Verifica formale di correttezza di programmi

A.Gargantini
Infomatica III
Unibg 2009/10

While programs

- I nostri programmi saranno composti dalle seguenti istruzioni
  - Assignments y := t
  - Composition S1; S2
  - If-then-else if e then S1 else S2 fi
  - While while e do S od
- Variabili intere

Cosa vuol dire provare che un programma è corretto

Proveremo

$$\{P\}S\{Q\}$$

- Detta tripla di Hoare
- Se P è vero prima di eseguire il programma S, allora dopo varrà Q
  - P: precondizione – vale prima dell'esecuzione
  - Q: postcondizione – vale alla fine dell'esecuzione
  - S: programma o frammento di programma
- P e Q vanno pensate e scritte

Asserzioni

- P e Q sono asserzioni
- In genere metteremo le asserzioni tra {}
- Scritte in logica proposizionale
  - Con l'uso di variabili che saranno anche nel programma
  - /\ per AND
    - Esempio   { x>0 /\ y = 3}
  - \/ per OR
  - ¬ per NOT
  - -> per implica : A -> B : A implica B, cioè A è più forte di B, es: x =2 -> x >0

Logica - breve

- per provare (1) A -> B
- posso ipotizzare A e poi con una serie di regole che mi fornisce la logica posso derivare una lista di proposizioni vere ... se alla fine derivo B, ho provato (1)
- Alcune regole dalla logica
  - Esempio se ho provato C and D, posso eliminare uno dei due
  - Se ho provato X e ho anche X -> Y, posso provare Y (modus ponens)
- Alcune potrebbero essermi date dalla logica particolare che considero come **ASSIOMI**
  - Esempio: se “piove -> bagnato”
- Alcune sono regole matematiche
  - $x > 10$ implica $x > 0$

Esemio: Greatest common divisor GCD

```
{x1>0 /\ x2>0}
y1:=x1;
y2:=x2;
while ¬(y1=y2) do
     if y1>y2 then y1:=y1-y2
                   else y2:=y2-y1 fi
od
{y1=gcd(x1,x2)}
```

Esempio 2

- calcolo del massimo tra due numeri:

{true}

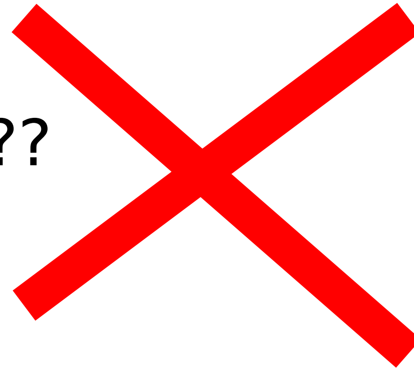
if x> y then max := x
else max:=y

// max è il massimo
$\{max \geq x \wedge max \geq y \wedge (max = x \vee max = y)\}$

Swap

- Se ho un programma che scambia due variabili x e y, come posso dire che alla fine sono scambiate?

- {}SWAP{x = y; y = x} ??



Come riferirsi al valore iniziale?

- Le post condizioni si riferiscono al valore **finale** delle variabili
  - es. x > 2 y   x e y come valgono alla fine
- Come faccio a riferirmi al valore che avevano all'inizio, ad esempio voglio dire che x varrà il doppio?
  - x = 2 * x'   (x' il valore iniziale)
- **Soluzione**
  - Aggiungo una istruzione del tipo x_old = x all'inizio e scrivo la post condizione utilizzando x_old
    - Es: x = x_old *2

Come fare a dimostrare {P}S{Q}

- Divideremo la dimostrazioni in piccoli passi
- ad ogni passo applichiamo una regola (assioma) che ci permette di ridurre la dimostrazione ad un caso più semplice

Con updates

- Useremo una variante della logica di Hoare
- Si chiama Hoare “Logic with Updates”
- e un tool: Key -hoare

Hoare Logic with Updates

- We will use a non-standard formulation of Hoare logic called “*Hoare Logic with Updates*".
  - With this formulation one more thing is included in the Hoare triple, namely an update which corresponds to the changes to the program state that have occurred “so far".
    - The update which does not contain any substitutions is denoted by [].
  - Atomic updates:
    - Loc:= term
      - Es. [x:= 5]
  - Diverse updates
    - in sequenza “,”
      - Es. [x:= 5, y:= x+1]
    - Posso avere anche più updates <u>in parallelo “|”</u>

      (dovranno essere consistenti)

Updates prima di un programma

- Davanti ad un programma mettiamo degli update che aggiornano lo stato a prima dell'esecuzione del programma:
  - [U]P
  - Applica gli update U ed esegui P
    - A normal Hoare triple {P}S{Q}, can easily be translated to the corresponding Hoare triple with updates, namely {P}[]S{Q}.
  - {P}[U]S{Q}
    - Correttezza di S: a partire da P, applica gli updates U, esegui S, vale Q

Rules

The inference rules describe a straight-forward strategy for dividing the program into smaller pieces until only plain logical formulas are left.

Each rule has the form

```
                  P1, P2, ... Pn
Rulename ------------------------------
                  {P}[U] S{Q}
```

You always apply the rule by matching the Hoare triple below the line to the current Hoare triple. The rule turns your current problem into n new problems, each being either a Hoare triple or a logical formula.

Empty program

- Caso più semplice ho un programma vuoto:
- {P}[U]{Q}
- Come faccio a provarlo?

Exit rule

- We then have a Hoare triple of the form

    {P}[U]{Q}

In order to turn this into a logical formula we must have a rule that performs the substitutions recorded in U

$$\text{exit} \quad \frac{P \rightarrow U(Q)}{\{P\}[U]\{Q\}}$$

U(Q):apply the updates in U in Q

Updates

- Updates in U collect the assignments
  - We write single substitutions on the form x := e.
  - Multiple substitutions are denoted as follows:

    [x1 := e1, x2 := e2, ..., xn := en]
- Update applicata alla condizione: significato
  - [x1 := e1, x2 := e2, ..., xn := en](P)
  - is the formula you get by first substituting xn by en in P, then x(n-1) by e(n-1) and so forth until the first substitution, x1 := e1 has been performed
  - [x1 := e1, x2 := e2, ..., xn := en](P)
  - > [x1 := e1, x2 := e2, ...,](P[xn<-en])
    - P[xn<-en]: P con en al posto di xn

Applicazione update

- Se applico un update ad una asserzione:

  $[x := t](P)$

  $\rightarrow$

- Sostitusco x con t:

  $P\ [x \leftarrow t]$

  - Nota: t potrebbe contenere x

Example

[x:=x+1, x := x * 3, x := x - 5] x = 13

- yields

[x := x + 1, x := x * 3] x - 5 = 13

[x := x + 1] x * 3 - 5 = 13

(x + 1)* 3 - 5 = 13

- Simplify

x = 5

Assignment Rule

- Let's start with the rule for assignment statements. Since we are using updates, this is really simple.

```
                    {P}[U, x:=e] s {Q}
Assignment  --------------------------------
                    {P}[U] x:=e; s {Q}
```

- The rule says: given a Hoare triple with update, if the program starts with an assignment, then turn it into a substitution and put it at the end of the update.

Rules

- All the rules (except exit) are applied at the first statement of the program

{P1}[U1]S2{Q}, ...{Pn}[Un]S2{Q}

Rulename ---

{P}[U] S1; S2 {Q}

Esempio

{x = 5}[] x := x + 1; x := 2*x;{x = 12}

- Apply the assignment rule

- {x=5}[x:=x+1] x:=2*x;{x=12}

- {x=5}[x:=x+1,x := 2*x]{x=12}

Exit rule

- With a program consisting only of assignments, we can apply the assignment rule repeatedly until no statements remain. We then have a Hoare triple of the form

  {P}[U]{Q}

In order to turn this into a logical formula we must have a rule that performs the substitutions recorded in U

$$\text{exit}\quad \frac{P \rightarrow U(Q)}{\{P\}[U]\{Q\}}$$

U(Q):apply the updates in U in Q

Esempio 1

- Per provare

  $\{x=5\}[x:=x+1, x := 2*x]\{x=12\}$
- Applico exit

  $x = 5 \rightarrow [x:=x+1, x := 2*x](x=12)$
- Applico gli update

  $x = 5 \rightarrow [x:=x+1](2*x=12)$

  $x = 5 \rightarrow [](2*(x+1)=12)$
- Vero !

Implicazione

- Alcune volte P → U(Q) perchè P è +più forte di P(Q), cioè implica
- Esempio
- {x>0} [x := x+1]{x>0}
- {x>0} → {x+1>0} ? cioè
- {x>0} → {x>-1} OK

Example

{x = 5}

[]

x = x + 1;

x = x * 3;

x = x - 5;

{x > 12}

- Apply the assignment rule
- ....
- {x=5}[]
- → nel tool

Key-hoare

- In KeY-Hoare you have to manually apply the inference rules (assignment, conditional, loop and exit). To do that you press the right button when everything but the precondition of the Hoare triple is highlighed. Sometimes you may also have to apply `remove block' when you have a empty block at the beginning of the program. Once you have chosen `exit' you have a logical formula. Logical formulas can be proved automatically by choosing "Apply rules automatically here". We recommend that you do this rather than manually proving formulas. You can choose automatic as soon as the Hoare triple is gone

Swap example in Key_hoare

- In order to do the verication in the tool, we have to declare the variables.

```
\functions{
   int x0;
   int y0;
}
\programVariables{
   int x,y, d;
}

\hoare{ { x = x0&y = y0 }
\[{  d = x;
     x = y;
     y = d; }\]
{ x = y0&y = x0 } }
```

```
\functions{
   ....
}


\programVariables{
   int x,y, d;
}

\hoare{
{ .... }
[ ... ]
\[{   ....   }\]
{...}
}
```

Fixed values (inputs)

Variables used by the program

Hare triple: precondition,
Updates (usually to the inputs)
Program
posconditions

Conditional

The conditional rule is applied when the first statement in the program is an if-statement.

```
                      {P & U(b)}[U]s1;s{Q},  {P & not U(b)}[U]s2;s {Q}
Conditional     -----------------------------------------------------------------
                              {P}[U] if (b) s1 else s2;  s {Q}
```

The rule says that if the program starts with an if-statement, then proving the Hoare triple amounts to proving it for the program that corresponds to the then part and the program that corresponds to the else-part.

We get two new proof obligations since there are two possible paths that the program is executed, and our specification must hold in both cases. The rest of the program, s, is appended to both the then and the else-part. This is natural since in both cases the execution will continue after the if-statement.

Esempio MAX

- Nel tool

While cycles

A first example

Let's look at the simplest possible example where we don't know how many times the loop will be repeated.

```
{ n >= 0 }
   while (n > 0) {
   n = n - 1;
}
{ n = 0 }
```

While example

```
{ n >= 0 }
while (n > 0) {
n = n - 1;
}
{ n = 0 }
```

- After the execution of the while loop has finished we know that n <= 0 (since the condition n > 0 evaluated to false).
- But that is not enough to conclude the postcondition, n = 0.
- Looking at the precondition, n >= 0, we see that if we also had this after the loop, then we could conclude the postcondition

(n <= 0 & n >= 0) implies n = 0.

- In order to have this, we need to show that if n >= 0 holds before the loop, then the same thing also holds after it.
- We can do this by showing that if n >= 0 holds before executing the loop body, it also holds after the loop body.

While example

```
{n >= 0 }
while (n > 0) {
n = n - 1;
}
{ n = 0 }
```

- We say that the constraint n >= 0 should be preserved when executing the loop body.
- If it does we call it a loop invariant(it doesn't vary between each repetition of the loop).
- Once we've shown that the constraint is preserved by the loop we can use arguments of induction to conclude that regardless of the number of times the loop is repeated, the loop invariant will hold after the loop if it held before it.

While statement

- Elements of proving a loop correct

1. Come up with some constraint, I , which is hopefully a loop invariant.

2. Show that I holds before the loop, ie. I is initially valid.

3. Show that I is preserved by the loop, ie. if it holds before an execution of the loop body, then it holds after. When showing this can also assume the loop condition to be true (otherwise the loop would have stopped).

invariants

4. Show that if I is valid after the loop, then after executing the rest of the program, the postcondition will be valid.

   – In other words, when proving that the rest of the program is correct we can use the invariant. Apart from using the invariant, we can also assume the loop condition to be false right after the loop (otherwise the loop would have continued).

Loop rule

P → U(I), {I and b}S{I}, {I and not b}[]s2{Q}

loop rule ---

{P}[U]while b do S od s2{Q}

I: invariant

The rule creates three new proof obligations. As we have seen, this rule differs from the others in that it requires you to invent something new, namely the loop invariant I . I doesn't appear anywhere below the inference line, so it cannot be directly read o the current Hoare triple.

Loop rule

P → U(I), {I and b}S{I}, {I and not b}[]s2{Q}

loop rule ---

{P}[U]while b do S od s2{Q}

- The new proof obligations are (names refer to what appears in the KeY-Hoare proof tree):

  P → U(I)

- Invariant Initially Valid - The invariant must hold at the beginning of the loop. This means it must be a consequence of P, hence the implication. Just as in the exit rule, the eects of the update, U, must be reflected in the consequent.

{I and b}S{I}

- Preserves Invariant - If the invariant holds before executing the loop body, it should hold afterwards. The Hoare triple has an empty update, corresponding to the fact the U was already applied before the loop, in the “Invariant Initially Valid". The changes to the program state before the loop are already reflected in I . Just as we could assume that the guard evaluated to true in the then-branch of the conditional statement, we can here add the same assumption when entering the loop body.

{I and not b}[]s2{Q}

- Use invariant – By inductive reasoning we can now assume that I holds after the loop. And we also have that b is false (loop exit). These two thing form the precondition in the Hoare triple for the rest of the program

A first example

```
{n >= 0 }
while (n > 0) {
n = n - 1;
}
{ n = 0 }
```

Invariant: n>=0 !!!

Proviamo con il tool

Esempio: divisione intera

```
{x>=0 /\ y>=0}
a:=0;
b:=x;
while b>=y do
   b:=b-y;
   a:=a+1
od.
{x=a*y+b /\ 0<=b /\ b<y}
```

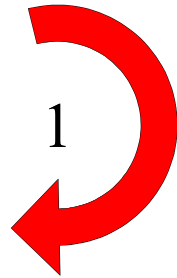
Invariante

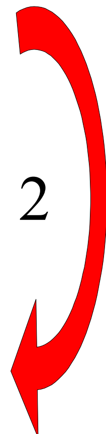
x=a*y+b /\ b>=0

calcola a = x/y e b il resto (x mod y)

Dimostrazione

```
{x>=0 /\ y>=0}
a:=0;
b:=x;
{I}
while b>=y do
   b:=b-y;
   a:=a+1
od.
{x=a*y+b /\ 0<=b /\ b<y}
```





divido in due
la dimostrazione

1 applico assegnamento

parte 2 : ciclo while

- I ha le proprietà degli invarianti?

1 {I /\ not B} -> {Q}:

l'invariante all'uscita del ciclo implica la postcondizione

2 {I /\ B } S {I} I è effettivamente invariante

3 P -> I : I è valido all'inizio del ciclo OK

Dimostro proprietà 1

{I /\ not B} -> {x=a*y+b /\ 0<=b /\ b<y}

{x = a*y+b /\ b>=0 /\ not b>=y} -> {x=a*y+b /\ 0<=b /\ b<y}

ok perchè not b >= y -> b<y

manca 2: {I /\ B } S {I}

{I /\ B } b:=b-y; a:=a+1{I}

Come “intuire” l'invariante

- L'invariante deve implicare all'uscita la postcondizione
  - Domandati perchè alla fine la post condizione vale?
- Prova a percorrere il ciclo con qualche caso di test, con un “giro” del ciclo, con due e così via

Come trovare invarianti

Using the experience from the previous examples we can sketch on a general plan for how to prove a loop correct:

- Look at the postcondition to see how it could be generalized.
- Perhaps dry-run the loop a few times to see a pattern of what is being preserved.
- From this choose a first attempt of the loop invariant.
- Try to prove that it's initially valid, preserved and entails the postcondition.
- While checking invariant preserved and use invariant you might encounter necessary extra conditions which you add to the invariant.
- Make sure that the new conditions are also initially valid and preserved.
- Repeat this until the whole proof goes through.

Esercizi

```
{y >= 0}
i = 0;
{I}
quad = 0;
while ( i != y) {
    quad = quad + y;
    i = i + 1;
}
{quad = y^2}

I = q = i * y
```

anche in questo caso divido la dimostrazione in due parti

questo I è assegnato

Altri esempi

- Esempio 1:somma

```
{n>0}
count = 0;
sum = 0;
while count < n  do
   count = count + 1;
   sum = sum + count;
end
{sum = 1 +2 + ... + n}
```

count cresce fino a diventare n, e sum accumula la somma da 1 a count:

I: sum = 1 + … + count

- Esempio 2: assegna

```
{x>=0}
y:= 0;
while (y < x) {
   y:= y+1;
}
{y=x}
```

Esempio 3

```
int old_x = x;
int dop = x;
while (x != 0) {
dop := dop +1 ; x := x-1;
}
{dop = 2 *old_x}
```

Soundness - Consistenza

la logica di Hoare è consistente (sound) nel senso che ogni cosa che può essere provata è corretta (cioè è vera)

Dim.: ogni assioma conserva la verità delle asserzioni (consistente).

Completezza

Un sistema di dimostrazioni è detto completo se ogni asserzioni vera può essere dimostrata

- la logica proposizionale è completa
- Per l'aritemtica invece non c'è sistema di regole che sia completo (Godel).

- puoi leggere “Gödel, Escher, Bach: An Eternal Golden Braid”, in italiano: Godel, Escher, Bach un’Eterna Ghirlanda Brillante di Douglas R. Hofstadter http://en.wikipedia.org/wiki/G%C3%B6del,_Escher,_Bach

E la logica di Hoare?

Due problemi:

1) incompletezza del calcolo aritemitico

2) problema della terminazione:

{p} S {false} significa che S non termina a partire dalle precondizioni p. Questo è indecidibile, cioè le regole non riescono a provarlo sempre:

il sistema però è relativamente completo