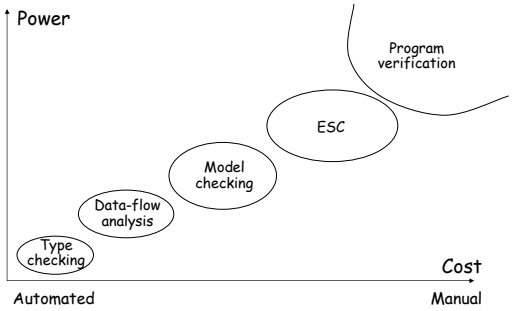
ESC Java

Static Analysis Spectrum



Is This Program Correct?

```
int square(int n) {
   int k = 0, r = 0, s = 1;
   while(k != n) {
      r = r + s; s = s + 2; k = k + 1;
   }
   return r;
}
```

- Type checking not enough to check this
  - Neither is data-flow analysis, nor model checking

Program Verification

- Program verification is the most powerful static analysis method
  - Can reason about all properties of programs
- Cannot fully automate
- But …
  - Can automate certain parts (ESC/Java)
  - Teaches how to reason about programs in a systematic way

Specifying Programs

- Before we check a program we must specify what it does
- We need formal specifications
  - English comments are not enough
- We use logic notation
  - Theory of pre- and post-conditions

State Predicates

- A predicate is a boolean expression on the program state (e.g., variables, object fields)
- Examples:
  - x == 8
  - x < y
  - true
  - false
  - (∀i. 0 <= i < a.length ⇒ a[i] >= 0)

Using Predicates to Specify Programs

- We focus first on how to specify a statement
- Hoare triple for statement S

precondition → { P } S { Q } ← postcondition

- Says that if S is started in a state that satisfies P, and S terminates, then it terminates in Q
  - This is the liberal version, which doesn't care about termination
  - Strict version: if S is started in a state that satisfies P then S terminates in Q

Hoare Triples. Examples.

- { true } x = 12 { x == 12 }
- { y >= 0 } x = 12 { x == 12 }
- { true } x = 12 { x >= 0 }

  (Programs satisfy many possible specifications)

- { x < 10 } x = x + 1 { x < 11 }
- { n >= 0 } x = fact(n) { x == n ! }
- { true } a = 0; if(x != 0) { a = 2 * x; } { a == 2*x }

Computing Hoare Triples

- We compute the triples using rules
  - One rule for each statement kind
  - Rules for composed statements

Assignment

- Assignment is the simplest operation and the trickiest one to reason about !
- { y >= 2 } x = 5 { ? }
- { x == y } x = x + 1 { ? }
- { ? } x = 5 { x == y }
- { ? } x = x + 1 { x == y }
- { ? } x = x + 1 { $x^2 + y^2 == z^2$ }
- { $x^2 + y^2 == z^2$ } x = x + 1 { ? }

Assignment Rule

- Rule for assignment

{ Q[x := E] } x = E { Q }

Q with x replaced by E

- Examples:
  - { 12 == 12 } x = 12 { x == 12 }

    x == 12 with x replaced by 12
  - { 12 >= 0 } x = 12 { x >= 0 }
  - { ? } x = x + 1 { x >= 0 }
  - { x >= 1 } x = x + 1 { ? }

Relaxing Specifications

- Consider { x >= 1 } x = x + 1 { x >= 2 }
  - It is very tight specification. We can relax it

{ P } if P ⇒ Q[x:=E]

x = E

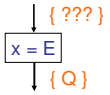
{ Q }

- Example: { x >= 5 } x = x + 1 { x >= 2 }

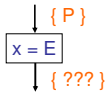
  (since x >= 5 ⇒ x + 1 >= 2)

Assignments: forward and backward

- Two ways to look at the rules:
  - Backward: given post-condition, what is pre-condition?

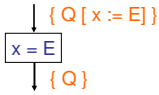


  - Forward: given pre-condition, what is post-condition?

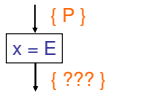


Assignments: forward and backward

- Two ways to look at the rules:
  - Backward: given post-condition, what is pre-condition?

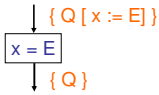


  - Forward: given pre-condition, what is post-condition?

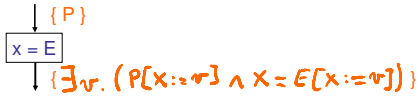


Assignments: forward and backward

- Two ways to look at the rules:
  - Backward: given post-condition, what is pre-condition?



  - Forward: given pre-condition, what is post-condition?



Example of running it forward

- { x == y } x = x + 1 { ? }

Example of running it forward

- { x == y } x = x + 1 { ? }

$$\exists v.\ (v == y \wedge x == v + 1)$$

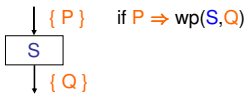
$$\Leftrightarrow \quad x == y + 1$$

Forward or Backward

- Forward reasoning
  - Know the precondition
  - Want to know what postconditon the code establishes
- Backward reasoning
  - Know what we want to code to establish
  - Must find in what precondition this happens
- Backward is used most often
  - Start with what you want to verify
  - Instead of verifying everything the code does

Weakest precondition

- wp(S, Q) is the weakest P such that { P } S { Q }
  - Order on predicates: Strong ⇒ Weak
  - wp returns the "best" possible predicate
- wp(x := E, Q) = Q[x := E]
- In general:

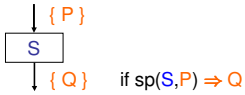


Weakest precondition

- This points to a verification algorithm:
  - Given function body annotated with pre-condition P and post-condition Q:
    - Compute wp of Q with respect to functon body
    - Ask a theorem prover to show that P implies the wp
- The wp function we will use is liberal (P does not guarantee termination)
  - If using both strict and liberal in the same context, the usual notation is wlp the liberal version and wp for the strict one

Strongest precondition

- sp(S, P) is the strongest Q such that { P } S { Q }
  - Recall: Strong ⇒ Weak
  - sp returns the "best" possible predicate
- sp(x := E, P) = …
- In general:

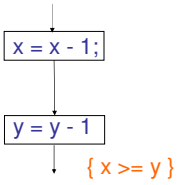


Strongest postcondition

- Strongest postcondition and weakest preconditions are symmetric
- This points to an equivalent verification algorithm:
  - Given function body annotated with pre-condition P and post-condition Q:
    - Compute sp of P with respect to functon body
    - Ask a theorem prover to show that the sp implies Q

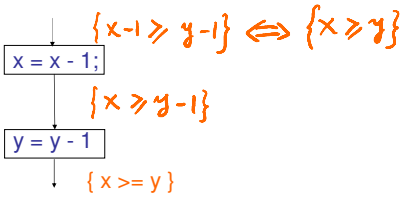
Composing Specifications

- If { P } S_1 { R } and { R } S_2 { Q }
  then { P } S_1; S_2 { Q }
- Example:



Composing Specifications

- If { P } S_1 { R } and { R } S_2 { Q }
  then { P } S_1; S_2 { Q }
- Example:

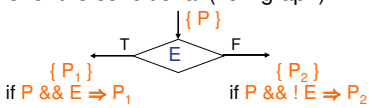


In terms of wp and sp

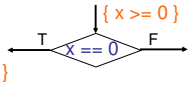
- wp(S_1; S_2 , Q) = wp(S_1,wp(S_2, Q))
- sp(S_1; S_2, P) = sp(S_2,sp(S_1, P))

Conditionals

- Rule for the conditional (flow graph)



- Example:

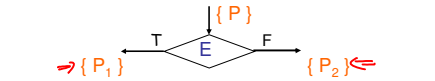


{ x == 0 } since x >= 0 && x == 0 ⇒ x == 0

{ x >= 1 } since x >= 0 && x != 0 ⇒ x >= 1

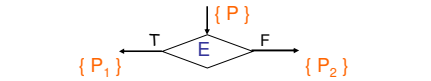
Conditionals: Forward and Backward

- Recall: rule for the conditional



Conditionals: Forward and Backward

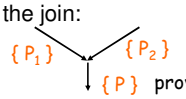
- Recall: rule for the conditional



- Forward: given P, find P_1 and P_2
  - pick P_1 to be P && E, and P_2 to be P && ! E
- Backward: given P_1 and P_2, find P
  - pick P to be (P_1 && E) || (P_2 && ! E)
  - Or pick P to be (E ⇒ P_1) && (! E ⇒ P_2)

Joins

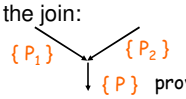
- Rule for the join:



{ P } provided P_1 ⇒ P and P_2 ⇒ P

Joins

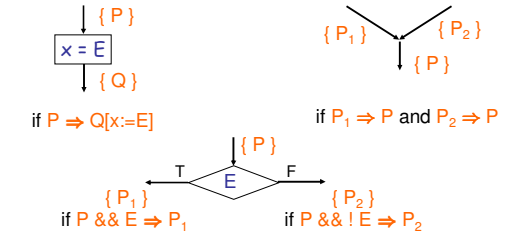
- Rule for the join:



{ P } provided P_1 ⇒ P and P_2 ⇒ P

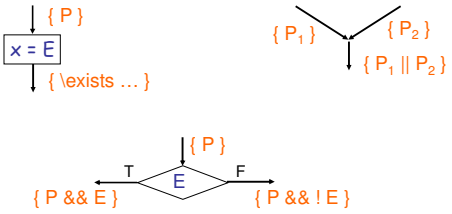
- Forward: pick P to be P_1 || P_2
- Backward: pick P_1, P_2 to be P

Review

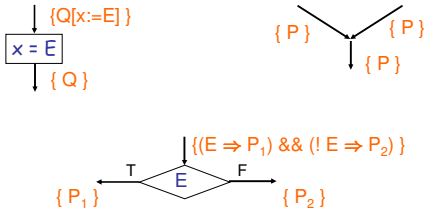


Implication is always in the direction of the control flow

Review: forward

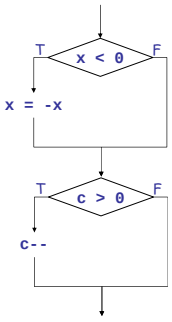


Review: backward

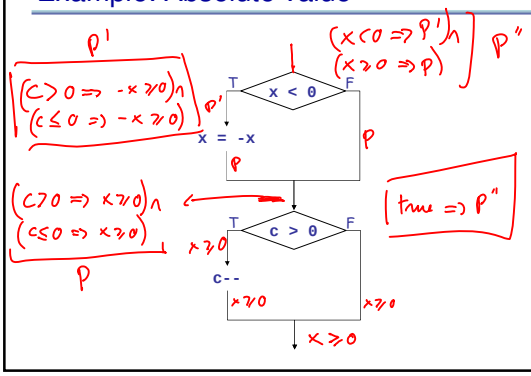


Example: Absolute value

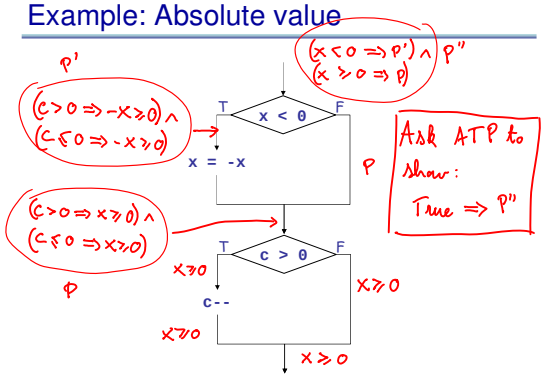
```
static int abs(int x)
//@ ensures \result >= 0
{
   if (x < 0) {
      x = -x;
   }
   if (c > 0) {
      c--;
   }
   return x;
}
```



Example: Absolute value



Example: Absolute value



In Simplify

```
>       (IMPLIES TRUE
         (AND (IMPLIES (< x 0)
                       (AND (IMPLIES (> c 0) (>= (- 0 x) 0))
                            (IMPLIES (<= c 0) (>= (- 0 x) 0))))
              (IMPLIES (>= x 0)
                       (AND (IMPLIES (> c 0) (>= x 0))
                            (IMPLIES (<= c 0) (>= x 0))))))
1: Valid.

>
```

So far…

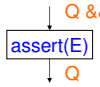
- Framework for checking pre and post conditions of computations without loops
- Suppose we want to check that some condition holds *inside* the computation, rather than *at the end*

Say we want to check that x > 0 here

```
static int abs(int x) {
   if (x < 0) {
      x = -x;
   }
   if (c > 0) {
      c--;
   }
   return x;
}
```

Asserts

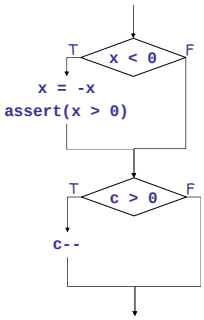
- { P } assert(E) { Q } if P ⇒ E && P ⇒ Q
- Backward: wp(assert(E), Q) = Q && E



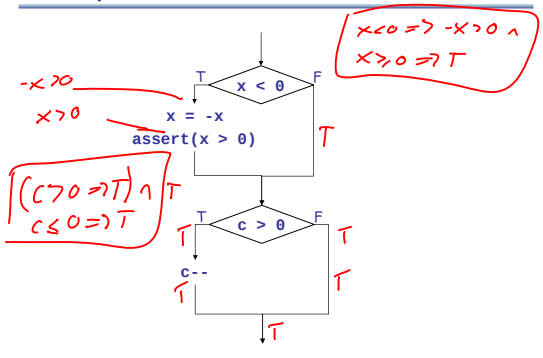
- Forward: sp(assert(E), P) = ???

Example: Absolute value with assert

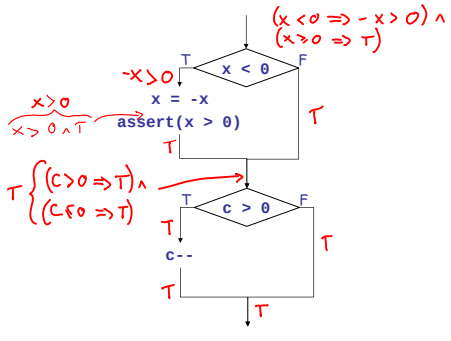
```
static int abs(int x)
{
   if (x < 0) {
      x = -x;
      assert(x > 0);
   }
   if (c > 0) {
      c--;
   }
   return x;
}
```



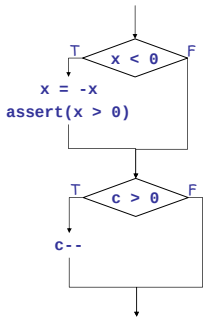
Example: Absolute value with assert



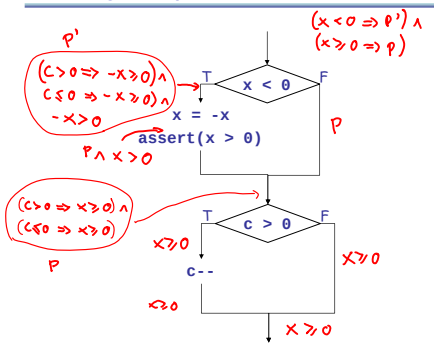
Example: Absolute value with assert



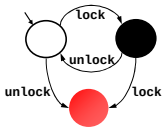
Adding the postcondition back in



Adding the postcondition back in



Another Example: Double Locking



*"An attempt to re-acquire an acquired lock or release a released lock will cause a **deadlock**."*

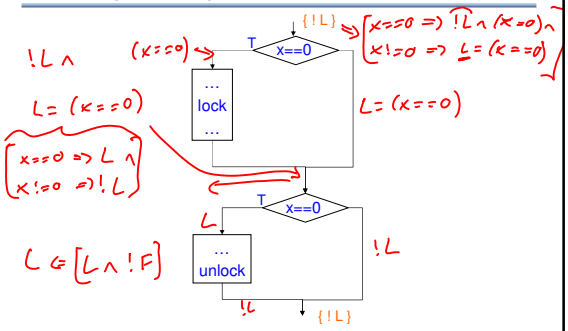
Calls to **lock** and **unlock** must **alternate**.

Locking Rules

- We assume that the boolean predicate locked says if the lock is held or not
- { ! locked && P[locked := true] } lock { P }
  - lock behaves as assert(! locked); locked = true
- { locked && P[locked := false] } unlock { P }
  - unlock behaves as assert(locked); locked = false

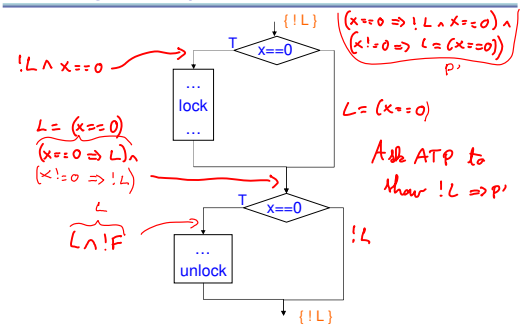
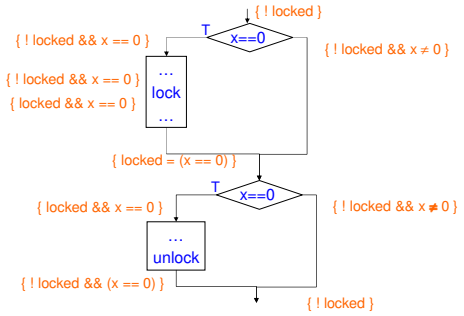
Locking Example

{ ! L && P[L := true] } lock { P }
{ L && P[L := false] } unlock { P }



Locking Example

{ ! L && P[L := true] } lock { P }
{ L && P[L := false] } unlock { P }



Locking Example: forward direction