Class 24 – 4/22

Proving properties of imperative programs – "Hoare logic"

- Judgments, a.k.a. "Hoare formulas"
- Axioms
- Rules of inference

“Invariants” in programming

Invariants are relationships among the variables of a program that always hold.

- Within a class, invariants represent consistency among fields, e.g. “the `count` field is always the same as the number of non-zero entries in the `values` array,” or “if the `visited` bit of this node is set, and this node is not the entry point of the graph, then at least one predecessor’s `visited` bit is set.”

"Invariants" in programming

- In a loop, invariants represent relationships that hold *no matter how often the body of the loop is executed.*

```
x = x0; y = 1;
 while ( x>0 )
       {y := y*x; x := x-1;}
```

Inv: $y = x+1 * \dots * x_0$

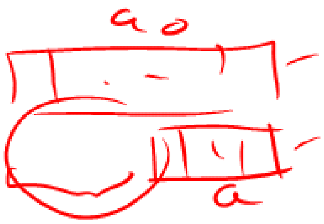
Inv + termination = desired result

```
a = a0; b = 0;
while (a != []) { b = b + hd a;
                  a = tl a; }
```

$b = \sum a0 - \sum a$

+ termination

$\Rightarrow b = \sum a0$



Hoare logic

- Hoare logic, introduced by C.A.R. Hoare, is an effort to formalize the proof of correctness of *imperative* programs.
- It is a proof system in which properties of programs are proved from properties of their component parts.
- It includes a formalization of the notion of *loop invariant*, which forms the hard part of most proofs.

Correctness of imperative programs

- "Hoare formula" says that if the variables in a program satisfy some properties, then after executing a given program, they satisfy some different properties.
- Examples:

```
x>0 {  while ( x>0 )
           {y := y*x; x := x-1;}  }  y = y * x!

 x=x0 & y=y0 { t := x; x := y; y := t }
                                     x=y0 & y=x0
```

```
true {  if ( x<0 ) x := -x; } $x = |x|$
```

```
true { n := length(a);  b := [hd a];
          a := tl a;
          while (a != []) {
                b = (hd a + hd b) :: b;
                a = tl a; }
       } $b_i = \sum_{k=0}^{n-i-1} a_k$   (where $b_i$ = hd (tl$^i$ b),
                                  and similarly for $a_k$)
```

Inference rules of Hoare logic

Judgments: P {S} Q

P, Q assertions about variables in the program

S a statement in this language:

```
Stmt -> Var :- Expr | Stmt;Stmt
          | if (Expr) then Stmt else Stmt
          | while (Expr) Stmt
```

Inference rules of Hoare logic

$$\frac{}{P[e/x]\ \{x := e\}\ P}$$

$$\frac{P\ \&\ b\ \{S\}\ P}{P\ \{\text{while } (b)\ S\}\ P\ \&\ \neg b}$$

$$\frac{P\ \{S_1\}\ Q \qquad Q\ \{S_2\}\ R}{P\ \{S_1;\ S_2\}\ R}$$

$$\frac{P \Rightarrow P' \quad P'\ \{S\}\ Q' \quad Q' \Rightarrow Q}{P\ \{S\}\ Q}$$ (consequence)

$$\frac{P\ \&\ b\ \{S_1\}\ Q \qquad P\ \&\ \neg b\ \{S_2\}\ Q}{P\ \{\text{if } (b) \text{ then } S_1 \text{ else } S_2\}\ Q}$$

Rule of assignment

$$\frac{}{P[e/x]\ \{x := e\}\ P}$$

if P is true about x, was true before the assignment

$$\frac{}{0*y=0\ \{x:=0\}\ x*y=0}$$

$$\frac{}{\underbrace{(x+1)=2}_{\equiv\ x=1}\ \{x:=\ x+1\}\ x=2}$$

Examples

y=2 { x:=y } x=2   y=2 & y=2 {x:=y} x=2 & y=2

y=2 { x:=2 } y=x

x+1=n+1 { x:=x+1 } x=n+1

x+1=n { x:=x+1 } x=n] ⇒ $\underbrace{x=n-1\ \{x:=x+1\}}_{\text{not an instance}}\ x=n$

2=2
~~true~~ { x:=2 } x=2

Rule of consequence

$$\frac{P \Rightarrow P' \quad P'\ \{S\}\ Q' \quad Q' \Rightarrow Q}{P\ \{S\}\ Q}$$

$$\frac{\begin{array}{c} x+1=n \\ \equiv x=n-1 \end{array} \qquad \dfrac{}{x+1=n\ \{x:=x+1\}\ x=n}\,\text{Assign} \qquad \begin{array}{c} x=n \\ \equiv x=n \end{array}}{x=n-1\ \{x:=x+1\}\ x=n}\,\text{Cons}$$

Sequence rule

$$\frac{P\ \{S_1\}\ Q \qquad Q\ \{S_2\}\ R}{P\ \{S_1;\ S_2\}\ R}$$

$$\dfrac{\dfrac{\dfrac{}{\begin{array}{c}x=x_0\ \&\\ x=x_0\ \&\\ y=y_0\end{array}\ \{t:=x\}\ \begin{array}{c}t=x_0\ \&\\ x=x_0\ \&\\ y=y_0\end{array}}\text{Assign}}{\begin{array}{c}x=x_0\\ \&\ y=y_0\end{array}\ \{t:=x\}\ \begin{array}{c}t=x_0\ \&\\ x=x_0\ \&\\ y=y_0\end{array}}(\text{con}) \qquad \dfrac{\dfrac{\{}{\begin{array}{c}t=x_0\ \&\\ x=x_0\ \&\\ y=y_0\end{array}\ \{x:=y\}\ \begin{array}{c}t=x_0\\ \&\ x=y_0\\ \&\ y=y_0\end{array}} \quad \dfrac{\}}{\begin{array}{c}t=x_0\\ \&\ x=y_0\\ \&\ y=y_0\end{array}\ \{y:=t\}\ \begin{array}{c}x=y_0\\ \&\ y=x_0\end{array}}}{\underbrace{\begin{array}{c}t=x_0\ \&\\ x=x_0\ \&\\ y=y_0\end{array}}_{P}\ \left\{\begin{array}{c}\overbrace{x:=y;}^{S_1}\\ \underbrace{y:=t}_{S_2}\end{array}\right\}\ \begin{array}{c}\overbrace{x=y_0}^{R}\\ \&\ y=x_0\end{array}}}{\underbrace{\begin{array}{c}x=x_0\\ \&\ y=y_0\end{array}}_{P}\ \{\underbrace{t:=x}_{S_1};\ \underbrace{x:=y;\ y:=t}_{S_2}\}\ \underbrace{x=y_0\ \&\ y=x_0}_{R}}\text{Seq}$$

If rule

$$\frac{P \,\&\, b\ \{S_1\}\ Q \qquad P \,\&\, \neg b\ \{S_2\}\ Q}{P\ \{\text{if } (b) \text{ then } S_1 \text{ else } S_2\}\ Q}$$

$$\dfrac{\dfrac{x=x_0 \,\&\, x<0 \Rightarrow -x=|x_0| \qquad \dfrac{}{-x=|x_0|\ \{x:=-x\}\ x=|x_0|}\,\text{Asgn}}{\underbrace{x=x_0}_{P} \,\&\, \underbrace{x<0}_{b}\ \{\underbrace{x:=-x}_{S_1}\}\ \underbrace{x=|x_0|}_{Q}}\,\text{Cons} \qquad \dfrac{x=x_0 \,\&\, x\not<0 \Rightarrow x=|x_0| \qquad \dfrac{}{x=|x_0|\ \{x:=x\}\ x=|x_0|}\,\text{Asgn}}{\underbrace{x=x_0}_{P} \,\&\, \underbrace{x\not<0}_{\neg b}\ \underbrace{\{x:=x\}}_{S_2}\ \underbrace{x=|x_0|}_{Q}}\,\text{Cons}}{\underbrace{x=x_0}_{P}\ \left\{\begin{array}{l} \text{if } \underbrace{x<0}_{b} \\ \text{then } \overbrace{x:=-x}^{S_1} \\ \text{else } \underbrace{x:=x}_{S_2} \end{array}\right\}\ \underbrace{x=|x_0|}_{Q}}$$

While rule

$$\frac{P \,\&\, b \;\{S\}\; P}{P \;\{\text{while } (b)\; S\}\; P \,\&\, \neg b}$$

✓

$$\frac{Inv \,\&\, i<n \;\{\quad\}\; Inv}{Inv \;\{\text{while} \ldots\}\; Inv \,\&\, \boxed{i=n}}$$

$i \not< n$

$\Rightarrow s = \sum_{j=1}^{n-1} j$

$s=0$
$\& \, i=0$ $\{$ while $(\overset{i \neq n}{\boxed{i<n}})$ $\{$

```
    s := s + i;
    i := i + 1;
}
```

$\} \; s = \sum_{j=1}^{n-1} j$

Invariant:

$s = \sum_{j=1}^{i-1} j$

$\& \; \underline{i \leq n}$

Comments on Hoare logic

- Proofs in Hoare logic are *almost* syntax-directed, i.e. almost have the same shape as the program being proved. The only exceptions are the uses of the rule of consequence.

- Applying Hoare rules is largely mechanical – given A and Q, most of the proof (including P) can be generated automatically. Creativity is required mainly in determining the invariant in a while loop, because Q may not have the form "P & ¬b", and so a formula of that form needs to be found (after which the rule of consequence can be used, proving P & ¬b ⇒ Q).

→ also for proving assertions in rule of consequence

Example: gcd algorithm

```
$a > 0$ & $b > 0$ & $a=a_0$ & $b=b_0$ {
        while ($a \neq b$)
Inv & $a \neq b$ →
            if ($a > b$) then a := a − b;
                    else b := b − a;
Inv →
      }  $a = gcd(a_0, b_0)$
```

$gcd(a, b) = gcd(a_0, b_0)$

$I : \gcd(a,b) = \gcd(a_0,b_0)$

Asgn

$$\frac{}{\{\gcd(a-b,b) = \gcd(a_0,b_0)\}\ a:=a-b\ \{\gcd(a,b) = \gcd(a_0,b_0)\}}$$

✓

$\Uparrow$?　　　$\Downarrow$　Cons

$$\frac{\{\gcd(a-b,b) = \gcd(a_0,b_0)\}\ a:=a-b\ \{\gcd(a,b) = \gcd(a_0,b_0)\}}{I \,\&\, a \neq b \,\&\, a > b\ \{a:=a-b\}\ I}
\qquad
\frac{\quad}{I \,\&\, a \neq b \,\&\, a \not> b\ \{b:=b-a\}\ I}$$

Similarly

$$\frac{}{\overbrace{I \,\&\, \underbrace{a \neq b}_{b}}^{P}\ \{\overset{b}{\text{if}}\ (a>b : \ldots\}\ \overbrace{\underbrace{I}_{P}}^{Q}}$$

While

$$\frac{}{\underbrace{I}_{P}\ \{\text{while}\ (\underbrace{a\neq b}_{b})\ \underbrace{\ldots}_{S}\}\ \underbrace{I}_{P} \,\&\, \underbrace{a=b}_{\neg b}}$$

Cons

$\Rightarrow I$

$$a>0 \,\&\, b>0 \,\&\, a=a_0 \,\&\, b=b_0\ \{\text{while}\ (a\neq b)\ \{\ldots\}\}\ a = \gcd(a_0,b_0)$$

A note about the assignment axiom

Assignment rule fails if aliasing is possible.
Aliasing is when two different expressions refer to the same location.

4=4 & a[j]=3 {a[i] := 4} a[i]=4 & a[j]=3] ← false if i=j

4=4 & s.x=3 {r.x := ~~3~~ 4} r.x=4 & s.x=3] ← false, if r=s

4=4 & y=3 {x := 4} x=4 & y=3] ← In C++, if x, y are ref param's, with same argument

Assignment axiom still valid as long as right-hand side of assignment is not aliasable.

f(int& x, int& y) { ... x=4; }

f(a, a) →

Sum of n

```
x = 0 & y = 0
{
   while (y < n) {
       y := y + 1;
       x := x + y
    }
}
x = 1 + ... + n
```

Fibonacci

$x = 0 \ \& \ y = 1 \ \& \ z = 1 \ \& \ 1 \le n$

```
{
   while (z < n) {
      y := x + y;
      x := y – x;
      z := z + 1;
   }
}
```

$y = \mathit{fib}\ n$

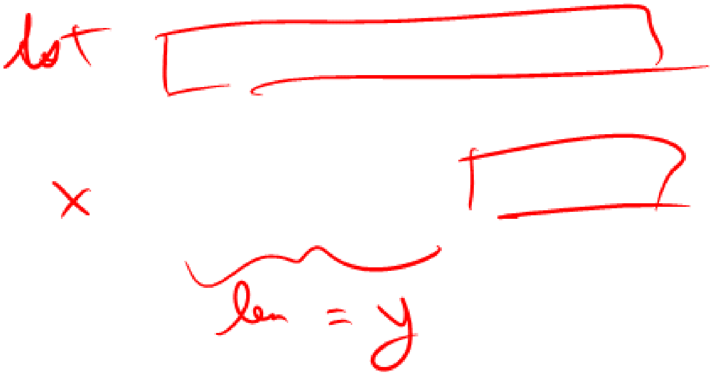
y:	1	1	2	3	5	8
x:	0	1	1	2	3	5
z=	1	2	3	4	5	6

Inv: $y = fib(z)$
$\& \ x = fib(z-1)$
$\& \ z \le n$

List length

```
x = lst & y = 0
{
    while (x ≠ []) {
        x := tl x;
        y := y + 1;
    }
}
y = len lst
```

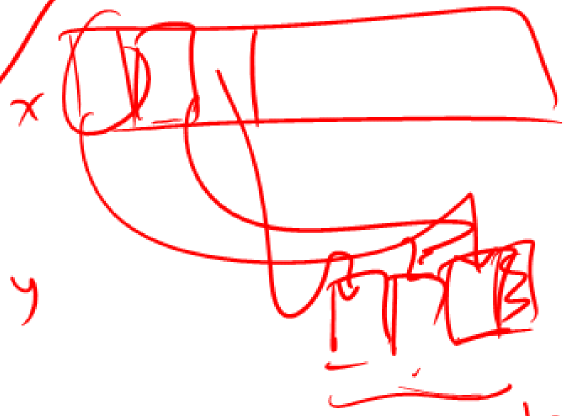
Inv : y = len lst - len x



List reverse

```
x = lst & y = []
{
    while (x ≠ []) {
        y := hd x :: y;
        x := tl x;
    }
}
y = rev lst
```

Inv: rev y @ x = lst



rev y = part of lst taken off x