Program verification 2019

Lecture 3: Hoare logic and Invariants

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2019-01-15

Where are we and where are we going?

We have

- defined a simple language
- defined small step operation semantics of the language
- defined logical view of program statements
- defined strongest post and weakest pre
- defined logical strongest post and weakest pre

We will

- Hoare logic
- labelled transition system
- we cover some methods that try/avoid to compute lfp

Topic 3.1

Hoare logic

Hoare logic - our first method of verification

- Computing a super set of the reachable states(lfp) that does not intersect with error states should be suffice for our goal
- Since we do not know how to compute lfp, we will first see a method of writing pen-paper proofs of program safety
- Such a proof method has following steps
  - write (guess) a super set of reachable states
  - show it is actually a super set
  - show it does not intersect with error states
- First such method was proposed by Tony Hoare
  - it is sometimes called axiomatic semantics

Hoare Triple

Definition 3.1

$$\{P\}\mathtt{c}\{Q\}$$

- $P : \Sigma(V)$, *usually called precondition*
- $\mathtt{c} : \mathcal{P}$
- $Q : \Sigma(V)$, *usually called postcondition*

Definition 3.2

$\{P\}\mathtt{c}\{Q\}$ *is valid if all the executions of* $\mathtt{c}$ *that start from* P *end in* Q, *i.e.,*

$$\forall v, v'.\ v \models P \wedge ((v, \mathtt{c}), (v', \mathtt{skip})) \in T^* \Rightarrow v' \models Q.$$

Hoare proof obligation/goal

The safety verification problem is slightly differently stated in Hoare logic.

We remove assert statement from the language and no *err* variable.

Here, a verification problem is proving validity of a Hoare triple.

Example 3.1

Program

```
assume(⊤)
r := 1;
i := 1;
while(i < 3)
{
  r := r + z;
  i := i + 1
}
assert(r = 2z + 1)
```

$\rightarrow$

Hoare triple

$\{\top\}$
```
r := 1;
i := 1;
while(i < 3)
{
  r := r + z;
  i := i + 1
}
```
$\{r = 2z + 1\}$

Hoare Proof System

$$\frac{}{\{P\}\texttt{skip}\{P\}}\text{(skip rule)}$$

$$\frac{}{\{P[exp/x]\}\texttt{x := } exp\{P\}}\text{(assign rule)}$$

$$\frac{}{\{\forall x.P\}\texttt{x := } havoc()\{P\}}\text{(havoc rule)}$$

$$\frac{}{\{P\}\texttt{assume(F)}\{\texttt{F} \wedge P\}}\text{(assume rule)}$$

We may freely choose any of sp and wp for pre/post pairs for data statements.

Hoare Proof System (contd.)

$$\frac{\{P\}\mathtt{c}_1\{Q\} \quad \{Q\}\mathtt{c}_2\{R\}}{\{P\}\mathtt{c}_1;\mathtt{c}_2\{R\}}\text{(composition rule)}$$

$$\frac{\{P\}\mathtt{c}_1\{Q\} \quad \{P\}\mathtt{c}_2\{Q\}}{\{P\}\mathtt{c}_1[]\mathtt{c}_2\{Q\}}\text{(nondet rule)}$$

$$\frac{\{\mathtt{F} \wedge P\}\mathtt{c}_1\{Q\} \quad \{\neg\mathtt{F} \wedge P\}\mathtt{c}_2\{Q\}}{\{P\}\mathtt{if(F)}\ \mathtt{c}_1\ \mathtt{else}\ \mathtt{c}_2\{Q\}}\text{(if rule)}$$

$$\frac{P_1 \Rightarrow P_2 \quad \{P_2\}\mathtt{c}\{Q_2\} \quad Q_2 \Rightarrow Q_1}{\{P_1\}\mathtt{c}\{Q_1\}}\text{(Consequence rule)}$$

$$\frac{\{\mathtt{I} \wedge \mathtt{F}\}\mathtt{c}\{\mathtt{I}\}}{\{\mathtt{I}\}\mathtt{while(F)}\ \mathtt{c}\{\neg F \wedge \mathtt{I}\}}\text{(While rule)}$$

Non-mechanical step: invent `I` such that the while rule holds. `I` is called loop-invariant.

Example Hoare proof

Example 3.2

Consider loop invariant: $\mathtt{I} = (i \leq 3 \wedge r = (i-1)z + 1)$

$\{\top\}$
$\mathtt{r} := 1;$
$\{\mathtt{r} = 1\}$
$\mathtt{i} := 1;$
$\{\mathtt{I}\}$
$\mathtt{while}(\mathtt{i} < 3)$
$\{$
$\quad\{\mathtt{I} \wedge \mathtt{i} < 3\}$
$\quad\mathtt{r} := \mathtt{r} + \mathtt{z}$
$\quad\{P_5\}$
$\quad\mathtt{i} := \mathtt{i} + 1$
$\}$
$\{r = 2z + 1\}$

$$\dfrac{\dfrac{}{\{\top\}\mathtt{r} := 1\{\mathtt{r} = 1\}} \quad \dfrac{}{\{\mathtt{r} = 1\}\mathtt{i} := 1\{\mathtt{I}\}}}{\{\top\}\mathtt{r} := 1; \mathtt{i} := 1; \{\mathtt{I}\}}$$

$$\dfrac{\dfrac{}{\begin{array}{c}\{i < 3 \wedge \mathtt{I}\}\\ \mathtt{r} := \mathtt{r} + \mathtt{z}\\ \{i < 3 \wedge r = iz + 1\}\end{array}} \quad \dfrac{}{\begin{array}{c}\{P5 \triangleq i < 3 \wedge r = iz + 1\}\\ \mathtt{i} := \mathtt{i} + 1\\ \{\mathtt{I}\}\end{array}}}{\{i < 3 \wedge \mathtt{I}\}\mathtt{r} := \mathtt{r} + \mathtt{z}; \mathtt{i} := \mathtt{i} + 1\{\mathtt{I}\}}$$

$$\dfrac{\dfrac{\{\top\}\mathtt{r} := 1; \mathtt{i} := 1; \{\mathtt{I}\} \quad \{i < 3 \wedge \mathtt{I}\}\mathtt{r} := \mathtt{r} + \mathtt{z}; \mathtt{i} := \mathtt{i} + 1\{\mathtt{I}\}}{\{\top\}\mathtt{r} := 1; ..; \mathtt{while}(..)..\{\mathtt{I} \wedge \mathtt{i} \geq 3\}} \quad \mathtt{I} \wedge \mathtt{i} \geq 3 \Rightarrow r = 2z + 1}{\{\top\}\mathtt{r} := 1; ..; \mathtt{while}(..)..\{r = 2z + 1\}}$$

Topic 3.2

Program as labeled transition system

A more convenient program model

- Simple language has many cases to write an algorithm
- automata like program models allow more succinct description of verification methods

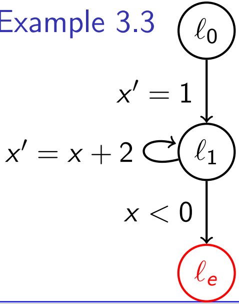
Program as labeled transition system (LTS)

Definition 3.3

A program P is a tuple $(V, L, \ell_0, \ell_e, E)$, where

- *V is a vector of variables,*
- *L be set of program locations,*
- *ℓ_0 is initial location,*
- *ℓ_e is error location, and*
- *$E \subseteq L \times \Sigma(V, V') \times L$ is a set of labeled transitions between locations.*

Example 3.3



$V = [\mathrm{x}]$
$L = \{\ell_0, \ell_1, \ell_e\}$
$E = \{(\ell_0, \mathrm{x}' = 1, \ell_1),$
$\quad (\ell_1, \mathrm{x}' = \mathrm{x} + 2, \ell_1),$
$\quad (\ell_1, \mathrm{x} < 0, \ell_e)\}$

Notation:
If $e = (\ell, \rho(V, V'), \ell') \in E$, then
$e(V, V') \triangleq \rho(V, V')$,
$e(loc) \triangleq \ell$ and
$e(loc') \triangleq \ell'$

Guarded command

Definition 3.4 (Guarded command)

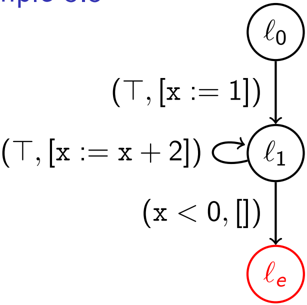
A guarded command is a pair of a formula in $\Sigma(V)$ and a sequence of update constraints (including havoc) of variables in V.

Note: we may write transition formulas as guarded commands. Havoc encodes inputs.

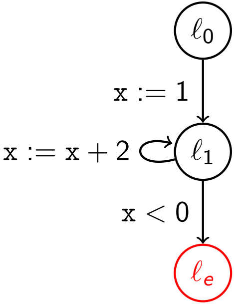
Example 3.4

Consider $V = [\mathrm{x}, \mathrm{y}]$. The formula represented by the guarded command $(\mathrm{x} > \mathrm{y}, [\mathrm{x} := \mathrm{x} + 1])$ is $\mathrm{x} > \mathrm{y} \wedge \mathrm{x}' = \mathrm{x} + 1 \wedge \mathrm{y}' = \mathrm{y}$.

Example 3.5



simplified view $\rightarrow$



Semantics

Consider program $P = (V, L, \ell_0, \ell_e, E)$.

Definition 3.5

A state $s = (\ell, v)$ of a program is program location ℓ and a valuation v of V.

Let $v(x) \triangleq$ value of variable x in v
For state $s = (\ell, v)$, let $s(x) \triangleq v(x)$ and $s(loc) \triangleq \ell$

Definition 3.6

A path $\pi = e_1, \dots, e_n$ in P is a sequence of transitions such that, for each $0 < i < n$, $e_i = (\ell_{i-1}, \_, \ell_i)$ and $e_{i+1} = (\ell_i, \_, \ell_{i+1})$.

Definition 3.7

An execution corresponding to path $e_1, \dots, e_n$ is a sequence of states $(\ell_0, v_0), \dots, (\ell_n, v_n)$ such that $\forall i \in 1..n$, $e_i(v_{i-1}, v_i)$ holds true.
An execution belongs to P if there is a corresponding path in P.

Definition 3.8

P is safe if there is no execution of P from ℓ_0 to ℓ_e.

Path constraints

$V_i \triangleq$ variable vector obtained by adding subscript i after each variable in V.

Definition 3.9

For a path $\pi e_1, \ldots, e_n$, path constraints is $\bigwedge_{i \in 1..n} e_i(V_{i-1}, V_i)$.
A path is feasible if corresponding path constraints is satisfiable.

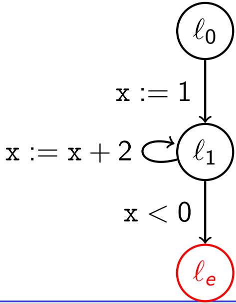
Let $\textsc{PathCons}(\pi)$ returns path constraints of π.

Path constraints are also known as “SSA formulas”

Theorem 3.1

A path is feasible then there is an execution that corresponds to the path.

Example 3.6



Consider path
$(\ell_0, \mathtt{x} := 1, \ell_1)$, $(\ell_1, \mathtt{x} := \mathtt{x} + 2, \ell_1)$, $(\ell_1, \mathtt{x} < 0, \ell_e)$

Path constraint for the path is
$F = (\mathtt{x}_1 = 1 \land \mathtt{x}_2 = \mathtt{x}_1 + 2 \land \mathtt{x}_2 < 0)$

Since F is unsat, there is no execution along the path

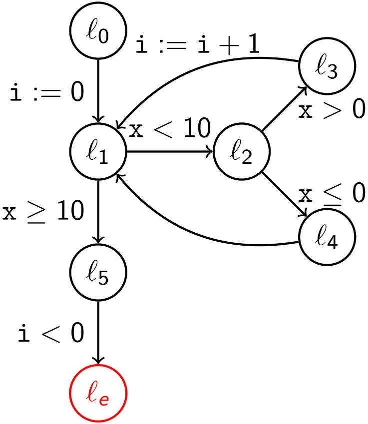
From simple language to labelled transition system

Theorem 3.2

Show simple programming language is isomorphic to the labelled transition system.

Example 3.7

```
L0: i = 0;
L1: while( x < 10 ) {
L2:  if x > 0 then
L3:   i := i + 1
     else
L4:   skip
    }
L5: assert( i >= 0 )
```



Cut-points

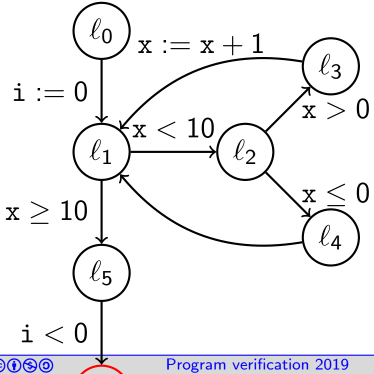
Definition 3.10

For a program $P = (V, L, \ell_0, \ell_e, E)$, $\textsc{CutPoints}(P)$ is the a minimal subset of L such that every path of P containing a loop passes through one of the location in $\textsc{CutPoints}(P)$.

$\textsc{CutPoints}(P)$ in LTS loop heads in simple language

Example 3.8

Consider the following program P.



$\textsc{CutPoints}(P) = \{\ell_1\}$

Reminder: symbolic strongest post

$$sp : \Sigma(V) \times \Sigma(V, V') \rightarrow \Sigma(V)$$

We define symbolic post over labels of P as follows.

$$sp(F, \rho) \triangleq (\exists V : F(V) \land \rho(V, V'))[V/V']$$

We assume that ρ and F are in a theory that admits quantifier elimination

Using polymorphism, we also define $sp((\ell, F), (\ell, \rho, \ell') \in E) \triangleq (\ell', sp(F, \rho))$.

For path $\pi = e_1, .., e_n$ of P, $sp((\ell, F), \pi) \triangleq sp(sp((\ell, F), e_1), e_2..e_n)$.

Topic 3.3

Loop invariants

Invariants

Definition 3.11
For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called invariant map *if, for each $\ell \in L$, all reachable states at ℓ satisfy $\mathrm{I}(\ell)$.*

Definition 3.12
For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called inductive *if, for each $(\ell, \rho, \ell') \in E$,*

$$sp(\mathrm{I}(\ell), \rho) \Rightarrow \mathrm{I}(\ell').$$

Definition 3.13
For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called safe *if $\mathrm{I}(\ell_0) = \top$ and $\mathrm{I}(\ell_e) = \bot$*

Theorem 3.3
For P, if I is inductive and safe then I is an invariant and P is safe.

Invariant checking: is I a safe inductive invariant map?

Exercise 3.1
What is the algorithm for invariant checking?

Cut-point invariant maps

Let P be a program and $C = \textsc{CutPoints}(P) \cup \{\ell_0, \ell_e\}$.

Definition 3.14

A map $\mathrm{I} : C \to \Sigma(V)$ *is called cut-point invariant map if, for each* $\ell \in C$*, all reachable states at* ℓ *satisfy* $\mathrm{I}(\ell)$*.*

Definition 3.15

A map $\mathrm{I} : C \to \Sigma(V)$ *is called inductive if, for each* $\ell, \ell' \in C$ *and* $\pi \in \textsc{LoopFreePaths}(P, \ell, \ell')$, $sp(\mathrm{I}(\ell), \pi) \Rightarrow \mathrm{I}(\ell')$.

Definition 3.16

A map $\mathrm{I} : C \to \Sigma(V)$ *is called safe if* $\mathrm{I}(\ell_0) = \top$ *and* $\mathrm{I}(\ell_e) = \bot$

Theorem 3.4

If I *is inductive and safe then* I *is an cut-point invariant map and* P *is safe.*

Proof.

Every path from ℓ_0 to ℓ_e can be segmented into loop free paths between cut-points. Therefore, no such path is feasible. □

Annotated verification: VCC demo

`http://rise4fun.com/Vcc`

Exercise 3.2

Complete the following program such that Vcc proves it correct

```
#include <vcc.h>
int main()
{
  int x, y;
  _(assume x > y +3 && x < 3000 )
  while( 0 < y ) _(invariant ....) {
    x = x + 1;
    y = y -1;
  }
  _(assert x >= y)
  return 0;
}
```

Annotated verification

- There are many tools like VCC that require user to write invariants at the loop heads and function boundaries
- Rest of the verification is done as discussed in earlier slides
- User needs to do a lot of work, not a very desirable method

What if we want to compute the invariants automatically?

Topic 3.4

Problems

Problem 1

1. (1) Prove the following Hoare triple is valid

```
{true}
assume( n > 1);
i = n;
x = 0;
while(i > 0) {
  x = x + i;
  i = i - 1;
}
{ 2x = n*(n+1) }
```

Problem 2

2. (1) Fill the annotations to prove following program correct via Vcc

```
#include <vcc.h>
int main()
{
  int x = 0, y = 2;
 _(assume 1==1 )
  while( x < 3 ) _(invariant ... ) {
    x = x + 1;
    y = 3;
   }
  _(assert y == 3)
  return 0;
}
```

Problem 3

3. (2) extend your tool in the last assignment in the following ways
   - define classes for
     - locations,
     - variables,
     - guarded commands,
     - transitions (give names to the transitions), and
     - programs
   - encode the program in example **??** using the class
   - Write a function that computes path constraints for a given path
   - Read path from command line as space separated transition names and output the path constraints

Problem 4

Exercise 3.3

Write inductive invariants at the loop heads in the following sorting algorithms such that they prove that at the end array is sorted.

- *Bubble sort*

```
procedure bubbleSort( A : list of sortable items )
   n = length(A)
   repeat
     swapped = false
     for i = 1 to n-1 inclusive do
       if A[i-1] > A[i] then
         swap( A[i-1], A[i] )
         swapped = true
       end if
     end for
   until not swapped
end procedure
```

- *Quick sort*

```
function merge_sort(list m)
    if length of m <= 1 then
        return m
    var left := empty list
    var right := empty list
    for each x with index i in m do
        if i =< (length of m)/2 then
            add x to left
        else
            add x to right
    left := merge_sort(left)
    right := merge_sort(right)
```

Exercise

Podelski Trace abstraction example. TAPAS'17

```
int main( int n ) {
  assume( p == 0 );
  while( n > 0 ) {
    assert( p != 0 );
    if( n== 0 ) {
       p = 0;
    }
    n--;
  }
}
```

Exercise

Write inductive loop invariants and prove Hoare logic!! Make a question out of the problem.

```
int main ( int A[ N ] , int B[ N ] , int C[ N ] ) {
  int i;
  int j = 0;
  for (i = 0; i < N ; i++) {
    if ( A[i] == B[i] ) {
      C[j] = i;
      j = j + 1;
    }
  }

  assert( forall ( int x ) :: ( 0 <= x && x < j ) ==> ( C[x] <=
  assert( forall ( int x ) :: ( 0 <= x && x < j ) ==> ( C[x] >=
}
```

End of Lecture 3