Università degli Studi di Bergamo
Facoltà di Ingegneria dell'Informazione

Informatica III

Elaborato 4: KeY-Hoare

Revisione	Data	Redatto
1.0	2006/02/16	Silvio Moioli

Indice dei contenuti

Storia delle revisioni

Rev. n	Data	Redatto	Descrizione
1.0	2006/02/06	Silvio Moioli	Versione iniziale.

Introduzione

Scopo del documento

Presentazione del quarto elaborato del corso sull'utilizzo della logica di Hoare con aggiornamento esplicito dello stato e dello strumento KeY-Hoare per la prova di correttezza dei programmi.

Definizioni, abbreviazioni e sigle

- BNF: Backus–Naur form;

Riferimenti bibliografici e Web

- http://www.key-project.org/, homepage del progetto KeY;
- http://en.wikipedia.org/wiki/KeY, pagina di Wikipedia su KeY;
- http://www.key-project.org/download/hoare/, pagina di Key-Hoare;
- http://www.key-project.org/download/hoare/students.pdf, spiegazione della logica di Hoare con aggiornamento esplicito dello stato.

Approccio seguito in questo elaborato

L'elaborato consiste nella presentazione della logica di Hoare con aggiornamento esplicito dello stato, dello strumento Key-Hoare e nella dimostrazione di correttezza con Key-Hoare di alcuni programmi visti a lezione. Per ogni programma è riportata la definizione nella logica di Hoare con aggiornamento esplicito dello stato e una traccia della sua dimostrazione.

Presentazione del programma

Logica di Hoare con aggiornamento esplicito dello stato

Introduzione, motivazioni all'uso

L'aspetto di maggiore importanza di KeY-Hoare è l'introduzione di un nuovo formalismo per la verifica di programmi, la logica di Hoare con aggiornamento esplicito dello stato, che è un'estensione della logica di Hoare classica.

Gli autori del sistema giustificano l'introduzione della nuova logica con diverse argomentazioni, tra cui le seguenti:

- la logica di Hoare, per come è definita, richiede che la dimostrazione sia condotta all'indietro in molti casi: anche se è teoricamente possibile ragionare in avanti questo risulta molto più difficile soprattutto in presenza di cicli. Il ragionamento a ritroso è però giudicato "innaturale";
- generalmente non è possibile controllare in maniera automatica le dimostrazioni in logica di Hoare;
- le dimostrazioni a mano sono "tediose".

Con la logica di Hoare "estesa" si possono risolvere questi problemi: il sistema di assiomi, infatti, permette che le prove possano essere condotte essenzialmente in avanti e, per buona parte, automatizzate da un dimostratore di teoremi. Lo strumento KeY-Hoare è stato creato esattamente per questa ragione.

Sintassi e assiomi

La logica di Hoare "estesa" si applica su un linguaggio iterativo del tutto simile a quello usato nella variante classica. Sono disponibili due tipi di dati incompatibili, intero e booleano, selezioni e iterazioni con una sintassi molto simile a quella di Java. La dichiarazione delle variabili non fa parte del linguaggio, è infatti specificata a parte in un'apposita sezione dei file che vanno forniti in input a KeY-Hoare. Segue la definizione in BNF della sintassi.

Program ::= (*Statement*)?
Statement ::= *EmptyStatement* | *AssignmentStatement* | *CompoundStatement* | *ConditionalStatement*| *LoopStatement*
EmptyStatement ::= ';'
AssignmentStatement ::= *Location* = *Expression*';'
CompoundStatement ::= *StatementStatement*
ConditionalStatement ::= if '('*BooleanExp*')' '{'*Statement*'}' else '{'*Statement*'}'
LoopStatement ::= while'('*BooleanExp*')' '{'*Statement*'}'
Expression ::= *BooleanExp* | *IntExp*
BooleanExp ::= *IntExp ComparisonOp IntExp* | *IntExp* == *IntExp* | *BooleanExp BooleanOp BooleanExp* | !*BooleanExp* | *Location* | **true** | **false**
IntExp ::= *IntExp IntOp IntExp* | $\mathbb{Z}$ | *Location*
ComparisonOp ::= < | <= | >= | >
BooleanOp ::= & | | | ==
IntOp ::= * | / | % | + | -

La logica di Hoare "estesa" è definita su quadruple anziché su triple. La scrittura fondamentale

$$\{P\}[U]s;\{Q\}$$

include infatti un nuovo elemento U, racchiuso tra parentesi quadre, oltre alle classiche precondizioni (P), postcondizioni (Q) e istruzioni (s). U rappresenta un insieme di assegnamenti alle variabili del programma di termini espressi nella logica del primo ordine. Questa "strana" definizione ha senso se si considera come U verrà usato nelle dimostrazioni:

- inizialmente U esprime gli assegnamenti alle variabili dei loro valori iniziali (prima che il programma le modifichi). Questa è una formalizzazione delle istruzioni del tipo `initialX=x` che solitamente si aggiungono in testa ai programmi nella logica di Hoare classica;
- gli assiomi dell'assegnamento, della selezione e dell'iterazione modificano U in modo che gli assegnamenti esprimano come il programma modifica le variabili. L'applicazione di questi assiomi è agevolata nel senso della lettura del programma, quindi in avanti;
- raggiunta l'ultima istruzione del programma (o del blocco di interesse) un altro assioma detto "assioma dell'uscita" può derivare la precondizione P da U.

U rappresenta quindi l'aggiornamento dello stato del programma, ed è utile in quanto permette di costruire un sistema di assiomi che non obblighi al ragionamento all'indietro come la logica di Hoare classica.

In pratica per provare la correttezza di un programma date le postcondizioni Q è sufficiente scrivere la U iniziale ed applicare un assioma per ogni assegnamento e struttura di controllo; arrivati al termine la derivazione di P è meccanica e automatizzabile.

Segue la definizione precisa degli assiomi nella logica di Hoare con aggiornamento esplicito dello stato.

$$\text{assignment}\ \frac{\{P\}\,[\mathcal{U},\ \mathtt{x} := \mathtt{e}]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{x=e;s}\,\{Q\}}$$

$$\text{exit}\ \frac{\vdash\ P \mathrel{->} \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\ \{Q\}} \qquad \text{skip}\ \frac{\{P\}\,[\mathcal{U}]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\ ;\mathtt{s}\,\{Q\}}$$

$$\text{conditional}\ \frac{\{P\ \&\ \mathcal{U}(\mathtt{b} \doteq \mathbf{true})\}\,[\mathcal{U}]\,\mathtt{s1;s}\,\{Q\} \qquad \{P\ \&\ \mathcal{U}(\mathtt{b} \doteq \mathbf{false})\}\,[\mathcal{U}]\,\mathtt{s2;s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathbf{if}\,(\mathtt{b})\{\mathtt{s1}\}\mathbf{else}\{\mathtt{s2}\}\mathtt{s}\,\{Q\}}$$

$$\text{loop}\ \frac{\vdash\ P \mathrel{->} \mathcal{U}(I) \qquad \{I\ \&\ \mathtt{b} \doteq \mathbf{true}\}\,[]\,\mathtt{s1}\,\{I\} \qquad \{I\ \&\ \mathtt{b} \doteq \mathbf{false}\}\,[]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathbf{while}\,(\mathtt{b})\{\mathtt{s1}\}\mathtt{s}\,\{Q\}}$$

Funzionalità di KeY-Hoare

KeY-Hoare è una variante del programma KeY per la verifica semiautomatica di programmi scritti in logica di Hoare con aggiornamento esplicito dello stato.

Key-Hoare riceve in input un file di testo opportunamente formattato che descrive il programma, le precondizioni, le postcondizioni e il parametro U iniziale. Esempi di questo file sono riportati nelle sezioni successive. Una volta caricato il file, KeY-Hoare permette di applicare gli assiomi tramite menù contestuali sul testo del programma in esame e di eseguire prove automatiche laddove questo fosse possibile.

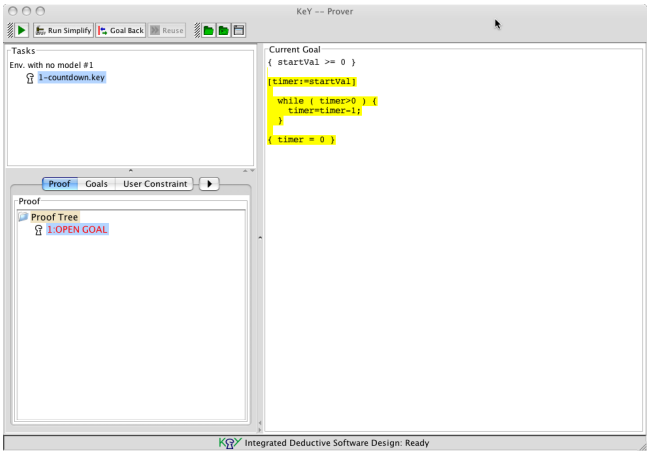

Figura 1: schermata iniziale di KeY-Hoare con un programma caricato.

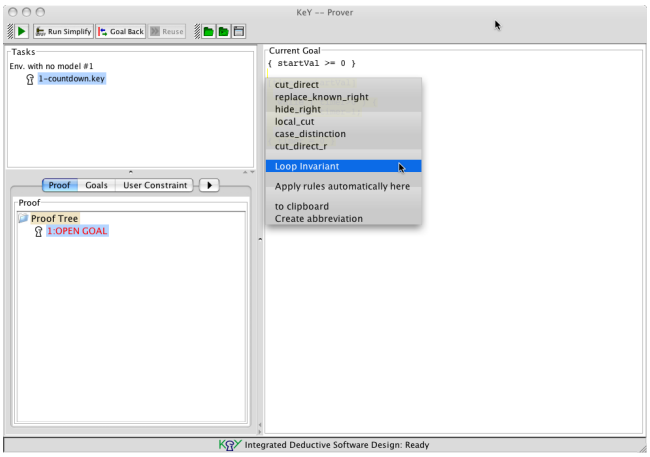

Figura 2: applicazione della regola dell'invariante.

La dimostrazione è suddivisa in obiettivi, ognuno dei quali può avere sotto-obiettivi. Un esempio è riportato nella figura seguente.

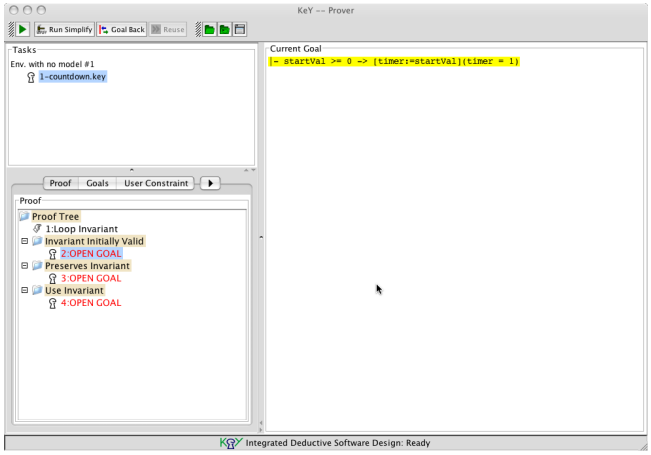

Figura 3: lista degli obiettivi da raggiungere per completare la dimostrazione.

Esempi della verifica di programmi

Vengono riportate le dimostrazioni in KeY-Hoare di alcuni programmi visti a lezione.

Conto alla rovescia

Questo programma:

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

esegue il conto alla rovescia della variabile `timer`, ammesso che inizialmente assuma un valore maggiore di zero. Quindi:

- Precondizione $\{P\}$: $timer \geq 0$
- Postcondizione $\{Q\}$: $timer = 0$
- Aggiornamento dello stato iniziale $[U]$: $timer := initialTimer$

Il programma, le precondizioni e le postcondizioni vanno specificate in un file `.key`, con la seguente sintassi, per essere interpretate da KeY-Hoare:

```
\functions {
   int intialTimer;
}

\programVariables {
  int timer;
}

\hoare {

{ intialTimer >= 0 }
```

```
[timer := intialTimer]


\[{
      while (timer>0) {
          timer = timer -1;
      }
}\]


{
   timer = 0
}
}
```

La sezione `\functions` specifica le variabili rigide, ossia quelle che possono comparire soltanto in U e non possono essere modificate dal programma, mentre la sezione `\programVariables` dichiara le variabili utilizzate dal programma.

La sezione successiva tra parentesi graffe rappresenta la precondizione, seguita da U, dal programma e dalla postcondizione.

Per la prova del programma è necessario:

- applicare l'assioma dell'invariante al ciclo; un invariante possibile è $timer \geq 0$; si generano tre sotto-obiettivi:
  - la validità iniziale dell'invariante, che è possibile provare automaticamente tramite la voce “Apply rules automatically here”;
  - il mantenimento dell'invariante, che si può provare applicando l'assioma dell'assegnamento e quello dell'uscita, seguiti dalla prova automatica;
  - la condizione di uscita, che si può provare applicando l'assioma dell'uscita, seguito dalla prova automatica.

L'output prodotto da KeY-Hoare è riportato nella figura seguente.

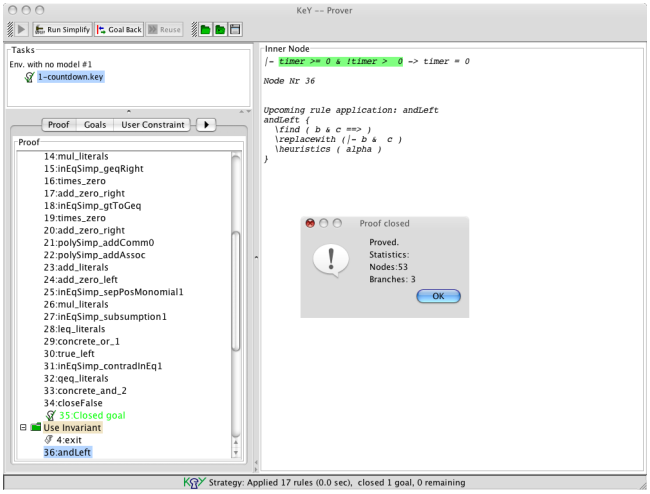

Figura 4: dimostrazione terminata.

Massimo tra due numeri

Questo programma:

```
if (x>y){
    max = x;
}
else{
    max = y;
}
```

assegna alla variabile `max` il massimo fra i valori di `x` ed `y`. Quindi:

- Precondizione $\{P\}$: sempre vera
- Postcondizione $\{Q\}$: $max \geq x \wedge max \geq y$
- Aggiornamento dello stato iniziale $[U]$: $x := initialX || y := initialY$

Il programma, le precondizioni e le postcondizioni si possono specificare come segue:

```
\functions {
   int initialX;
   int initialY;
}

\programVariables {
    int x;
    int y;
    int max;
}

\hoare {

{ true }

[x:=initialX || y:=initialY]

\[{
      if (x>y){
          max = x;
      }
      else{
          max = y;
      }
}\]

{
   max >= x & max >= y
```

```
}
}
```

Per la prova del programma è necessario:

- applicare l'assioma della selezione; si generano due sotto-obiettivi:
  - per il caso in cui la condizione è vera, applicare l'assioma di assegnamento, quello dell'uscita ed il ragionamento automatico;
  - per il caso in cui la condizione è falsa, applicare l'assioma di assegnamento, quello dell'uscita ed il ragionamento automatico;

Divisione intera

Questo programma:

```
result = 0;
rest = dividend;
while (rest>=divisor){
    rest = rest-divisor;
    result = result+1;
}
```

esegue la divisione intera tra `dividend` e `divisor` nelle variabili `result` (quoziente) e `rest` (resto). Quindi:

- Precondizione $\{P\}$: $dividend \geq 0 \wedge divisor \geq 0$
- Postcondizione $\{Q\}$: $dividend = result \cdot divisor + rest \wedge rest \geq 0 \wedge rest < divisor$
- Aggiornamento dello stato iniziale $[U]$: $dividend := initialDividend \| divisor := initialDivisor$

Il programma, le precondizioni e le postcondizioni si possono specificare come segue:

```
\functions {
   int initialDividend;
   int initialDivisor;
}

\programVariables {
    int dividend;
    int divisor;
    int result;
    int rest;
}

\hoare {

{ initialDividend>=0 & initialDivisor>=0 }

[dividend:=initialDividend || divisor:=initialDivisor]
```

```
\[{
    result = 0;
    rest = dividend;
    while (rest>=divisor){
        rest = rest-divisor;
        result = result+1;
    }
}\]


{
   dividend=result*divisor+rest & rest>=0 & rest<divisor
}
}
```

Per la prova del programma è necessario:

- applicare due volte l'assioma dell'assegnamento;
- applicare l'assioma dell'invariante al ciclo; un invariante possibile è $dividend = result \cdot divisor + rest \wedge rest \geq 0$; si generano tre sotto-obiettivi:
  - la validità iniziale dell'invariante, che è possibile provare automaticamente;
  - il mantenimento dell'invariante, che si può provare applicando due volte l'assioma dell'assegnamento, poi quello dell'uscita ed infine la prova automatica;
  - la condizione di uscita, che si può provare applicando l'assioma dell'uscita, seguito dalla prova automatica.

Quadrato

Questo programma:

```
i = 0;
result = 0;
while (! (i == x)) {
    result = result + x;
    i = i + 1;
}
```

calcola il quadrato di `x` nella variabile `result`. Quindi:

- Precondizione $\{P\}$: $x \geq 0$
- Postcondizione $\{Q\}$: $result = x \cdot x$
- Aggiornamento dello stato iniziale $[U]$: $x := initialX$

Il programma, le precondizioni e le postcondizioni si possono specificare come segue:

```
\functions {
   int initialX;
}
```

```
\programVariables {
    int x;
    int i;
    int result;
}

\hoare {

{ initialX>=0 }

[x:=initialX]

\[{
    i = 0;
    result = 0;
    while (! (i == x)) {
       result = result + x;
       i = i + 1;
    }
}\]

{
   result = x*x
}

}
```

Per la prova del programma è necessario:

- applicare due volte l'assioma dell'assegnamento;
- applicare l'assioma dell'invariante al ciclo; un invariante possibile è $result = x \cdot i$; si generano tre sotto-obiettivi:
  - la validità iniziale dell'invariante, che è possibile provare automaticamente;
  - il mantenimento dell'invariante, che si può provare applicando due volte l'assioma dell'assegnamento, poi quello dell'uscita ed infine la prova automatica;
  - la condizione di uscita, che si può provare applicando l'assioma dell'uscita, seguito dalla prova automatica.

Scambio di due variabili

Questo programma:

```
temp=a;
a=b;
b=temp;
```

scambia il contenuto delle variabili `a` e `b`. Quindi:

- Precondizione $\{P\}$: sempre vera
- Postcondizione $\{Q\}$: $a = initialB \wedge b = initialA$
- Aggiornamento dello stato iniziale $[U]$: $a := initialA \| b := initialB$

Il programma, le precondizioni e le postcondizioni si possono specificare come segue:

```
\functions {
   int initialA;
   int initialB;
}

\programVariables {
    int a;
    int b;
    int temp;
}

\hoare {

{ true }

[a:=initialA || b:=initialB]

\[{
    temp=a;
    a=b;
    b=temp;
}\]

{
   a=initialB & b=initialA
}

}
```

Per la prova del programma è necessario:

- applicare tre volte l'assioma dell'assegnamento;
- applicare l'assioma di uscita;
- applicare la prova automatica.