ESC Java

Is This Program Correct?

```java
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
  - (\forall i. 0 <= i < a.length \Rightarrow 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.

• $\{ \text{true} \}$ $x = 12$ $\{ x == 12 \}$
• $\{ y >= 0 \}$ $x = 12$ $\{ x == 12 \}$
• $\{ \text{true} \}$ $x = 12$ $\{ x >= 0 \}$
  (Programs satisfy many possible specifications)

• $\{ x < 10 \}$ $x = x + 1$ $\{ x < 11 \}$
• $\{ n >= 0 \}$ $x = \text{fact}(n)$ $\{ x == n! \}$
• $\{ \text{true} \}$ $a = 0$; 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!

  • $\{ y >= 2 \}$ $x = 5$ $\{ ? \}$
  • $\{ x == y \}$ $x = x + 1$ $\{ ? \}$ $x >= y + 1$
  • $\{ ? \}$ $x = 5$ $\{ x == y \}$
  • $\{ ? \}$ $x = x + 1$ $\{ x == y \}$ $\forall x = x$
  • $\{ ? \}$ $x = x + 1$ $\{ x^2 + y^2 == z^2 \}$
  • $\{ x^2 + y^2 == z^2 \}$ $x = x + 1$ $\{ ? \}$

Assignment Rule

• Rule for assignment

  $\begin{align*}
  \{ \text{Q}(x := E) \} & \quad x = E \quad \{ \text{Q} \} \\
  \text{Q with } x \text{ replaced by } E
  \end{align*}$

• Examples:

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

Relaxing Specifications

• Consider $\{ x >= 1 \}$ $x = x + 1$ $\{ x >= 2 \}$

  – It is very tight specification. We can relax it

  $\begin{align*}
  \{ \text{P} \} & \quad \text{if } P \Rightarrow \text{Q}(x := E) \\
  \{ x = E \} & \quad \{ \text{Q} \}
  \end{align*}$

• Example: $\{ x >= 5 \}$ $x = x + 1$ $\{ x >= 2 \}$

  (since $x >= 5 \Rightarrow x + 1 >= 2$)
Assignments: forward and backward

- Two ways to look at the rules:
  - Backward: given post-condition, what is pre-condition?
    \[
    \begin{align*}
    x = E & \quad \{ ??? \} \\
    x = E & \quad \{ Q \}
    \end{align*}
    \]
  - Forward: given pre-condition, what is post-condition?
    \[
    \begin{align*}
    x = E & \quad \{ P \} \\
    x = E & \quad \{ ??? \}
    \end{align*}
    \]

Example of running it forward

- \((x \equiv y) x = x + 1 (\ ?)\)
  
  \[\exists v. (v = y \land x = x + 1)\]
  \[\Leftrightarrow x = y + 1\]

Forward or Backward

- Forward reasoning
  - Know the pre-condition
  - Want to know what post-condition the code establishes

- Backward reasoning
  - Know what we want the code to establish
  - Must find in what pre-condition 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:

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

Strongest postcondition

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

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

Composing Specifications

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

\[
\begin{array}{c}
\{ x = x - 1; \} \\
\{ y = y - 1 \} \\
\{ x \geq y \}
\end{array}
\]

Weakest precondition

- This points to a verification algorithm:
  - Given function body annotated with pre-condition \( P \) and post-condition \( Q \):
    - Compute \( \text{wp} \) of \( Q \) with respect to function body
    - Ask a theorem prover to show that \( P \) implies the \( \text{wp} \)
- The \( \text{wp} \) 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 \( \text{wp} \) the liberal version and \( \text{wp} \) for the strict one

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 \( \text{sp} \) of \( P \) with respect to function body
    - Ask a theorem prover to show that the \( \text{sp} \) implies \( Q \)

Composing Specifications

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

\[
\begin{array}{c}
\{ x = x - 1; \} \\
\{ x \geq y; y = y - 1 \} \\
\{ x \geq y \}
\end{array}
\]

\[
\begin{array}{c}
\{ x = x - 1; \} \\
\{ x \geq y; y = y - 1 \} \\
\{ x \geq y \}
\end{array}
\]
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)

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

- Example:

\[
\begin{align*}
\{x = 0\} &\text{ if } P \land E \Rightarrow \{P_1\} \\
\{x = 1\} &\text{ if } P \land \lnot E \Rightarrow \{P_2\} \\
\end{align*}
\]

since \(x > 0 \land x = 0 \Rightarrow x = 0\) since \(x > 0 \land x > 0 \Rightarrow x > 1\)

Conditionals: Forward and Backward

- Recall: rule for the conditional

\[
\begin{align*}
\text{if } P \Rightarrow P_1 \\
\text{if } P \Rightarrow P_2 \\
\end{align*}
\]

\[
\text{provided } P \land E \Rightarrow P_1 \quad \text{provided } P \land \lnot E \Rightarrow P_2
\]

- 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 \((P_1 \land \lnot E) \lor (P_2 \land \lnot E)\)
  - Or pick \(P\) to be \((E \Rightarrow P_1) \land (\lnot E \Rightarrow P_2)\)

Joins

- Rule for the join:

\[
\begin{align*}
\{P_1\} &\lor \{P_2\} \Rightarrow P \\
\end{align*}
\]

\[
\text{provided } P_1 \Rightarrow P \text{ and } P_2 \Rightarrow P
\]

- Forward: pick \(P\) to be \(P_1 \lor P_2\)

- Backward: pick \(P_1, P_2\) to be \(P\)

Review

- \(\begin{cases} P \\ x = E \end{cases} \implies \{Q\}\)

- \(\begin{cases} P_1 \\ P_2 \end{cases} \implies \{P\}\)

- \(\begin{cases} P_1 \\ P_2 \end{cases} \implies \{P\}\)

- \(\begin{cases} P \land E \Rightarrow P_1 \\ P \land \lnot E \Rightarrow P_2 \end{cases}\)

Implication is always in the direction of the control flow

Review: forward

- \(\begin{cases} P \\ x = E \end{cases} \implies \exists \{Q\}\)

- \(\begin{cases} P_1 \\ P_2 \end{cases} \implies \{P\}\)

- \(\begin{cases} P \land E \Rightarrow P_1 \\ P \land \lnot E \Rightarrow P_2 \end{cases}\)
Review: backward

\[ \{ Q(x := E) \} \]

\[ \{ P \} \]

\[ \{ P \} \]

\[ (E \Rightarrow P_1) \&\& (! E \Rightarrow P_2) \]

\[ \{ P_1 \} \]

\[ \{ P_2 \} \]

Example: Absolute value

\[ \text{static int abs(int x) } \]

\[ //\text{ ensures } \{ \text{result} \geq 0 \} \]

\[ \{
\text{if (x < 0) } \{
\quad x = -x;
\}\}
\]

\[ \text{if (c > 0) } \{
\quad c--;
\}\]

\[ \text{return x; } \]

Example: Absolute value

Example: Absolute value

Example: Absolute value

In Simplify

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

\[ \text{AND (IMPLIES (>= x 0) (>= x 0)))
\]

\[ \text{AND (IMPLIES (= x 0) (>= x 0)))
\]

\[ \text{AND (IMPLIES (= x 0) (>= x 0)))
\]

\[ \text{AND (IMPLIES (= x 0) (>= x 0)))
\]

\[ 1: \text{Valid.} \]

\[ > \]

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

\[ \text{static int abs(int x) } \]

\[ \text{if (x < 0) } \{
\quad x = -x;
\}\]

\[ \text{if (c > 0) } \{
\quad c--;
\}\]

\[ \text{return x; } \]
Asserts

- \{ Q \land E \} \ assert(E) \{ Q \}
- Backward: wp(assert(E), Q) = Q \land E
  
  \[ \begin{align*}
  &\text{assert}(E) \\
  &\downarrow \\
  &Q
  \end{align*} \]
- Forward: sp(assert(E), P) = ???
  
  \[ \begin{align*}
  &\text{assert}(E) \\
  &\downarrow \\
  &???
  \end{align*} \]

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;
}
```

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
  - \( \{ ! \text{locked} \land P[\text{locked} := \text{true}] \} \) lock \( \{ P \} \)
    - lock behaves as assert(! locked); locked = true
  - \( \{ \text{locked} \land P[\text{locked} := \text{false}] \} \) unlock \( \{ P \} \)
    - unlock behaves as assert(locked); locked = false

Locking Example

\[ \{ ! \text{L} \land P[\text{L} := \text{true}] \} \] lock \( \{ P \} \)
\[ \{ \text{L} \land P[\text{L} := \text{false}] \} \] unlock \( \{ P \} \)

In Simplify

\[
\begin{aligned}
&\text{assert (isLocked L) (EQ L locked))} \\
&\text{IMPLIES (NOT (isLocked L)) (AND (IMPLIES (EQ x 0) (AND (NOT (isLocked L)) (AND (IMPLIES (EQ x 0) (AND (isLocked L) (NOT FALSE))) (IMPLIES (NEQ x 0) (NOT (AND (isLocked L) (NOT FALSE))))) (IMPLIES (NEQ x 0) (AND (isLocked L) (NOT FALSE))))) (IMPLIES (NEQ x 0) (AND (isLocked L) (NOT FALSE))))})
\end{aligned}
\]
Reasoning About Programs with Loops

- Loops can be handled using conditionals and joins
- Consider the \( \text{while}(E) S \) statement

\[
\begin{align*}
\text{S} & \quad \text{T} & \quad \text{E} & \quad \text{F} \\
\{ I & \} & \{ I \} & \{ I && E \} & \\
(1) & \quad (P) & \quad \text{P} & \quad \text{Q}
\end{align*}
\]

if (1) \( P \Rightarrow I \) (loop invariant holds initially)
and (2) \( I && E \Rightarrow Q \) (loop establishes the postcondition)
and (3) \( (I && E) S (I) \) (loop invariant is preserved)

Loops in the backward direction

- Given \( Q \), want to find weakest invariant \( I \) that will establish (2) and (3), then pick \( P \) to be \( I \)
- Finding weakest \( I \) is:
  - Undecidable in theory
  - Hard in practice

Loops in the forward direction

- Given \( P \), want to find strongest invariant \( I \) that will establish (1) and (3), then pick \( Q \) to be \( I && E \)
- Again, finding \( I \) is hard

Loop Example

- Let's verify \( \{ x == 8 && y == 16 \} \text{while}(x > 0) \{ x --; y -= 2; \} \{ y == 0 \} \)
  - Is this true?
  - We must find an appropriate invariant \( I \)
    - Try one that holds initially \( x == 8 && y == 16 \)
    - Try one that holds at the end \( y == 0 \)

Loop Example (II)

- Guess the invariant \( y == 2^x \)
  - Initial: \( x == 8 \) \& \( y == 16 \) \( \Rightarrow y == 2^x \)
  - Preservation: \( y == 2^x \) \& \( x > 0 \) \( \Rightarrow y - 2 == 2^x \) \( (x - 1) \)
  - Final: \( y == 2^x \) \& \( x <= 0 \) \( \Rightarrow y == 0 \)

Loop Example (III)

- Guess the invariant \( y == 2^x \& \& x >= 0 \)
  - Initial: \( x == 8 \) \& \( y == 16 \) \( \Rightarrow y == 2^x \& \& x >= 0 \)
  - Preservation:
    - \( y == 2^x \& \& x > 0 \Rightarrow y - 2 == 2^x \& \& x - 1 > 0 \)
    - Final: \( y == 2^x \& \& x >= 0 \& \& x <= 0 \Rightarrow y == 0 \)
Functions

• Consider a binary search function \texttt{bsearch}

\begin{verbatim}
int bsearch(int a[], int p) {
    \{ sorted(a) \}
    \ldots
    \{ r == -1 || (r >= 0 && r < a.length && a[r] == p) \}
    return res;
}\end{verbatim}

• The precondition and postcondition are the function specification
  – Also called a contract

Function Calls

• Consider a call to function \texttt{F(int in)}
  – With return variable \texttt{out}
  – With precondition \texttt{Pre}, postcondition \texttt{Post}

• Rule for function call:

\begin{verbatim}
\begin{array}{c}
\{ P \}
\text{if } P \Rightarrow Pre[in:=E] \\
y = F(E) \\
\{ Q \}
\text{and } Post[out := y, in := E] \Rightarrow Q
\end{array}
\end{verbatim}

Function Call Example

• Consider the call

\begin{verbatim}
{ sorted(array) }
y = bsearch(array, 5)
if( y != -1) {
    { array[y] == 5 }
}
\end{verbatim}

• Show \texttt{Post[r := y, a := array, p := 5]}
  \Rightarrow array[y] == 5
  – Need to know \texttt{y != -1}!

• Show \texttt{sorted(array) \Rightarrow Pre[a := array]}

Function Calls: backward

• Consider a call to function \texttt{F(int in)}
  – With return variable \texttt{out}
  – With precondition \texttt{Pre}, postcondition \texttt{Post}

\begin{verbatim}
\begin{array}{c}
\{ Q \}
\text{and } Post[out := y, in := E] \Rightarrow Q
\end{array}
\end{verbatim}

Pointers and aliasing

\begin{verbatim}
\begin{array}{c}
\{ ??? \}
\text{x := y := 5} \\
x = y = 1 \\
\{ x := 5 \}
\end{array}
\end{verbatim}
Pointers and aliasing

\[
x = \ast y + 1 \quad \{ \ast y = 4 \}
\]

Regular rule worked in this case!

Example where regular rule doesn’t work

\[
x = \ast y + 1 \quad \{ \ast y = 4 \}
\]

Example where regular rule doesn’t work

\[
x = \ast y + 1 \quad \{ \ast y = 4 \}
\]

Example where regular rule doesn’t work

\[
x = \ast y + 1 \quad \{ \ast y = 4 \}
\]

Example where regular rule doesn’t work

\[
x = \ast y + 1 \quad \{ \ast y = 4 \}
\]

Pointer stores

\[
\ast x = y + 1 \quad \{ y == 5 \}
\]

Pointer stores

\[
\ast x = y + 1 \quad \{ y == 5 \}
\]
One solution

- Perform case analysis based on all the possible alias relationships between the LHS of the assignment and part of the postcondition
- Can use a static pointer analysis to prune some cases out
- However, exponentially many cases in the pointer analysis, which leads to large formulas.
- eg, how many cases here:

\[ \begin{align*}
    x &= y + a \\
    z &= v + b
\end{align*} \]

Another solution

- Up until now the program state has been implicit. Let’s make the program state explicit...
- A predicate is a function from program states to booleans.
- So for \( wp(S, Q) \), we have:
  - \( Q(\sigma) \) returns true if \( Q \) holds in \( \sigma \)
  - \( wp(S, Q)(\sigma) \) returns true if \( wp(S, Q) \) holds in \( \sigma \)

New formulation of \( wp \)

- Suppose \( step(S, \sigma) \) returns the program state resulting from executing \( S \) starting in program state \( \sigma \).

- Then we can express \( wp \) as follows:

\[ wp(S, Q)(\sigma) = Q(step(\sigma)) \]

Example in Simplify syntax

From previous slide: \( wp(S, Q)(\sigma) = Q(step(\sigma)) \)

\[ \begin{align*}
    x &= y + 1 \\
    \{ y = 5 \}
\end{align*} \]

- \( Q \) is: \( (EQ (\text{select } s y) 5) \)
- \( step(S, \sigma) \) is: \( (\text{store } s (\text{select } s x) (+ (\text{select } s y) 1)) \)
- \( wp(S, Q) \) is: \( (EQ (\text{select } s y) 5) \)

From previous slide: \( wp(S, Q)(\sigma) = Q(step(\sigma)) \)

\[ \begin{align*}
    x &= y + 1 \\
    \{ y = 5 \}
\end{align*} \]

- \( Q \) is: \( (EQ (\text{select } s y) 5) \)
- \( step(S, \sigma) \) is: \( (\text{store } s (\text{select } s x) (+ (\text{select } s y) 1)) \)
- \( wp(S, Q) \) is: \( (EQ (\text{select } s y) 5) \)
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