ESC Java
Static Analysis Spectrum

Power

Cost

Automated

Manual

- Type checking
- Data-flow analysis
- Model checking
- ESC
- Program verification
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 $S$

  \[
  \text{precondition} \quad \{ P \} \quad S \quad \{ Q \} \quad \text{postcondition}
  \]

- 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.

• \( \{ \text{true} \} \ x = 12 \ \{ \ x == 12 \} \)

• \( \{ \ y \geq 0 \} \ x = 12 \ \{ \ x == 12 \} \)

• \( \{ \text{true} \} \ x = 12 \ \{ \ x \geq 0 \} \)

  (Programs satisfy many possible specifications)

• \( \{ \ x < 10 \} \ x = x + 1 \ \{ \ x < 11 \} \)

• \( \{ \ n \geq 0 \} \ x = \text{fact}(n) \ \{ \ x == n ! \} \)

• \( \{ \text{true} \} \ a = 0; \text{if}(x != 0) \ { \ a = 2 \ * \ x; } \ \{ \ a == 2 \*x \} \)
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 \geq 2 \} \ x = 5 \ {?} \ x = 5
  
  • \{ x = y \} \ x = x + 1 \ {?} \ x = y+1
  
  • \{ ? \} \ x = 5 \ {x = y} \ y = 5
  
  • \{ ? \} \ x = x + 1 \ {x = y} \ y = x-1
  
  • \{ ? \} \ 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 \( \{ x \geq 1 \} \ x = x + 1 \ \{ x \geq 2 \} \)
  - It is a very tight specification. We can relax it

\[
\begin{align*}
\{ P \} & \quad \text{if } P \Rightarrow Q[x:=E] \\
\text{if } x = E & \\
\{ Q \} &
\end{align*}
\]

- Example: \( \{ x \geq 5 \} \ x = x + 1 \ \{ x \geq 2 \} \)
  (since \( x \geq 5 \Rightarrow x + 1 \geq 2 \))
Assignments: forward and backward

- Two ways to look at the rules:
  - Backward: given post-condition, what is pre-condition?
    - \( x = E \)
    - \{ Q \}
  - Forward: given pre-condition, what is post-condition?
    - \{ P \}
    - \( x = E \)
    - \{ ??? \}
Assignments: forward and backward

- Two ways to look at the rules:
  - Backward: given post-condition, what is pre-condition?
    
    \[
    x = E \downarrow \{ Q \}
    \]
    
    \[
    x = E \downarrow \{ Q \}
    \]
  
  - Forward: given pre-condition, what is post-condition?
    
    \[
    x = E \downarrow \{ P \}
    \]
    
    \[
    x = E \downarrow \{ \lor \exists v. (P(x \rightarrow v) \land x = E(x \rightarrow v)) \}
    \]
Assignments: forward and backward

• Two ways to look at the rules:
  – Backward: given post-condition, what is pre-condition?
    \[
    \begin{align*}
    &\{ Q \left[ x := E \right] \} \\
    &x = E \\
    &\{ Q \}
    \end{align*}
    \]
  – Forward: given pre-condition, what is post-condition?
    \[
    \begin{align*}
    &\{ P \} \\
    &x = E \\
    &\{ \exists v. \left( P\left[ x := v \right] \land x = E\left[ x := v \right] \right) \}
    \end{align*}
    \]
Example of running it forward

- \{ x == y \} \textcolor{orange}{x = x + 1} \{ ? \} x == y + 1

\exists v. \left( v == y \land x == v + 1 \right)
Example of running it forward

- \( \{ x == y \} x = x + 1 \{ ? \} \)

\[ \exists \nu. (\nu == y \land x == \nu + 1) \]

\[ \iff \quad x == y + 1 \]
Forward or Backward

• Forward reasoning
  – Know the precondition
  – Want to know what postcondition 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

• $\text{wp}(S, Q)$ is the weakest $P$ such that $\{ P \} S \{ Q \}$
  – Order on predicates: Strong $\Rightarrow$ Weak
  – $\text{wp}$ returns the “best” possible predicate

• $\text{wp}(x := E, Q) = Q[x := E]$

• In general:
  $\text{wp}(S, Q)$
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 function 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 postcondition

• \( sp(S, P) \) is the strongest \( Q \) such that \( \{ P \} S \{ Q \} \)
  – Recall: Strong \( \Rightarrow \) Weak
  – \( sp \) returns the “best” possible predicate

• \( sp(x := E, P) = \ldots \)

• In general:

\[
\begin{array}{c}
\{ P \} \\
S \\
\{ Q \} \quad \text{if} \ sp(S, P) \Rightarrow Q
\end{array}
\]
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 function 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:

\[
\begin{align*}
    & x = x - 1; \\
    & y = y - 1 \\
    & \{ x \geq y \}
\end{align*}
\]
Composing Specifications

• If \{ P \} S_1 \{ R \} and \{ R \} S_2 \{ Q \}
then \{ P \} S_1; S_2 \{ Q \}

• Example:

\[
\begin{align*}
\{ x-1 \geq y-1 \} & \iff \{ x \geq y \} \\
x = x - 1; & \\
y = y - 1 & \{ x \geq y \}
\end{align*}
\]
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)

```
if P && E  ⊨  P

{ P }
T
E
F
{ P₁ }
if P && E  ⊨  P₁
{ P₂ }
if P && ! E  ⊨  P₂
```

• Example:

```
if x == 0  ⊨  x >= 0

{x >= 0}
T
x == 0
F
{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

\[
\begin{align*}
\{ P \} & \quad \text{provided } P \land E \Rightarrow P_1 \\
\{ P \} & \quad \text{provided } P \land \lnot E \Rightarrow P_2
\end{align*}
\]

• Forward: given \( P \), find \( P_1 \) and \( P_2 \)
  - pick \( P_1 \) to be \( P \land E \), and \( P_2 \) to be \( P \land \lnot E \)

• Backward: given \( P_1 \) and \( P_2 \), find \( P \)
  - pick \( P \) to be \( (P_1 \land E) \lor (P_2 \land \lnot E) \)
  - Or pick \( P \) to be \( (E \Rightarrow P_1) \land (!E \Rightarrow P_2) \)
Joins

• Rule for the join:

\[
\begin{align*}
\{ P_1 \} \quad \{ P_2 \} & \quad \{ P \} \\
\text{providing } P_1 \Rightarrow P & \text{ and } P_2 \Rightarrow P
\end{align*}
\]

• Forward: pick \( P \) to be \( P_1 \parallel P_2 \)

• Backward: pick \( P_1, P_2 \) to be \( P \)
Implication is always in the direction of the control flow
Review: forward

\[ x = E \]

\{ P \}

\{ \exists \ldots \}
Review: backward

\[\begin{align*}
x = E & \quad \{Q[x:=E]\} \\
E & \quad \{P\} \\
E & \quad \{P\} \\
T & \quad \{(E \Rightarrow P_1) \land \neg(E \Rightarrow P_2)\} \\
F & \quad \{P_1\} \\
F & \quad \{P_2\}
\end{align*}\]
Example: Absolute value

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

```
T   x < 0  F
    x = -x
    x > 0
T   c > 0  F
    c--
    x > 0
```
Example: Absolute value

\(x < 0\)

\(x = -x\)

\(c > 0\)

\(c--\)

\((x < 0 \Rightarrow p', \land)\) \(\Rightarrow wp\)

\(c := c - 1\)

\(x > 0\)

\(x := 0\)
Example: Absolute value

\[ x < 0 \]
\[ x = -x \]

\[ (c > 0 \Rightarrow -x \geq 0) \land (c < 0 \Rightarrow x > 0) \]

\[ (x < 0 \Rightarrow p') \land (x \geq 0 \Rightarrow p) \]

\[ (c > 0 \Rightarrow x \geq 0) \land (c < 0 \Rightarrow x > 0) \]

Ask ATP to show:
\[ \text{True} \Rightarrow p'' \]
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 \textit{inside} the computation, rather than \textit{at the end}

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

Say we want to check that $x > 0$ here
Asserts

- \{ Q \land E \} \text{assert}(E) \{ Q \}

- Backward: \text{wp}(\text{assert}(E), Q) = Q \land E

- Forward: \text{sp}(\text{assert}(E), P) = ???
Example: Absolute value with assert

```c
static int abs(int x) {
    if (x < 0) {
        x = -x;
        assert(x > 0);
    }
    if (c > 0) {
        c--;
    }
    return x;
}
```
Example: Absolute value with assert

\[ x < 0 \]
\[ x = -x \]
\[ \text{assert}(x > 0) \]

\[ c > 0 \]
\[ c-- \]
Example: Absolute value with assert

\[(x < 0 \implies -x > 0) \land (x \geq 0 \implies T)\]

- \(x < 0\)
  - \(-x > 0\)
    - \(x > 0\)
  - \(x = -x\)
- \(x \geq 0\)
  - \(c > 0\)
    - \(c--\)
  - \(c < 0\)
    - \(T\)

assert \(x > 0\)
Adding the postcondition back in

T
\[x < 0\]
F
\[x = -x\]
assert(\(x > 0\))

T
\[c > 0\]
F
\[c--\]
Adding the postcondition back in

\[ x < 0 \quad x = -x \quad \text{assert}(x > 0) \]

\[ c > 0 \quad c-- \quad x > 0 \]

\[ (c > 0 \Rightarrow x > 0) \land (c \leq 0 \Rightarrow x > 0) \land -x > 0 \]

\[ (x < 0 \Rightarrow \neg P') \land (x \geq 0 \Rightarrow P) \]

\[ P \land x > 0 \]
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] \}_\text{lock} \{ P \}
  – lock behaves as \text{assert(! locked); locked = true}

• \{ locked && P[locked := false] \}_\text{unlock} \{ P \}
  – unlock behaves as \text{assert(locked); locked = false}
Locking Example

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

\{ L \&\& P[L := false] \} unlock \{ P \}

```
if (P != null) {
    assert (P == null)
    f(...)
}
```

```
... lock ...

T

x==0

... unlock

{ !L }
Locking Example

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

{ L && P[L := false] } unlock { P }

{ !L }
Other challenges

• Loops
  – verifying loops requires loop invariants
  – inferring good invariants is hard

• Procedures
  – annotate procedures with pre and post conditions
  – “plug” pre/post conditions into caller

• Pointers
  – update the weakest pre-condition rules to handle pointers
Discussion
ESC/Java summary

• Very general verification framework
  – Based on pre- and post-conditions

• Generate VC from code
  – Instead of modelling the semantics of the code inside the theorem prover

• Loops and procedures require user annotations
  – But can try to infer these