * input2 + x and
0 <= x < input2 }
```

- Invarianti inutili: x >= 0, div >=0, div > 1000

Scelta di un invariante

- x vale inizialmente input1 e viene decrementato di y a ogni iterazione
- div vale inizialmente 0 e viene incrementato di 1 a ogni iterazione
- Possibilità:

  input1 = x + div*y
- E' invariante e vale all'ingresso nel ciclo, ma è troppo debole

Scelta di un invariante

input1 = x + div*y and x >= 0 and y = input2

- E' invariante: con due backward substitution,

  input1 = x – y + (div+1) * y and x - y >= 0 and y = input2, implicato da I e cond
- E' abbastanza forte: implica la postcondizione
- Vale all'entrata del ciclo: con backward substitution,

  input1 = input1 + 0*input2 and input1>=0 and input2 = input2

Programmi con array

- L'asserzione

{a(i) = 3}

a(i) := 4

{a(3) = 2 and a(i) = 4}

- è falsa se al momento dell'assegnamento i = 3
- Il problema nasce dalla dipendenza dell'istruzione dal valore dell'indice

Regola di sostituzione in presenza di array

- Dati
  - una formula Post
  - un assegnamento a(i) := expression,

  sia Pre l'asserzione che si ottiene da Post sostituendo ogni occorrenza di a(j) con if i = j then expression else a(j)
- Allora vale l'asserzione

  {Pre} a(i) := expression {Post}

Esempio

{(if i=3 then 4 else a(3)) = 2 and ((if i = i then 4 else a(i)) = 4}

a(i) := 4

{a(3) = 2 and a(i) = 4}

- Semplificando:

{i != 3 and a(3) = 2}

a(i):=4

{a(3) = 2 and a(i) = 4}

Esempio

- Inserimento di un elemento di una tabella: se la tabella non è piena, l'elemento deve essere inserito.

{n < nmax}

if n < nmax then

  n := n + 1;

  table(n) := x;

end if

{n <= nmax and (exists i (1 <= i <= n and table (i) = x))}

Esempio

Con backward substitution

{n+1 <= nmax and

exists i (1 <= i <=n+1 and (if n+1=i then x else table(i))=x)}

- n+1 <= nmax discende dalla precondizione
- exists i (1 <= i <=n+1 and (if n+1=i then x else table(i))=x) è soddisfatta da i = n+1

Esempio

```
{n>=1}
i := 1; j := 1;
found := false;
while i<=n loop
     if table (i) = x then
          found := true;
          i := i + 1
     else
          table (j) := table (i);
          i := i + 1; j := j + 1;
     end if;
end loop;
n := j - 1;
{not exists m (1<=m<=n and table (m) = x) and
found =exists m (1<=m<=old_n and old_table (m) = x)}
```

*old*_table, *old*_n: costanti che rappresentano i valori delle variabili prima dell'esecuzione.

Prova

- Invariante di ciclo I:

{(j<=i) and (i<=*old*_n + 1) and

(not exists m (1<=m<j and table (m) = x)) and

(n = *old*_n) and

found = exists m (1<=m < i and *old*_table(m) = x)}

- Insieme a i>n implica la postcondizione (n:= j -1)

Prova: invariante

- Backward substitution in ramo then:

{(j<=i+1) and (i+1<=*old*_n + 1) and

(not exists m (1<=m<j and table (m) = x)) and

(n = *old*_n) and

exists m (1<=m < i+1 and *old*_table(m) = x)}

- Implicato da I e i<=n e table(i) = x

Prova: invariante

- Backward substitution in ramo else

{(j+1<=i+1) and (i+1<=*old*_n + 1) and

(not exists m (1<=m<j+1 and

(if m=j then table(i) else table (m)) = x)) and (n = *old*_n) and

found = exists m (1<=m < i+1 and *old*_table(m) = x)}

- Terza clausola: per 1<=m<j implicata dall'invariante, per m=j dalla condizione dell'if
- Ultima clausola: found può cambiare di valore solo se table(i) = x, escluso dalla condizione dell'if.

Prova

- Backward substitution su I:

{(1<=1) and (1<=*old*_n + 1) and

(not exists m (1<=m<1 and table (m) = x))

and (n = *old*_n)

and false = exists m (1<=m < 1 and *old*_table(m) = x)}

- Implicato dalla precondizione

Prove di correttezza in grande

- Le prove formali di correttezza sono complesse anche per frammenti brevi di codice.
- Dimostrare formalmente la correttezza di sistemi di grandi dimensioni è spesso impraticabile
- Si possono comunque modularizzare le prove di correttezza.

Prove di correttezza in grande

- Dimostrazione di proprietà di un modulo in due fasi:

1. Dimostrazione di proprietà del modulo a partire dalla specifica formale delle operazioni
2. Dimostrazione della correttezza dell'implementazione delle operazioni rispetto alla loro specifica

Esempio

```
module TABLE;
exports
    type Table_Type (max_size: NATURAL): ?;
    la tabella non può memorizzare più di max_size
    elementi
    procedure Insert (Table: in out TableType ;
        ELEMENT: in ElementType);
    procedure Delete (Table: in out TableType;
        ELEMENT: in ElementType);
    function Size (Table: in Table_Type) return NATURAL;
    fornisce la dimensione attuale della tabella
    …
end TABLE
```


Specifica di operazioni

{true}
Delete (Table, Element);
{Element ∉ Table};

{Size (Table) < max_size}
Insert (Table, Element)
{Element ∈ Table};

- Si dimostra banalmente che l'inserimento (in tabella non piena) e l'eliminazione successiva di un elemento E implica che E non appartenga alla tabella

Prove di correttezza in pratica

- Proposte fin dagli anni Settanta
- Tuttora poco applicate
- Obiezioni più comuni:
  1. Sono spesso più complesse dei frammenti di codice di cui dimostrano la correttezza, e quindi ancora più soggette ad errore
  2. Richiedono conoscenze matematiche fuori della portata di un normale programmatore

Prove di correttezza in pratica

1. Risultati simili si possono ottenere a costo inferiore con analisi informali
2. Anche un sistema dimostrato corretto è soggetto a fallimenti dovuti all'hardware, alla piattaforma, agli utenti, etc.
3. Le prove formali spesso non considerano che l'implementazione dei tipi è un'approssimazione dei concetti matematici corrispondenti

Prove di correttezza in pratica

- Vantaggi:
  1. Portano a un'analisi approfondita del codice
  2. Il ragionamento formale, anche se soggetto a errore, è comunque il più affidabile
  3. Aumentano la fiducia nella bontà di un sistema anche se non lo riguardano nella sua interezza

Prove di correttezza in pratica

- Decidere criticamente quando investire in formalità:
  - Porzioni critiche di codice (es. funzione di libreria utilizzata da molti client)
  - Asserzioni intermedie (es.: un indice non uscirà mai dall'intervallo corretto) anziché intera specifica
- Strumenti (semi-)automatici