innovate achieve lead

BITS Pilani
Pilani Campus

CS/IS F214
Logic in Computer Science

MODULE: **PROGRAM VERIFICATION**

Floyd-Hoare Logic: Examples

Hoare Logic – Example 1: Computing the *square root*

- *Compute the square root of a given value* (special case of Newton-Raphson):
  - Newton-Raphson is an iterative technique for computing the solution of an equation y = f(x)
    - i.e. we start with an initial value – a guess, and estimate the next value, based on “rate of change”

Example 1: Computing the *square root* : Guess

- In our case, find an x that satisfies the equation $y = x^2$
  - how do we estimate the next value?
    - each "**guess**" or "**estimate**" is the length (**l**) and breadth (**b**) of a rectangle such that **l*b=y**

y = x * x

y = l * b

b <= x ∧ x <= l

Example 1: Computing the *square root* : Iterate

- In our case, find an x that satisfies the equation $y = x^2$
  - how do we estimate the next value?
    - each "**guess**" or "**estimate**" is the length (**l**) and breadth (**b**) of a rectangle such that **l*b=y**

$$y = x * x$$

$$y = l * b$$

$$b <= x \wedge x <= l$$

    - in each iteration *increase* **b** and *decrease* **l** until they match

$$y = l_0 * b_0$$

$$y = l_1 * b_1$$

$$y = l_2 * b_2$$

…

Example 1: Computing the *square root* : Iterate

$y = l_0 * b_0$

$y = l_1 * b_1$

…

$y = l_n * b_n$

Invariant: $b_i <= x \wedge x <= l_i$

Hoare Logic – Example1 : Computing Square Root

- Since **y** is fixed, we need to guess only one side (say **r**)
  - the other side is automatically obtained (y/r)
    - /*Invariant: **(r<=√y ∧ √y<=y/r) ∨ (y/r<=√y ∧ √y<=r)** */
      - This invariant is trivially true for any **r**, given a **y**. Why?

- Since we are doing real-number computations *there is bound to be an error:*
  - so, we recompute **r** until **r*r** gets close to **y** i.e.
    - we recompute **r** until **err < EPS**,
      - where **err = abs(r*r-y)/y** and **EPS** is the margin

Hoare Logic – Example1 : Computing Square Root

- With this we can write down an outline:

  r = *init_guess* ;   /* Does the initial value matter? */

  err = abs(r*r – y) / y ;

  /*Precondition: $(r<=\sqrt{y} \wedge \sqrt{y}<=y/r) \vee (y/r<=\sqrt{y} \wedge \sqrt{y}<=r)$ */

  ...

  /* Postcondition: $((r<=\sqrt{y} \wedge \sqrt{y}<=y/r) \vee (y/r<=\sqrt{y} \wedge \sqrt{y}<=r)) \wedge$ err <= EPS */

Hoare Logic – Example1 : Computing Square Root

- With this we can write down an outline:

```
r = init_guess ;   /* Does the initial value matter? */
err = abs(r*r – y) / y ;
/*Precondition: (r<=√y ∧ √y<=y/r) ∨ (y/r<=√y ∧ √y<=r) */
while (err > EPS) {
    /*Invariant: (r<=√y ∧ √y<=y/r) ∨ (y/r<=√y ∧ √y<=r) */
     ...
    err = abs(r*r – y) / y;
 }
/* Postcondition ((r<=√y ∧ √y<=y/r) ∨ (y/r<=√y ∧ √y<=r)) ∧
                          err <= EPS */
```

Hoare Logic – Example1 : Computing Square Root

- We need to estimate the next value of r such that err reduces:

```
r = y/2;
err = abs(r*r – y) / y ;
/*Precondition:   (r<=√y ∧ √y<=y/r) ∨ (y/r<=√y ∧ √y<=r)*/
while (err > EPS) {
    /*Invariant:   (r<=√y ∧ √y<=y/r) ∨ (y/r<=√y ∧ √y<=r)*/
     r = (r + y/r)/2.0 ;
    err = abs(r*r – y) / y;
 }
/* Postcondition: ((r<=√y ∧ √y<=y/r) ∨ (y/r<=√y ∧ √y<=r)) ∧
                    err <= EPS */
```

Hoare Logic – Example1 : Computing Square Root

We need to estimate the next value of r such that err reduces:

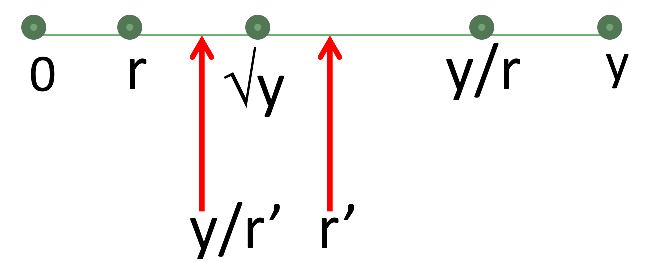
```
r = y/2;
err = abs(r*r – y) / y ;
while (err > EPS) {
    r = (r + y/r)/2.0 ;
    err = abs(r*r – y) / y;
}
```

Termination Argument:

err reduces below the limit eventually

because:

$\sqrt{y}$ lies between r and y/r



(r + y/r)/2 lies between r and y/r.

Formal Termination Argument:

1. Identify a quantity that is finite and reduces in each iteration. [**Hint:** *The interval (b,l) gets shorter in every iteration. Translate this to program variables*. **End of Hint**.]
2. Prove that this quantity will eventually result in a value that implies termination.

Example 2

- /* Pre-condition: $y = 2^k$, for some k>=0 */

  int power2(int x, int y)

  {

  /* Pre-condition $x^y = A^B \wedge x{=}A \wedge y{=}B$ */

  while (y > 1) { /* Loop Invariant: $x^y = A^B$ */

  Prove this!

  x = x * x;

  y = y / 2;

  }

  /* Post-condition: $x^y = A^B \wedge y{=}1$ */

  return x;

  }

- /* power2(A,B) returns A^B */

Example 2a

```
/* Precondition: y>=0 */
int pow(int x, int y)
{
/* Derive the algorithm and the loop-invariant */
/* Prove that the algorithm will terminate */
    }
```

/* Postcondition: returns x^y */

Example 5 - Length of a linked list

/* Pre-condition: **ls** is a linear linked list i.e. **ls** is not cyclic */

```
int length(LINK ls)
{
  ...
}
```

/* Post-condition: returns **len**, the length of ls */

Example 5 - Length of a linked list

```
/* Pre-condition:  ls is a linear linked list i.e. ls is not cyclic */
    int len(LINK ls)
    {
      int count = 0;
      while (ls != NULL) {
        /* Loop Invariant:  length(ls_init) = count + length(ls) */
       ...
      }
      return count;
    }
/* Post-condition: returns length(ls) */
```

Example 5 - Length of a linked list

/* Pre-condition: **ls** is a linear linked list i.e. **ls** is not cyclic */

```
int len(LINK ls)
{
  int count = 0;
  while (ls != NULL) {
    /* Loop Invariant:  length(ls_init) = count + length(ls) */
    ls = ls-->next;
    count = count + 1;
  }
  return count;
}
```

Prove that this condition remains invariant!

/* Post-condition: returns length(**ls_init**) */

Example 5 - Length of a linked list

/* Pre-condition: **ls** is a linear linked list i.e. **ls** is not cyclic */

```
int len(LINK ls)
{
  int count = 0;
  while (ls != NULL) {
    ls = ls-->next;
    count = count + 1;
  }
  return count;
}
```

Prove that this loop terminates!

/* Post-condition: returns length(**ls_init**) */