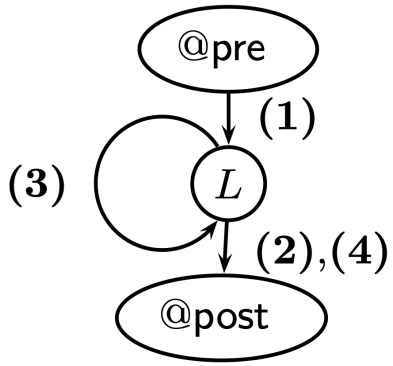
EC4219: Software Engineering

Lecture 11 — Program Verification (2)
Inductive Assertion Method

Sunbeom So
2024 Spring

Proving Partial Correctness

- A function is **partially correct** if its postcondition is satisfied when (1) the function's precondition is satisfied on entry and (2) the function returns.
- To prove the partial correctness, we use **inductive assertion method**.
- The inductive assertion method is based on mathematical induction.



- Four steps of the inductive assertion method:
  1. Break a function down into a finite set of basic paths.
  2. Derive verification conditions (VCs) from every basic path.
  3. Check the validity of VCs by an SMT solver.
  4. If all of the VCs are valid, the function is partially correct.

Basic Path

- A basic path is a sequence of **atomic statements** that
  - begins at the function precondition or a loop invariant, and
  - ends at a loop invariant or the function postcondition.
- Moreover, a loop invariant can only occur at the beginning or the ending of a basic path. That is, basic paths do not cross loops.

Basic Path

- A basic path is a sequence of **atomic statements** that
  - begins at the function precondition or a loop invariant, and
  - ends at a loop invariant or the function postcondition.
- Moreover, a loop invariant can only occur at the beginning or the ending of a basic path. That is, basic paths do not cross loops.

Q. Why basic paths? Why not just consider every possible path instead of basic paths?

Basic Path

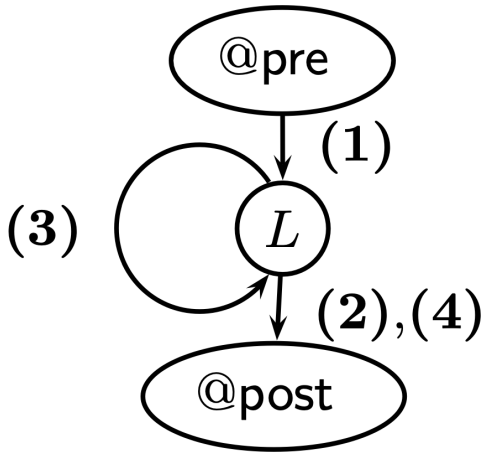
- A basic path is a sequence of **atomic statements** that
  - begins at the function precondition or a loop invariant, and
  - ends at a loop invariant or the function postcondition.
- Moreover, a loop invariant can only occur at the beginning or the ending of a basic path. That is, basic paths do not cross loops.

Q. Why basic paths? Why not just consider every possible path instead of basic paths?
A. Loops and recursive functions complicate proofs as they create an unbounded number of paths.

Basic Path Example 1: Linear Search

@pre: $0 \leq l \wedge u < |a|$
@post: $rv \leftrightarrow \exists i.l \leq i \leq u \wedge a[i] = e$

```
bool LinearSearch (int a[], int l, int u, int e) {
  int i := l;
  while
  @L : l ≤ i ∧ (∀j.l ≤ j < i → a[j] ≠ e)
  (i ≤ u) {
    if (a[i] = e) return true;
    i := i + 1;
  }
  return false;
}
```



(1)
@pre: $0 \leq l \wedge u < |a|$
$i := l;$
@$L : l \leq i \wedge (\forall j.l \leq j < i \rightarrow a[j] \neq e)$

(2)
@$L : l \leq i \wedge (\forall j.l \leq j < i \rightarrow a[j] \neq e)$
assume $i \leq u;$
assume $a[i] = e;$
$rv :=$ true;
@post: $rv \leftrightarrow \exists i.l \leq i \leq u \wedge a[i] = e$

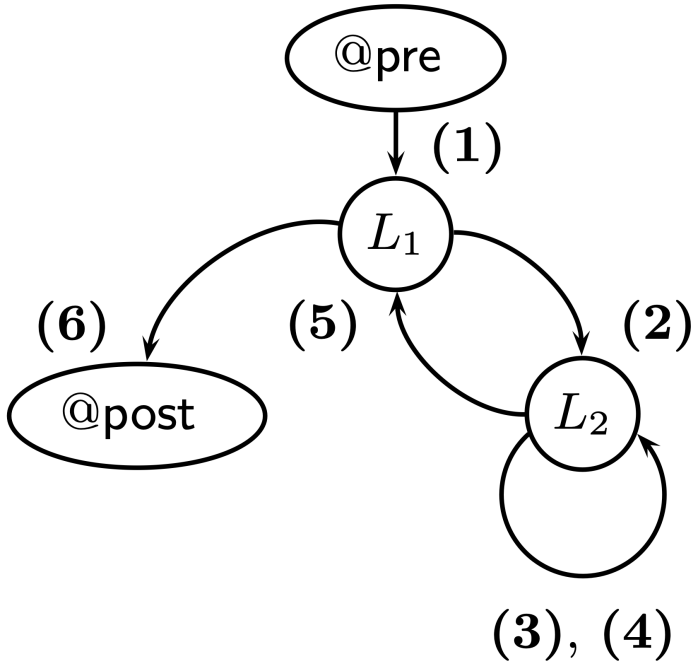
(3)
@$L : l \leq i \wedge (\forall j.l \leq j < i \rightarrow a[j] \neq e)$
assume $i \leq u;$
assume $a[i] \neq e;$
i := i+1;
@$L : l \leq i \wedge (\forall j.l \leq j < i \rightarrow a[j] \neq e)$

(4)
@$L : l \leq i \wedge (\forall j.l \leq j < i \rightarrow a[j] \neq e)$
assume $i > u;$
$rv :=$ false;
@post: $rv \leftrightarrow \exists i.l \leq i \leq u \wedge a[i] = e$

Basic Path Example 2: Bubble Sort

@pre: $\top$
@post: $\textbf{sorted}(rv, 0, |rv| - 1)$

```
bool BubbleSort (int a[]) {
  int[] a := a_0;
  @L_1 : { -1 ≤ i < |a|
         ∧ partitioned(a, 0, i, i + 1, |a| − 1)
         ∧ sorted(a, i, |a| − 1) }
  for (int i := |a| − 1; i > 0; i := i − 1) {
    @L_2 : { 1 ≤ i < |a| ∧ 0 ≤ j ≤ i
           ∧ partitioned(a, 0, i, i + 1, |a| − 1)
           ∧ partitioned(a, 0, j − 1, j, j)
           ∧ sorted(a, i, |a| − 1) }
    for (int j := 0; j < i; j := j + 1) {
      if (a[j] > a[j + 1]) {
        int t := a[j];
        a[j] := a[j + 1];
        a[j + 1] := t;
      }
    }
  }
  return a;
}
```



Basic Path Example 2: Bubble Sort

(1)
@pre: $\top$
$a := a_0;$
$i := |a| - 1;$
$@L_1 : -1 \leq i < |a| \wedge$ **partitioned**$(a, 0, i, i+1, |a|-1) \wedge$ **sorted**$(a, i, |a|-1)$

(2)
$@L_1 : -1 \leq i < |a| \wedge$ **partitioned**$(a, 0, i, i+1, |a|-1) \wedge$ **sorted**$(a, i, |a|-1)$
assume $i > 0;$
$j := 0;$
$@L_2 : \left\{ \begin{array}{l} 1 \leq i < |a| \wedge 0 \leq j \leq i \ \wedge \textbf{ partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge \textbf{ partitioned}(a, 0, j-1, j, j) \ \wedge \textbf{ sorted}(a, i, |a|-1) \end{array} \right\}$

(3) true branch of the inner loop
$@L_2 : \left\{ \begin{array}{l} 1 \leq i < |a| \wedge 0 \leq j \leq i \ \wedge \textbf{ partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge \textbf{ partitioned}(a, 0, j-1, j, j) \ \wedge \textbf{ sorted}(a, i, |a|-1) \end{array} \right\}$
assume $j < i;$
assume $a[j] > a[j+1];$
$t := a[j];$
$a[j] := a[j+1];$
$a[j+1] := t;$
$j := j + 1;$
$@L_2 : \left\{ \begin{array}{l} 1 \leq i < |a| \wedge 0 \leq j \leq i \ \wedge \textbf{ partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge \textbf{ partitioned}(a, 0, j-1, j, j) \ \wedge \textbf{ sorted}(a, i, |a|-1) \end{array} \right\}$

Basic Path Example 2: Bubble Sort

(4) false branch of the inner loop

$$@L_2 : \left\{ \begin{array}{l} 1 \le i < |a| \land 0 \le j \le i \ \land \ \textbf{partitioned}(a, 0, i, i+1, |a|-1) \\ \land \ \textbf{partitioned}(a, 0, j-1, j, j) \ \land \ \textbf{sorted}(a, i, |a|-1) \end{array} \right\}$$

assume $j < i$;

assume $a[j] \le a[j+1]$;

$j := j + 1$;

$$@L_2 : \left\{ \begin{array}{l} 1 \le i < |a| \land 0 \le j \le i \ \land \ \textbf{partitioned}(a, 0, i, i+1, |a|-1) \\ \land \ \textbf{partitioned}(a, 0, j-1, j, j) \ \land \ \textbf{sorted}(a, i, |a|-1) \end{array} \right\}$$

(5)

$$@L_1 : -1 \le i < |a| \land \ \textbf{partitioned}(a, 0, i, i+1, |a|-1) \land \ \textbf{sorted}(a, i, |a|-1)$$

assume $j \ge i$;

$i := i - 1$;

$$@L_1 : -1 \le i < |a| \land \ \textbf{partitioned}(a, 0, i, i+1, |a|-1) \land \ \textbf{sorted}(a, i, |a|-1)$$

(6)

$$@L_1 : -1 \le i < |a| \land \ \textbf{partitioned}(a, 0, i, i+1, |a|-1) \land \ \textbf{sorted}(a, i, |a|-1)$$

assume $i \le 0$;

$rv := a$;

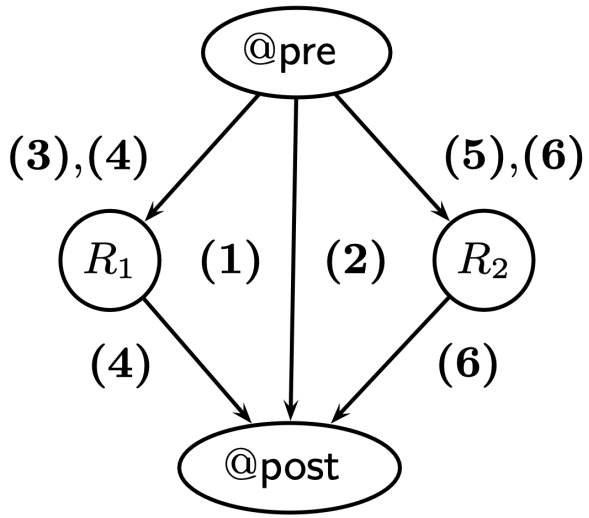
@post: $\textbf{sorted}(rv, 0, |rv|-1)$

Basic Paths: Function Calls

- Like loops, recursive function calls may create an unbounded number of paths.
  - For loops, loop invariants cut loops to produce a finite number of basic paths.
  - For function calls, we use function specifications to cut calls.
- The postcondition of a function summarizes the effects of calling the function, as it relates the return variable and the formal parameters.
  - So we can use these summaries to compute the effects of function calls.
- However, note that the function postcondition holds only when the precondition is satisfied on entry.
  - To ensure this, we generate an extra basic path that asserts the precondition, called function call assertion.

Basic Path Example 3: Binary Search

```
1   @pre: 0 ≤ l ∧ u < |a| ∧ sorted(a, l, u)
2   @post: rv ↔ ∃i.l ≤ i ≤ u ∧ a[i] = e
3   bool BinarySearch (int a[], int l, int u, int e) {
4     if (l > u) return false;
5     else {
6       int m := (l + u) div 2;
7       if (a[m] = e) return true;
8       else if (a[m] < e) return BinarySearch (a, m + 1, u, e)
9       else return BinarySearch (a, l, m − 1, e)
10    }
11  }
```



Exercise) Try by yourself!

Generating VCs

- To generate VCs for each basic path, we need to express the effects of program statements in terms of FOL formulas.
- The **weakest precondition** predicate transformer is the mechanism; it generates the most general one among valid preconditions.

$$\textbf{wp} : \textbf{FOL} \times \textbf{stmts} \rightarrow \textbf{FOL}$$

What is the precondition that must hold before the statement to ensure that the postcondition holds afterwards?

- $\{\quad\quad\quad\quad\quad\quad\}$ `x:=x+1` $\{x > 0\}$
- $\{\quad\quad\quad\quad\quad\quad\}$ `y:=2*y` $\{y < 5\}$
- $\{\quad\quad\quad\quad\quad\quad\}$ `x:=x+y` $\{y > x\}$
- $\{\quad\quad\}$ `assume` $(a \leq 5)$ $\{a \leq 5\}$
- $\{\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\}$ `assume` $(a \leq b)$ $\{a \leq 5\}$

Generating VCs

- To generate VCs for each basic path, we need to express the effects of program statements in terms of FOL formulas.
- The **weakest precondition** predicate transformer is the mechanism; it generates the most general one among valid preconditions.

$$\textbf{wp} : \textbf{FOL} \times \textbf{stmts} \rightarrow \textbf{FOL}$$

What is the precondition that must hold before the statement to ensure that the postcondition holds afterwards?

- $\{x + 1 > 0\}$ `x:=x+1` $\{x > 0\}$
- $\{2 * y < 5\}$ `y:=2*y` $\{y < 5\}$
- $\{y > x + y\}$ `x:=x+y` $\{y > x\}$
- $\{true\}$ `assume` $(a \leq 5)$ $\{a \leq 5\}$
- $\{a \leq b \rightarrow a \leq 5\}$ `assume` $(a \leq b)$ $\{a \leq 5\}$

Weakest Precondition Predicate Transformer

Weakest precondition $\mathbf{wp}(F, S)$ for statements S of basic paths:

- Assumption: What must hold before the $\texttt{assume}(c)$ is executed to ensure that F holds afterwards? If $c \rightarrow F$ holds before, then satisfying c guarantees that F holds afterwards:

$$\mathbf{wp}\ (F, \texttt{assume}(c)) \iff c \rightarrow F.$$

- Assignment: What must hold before the statement $v := e$ is executed to ensure that $F[v]$ holds afterward? If $F[e]$ holds before, then assining e to v makes $F[v]$ holds afterward:

$$\mathbf{wp}(F, v := e) \iff F[e]$$

- For a sequence of statements $S_1; \cdots ; S_n$, compute backwardly.

$$\mathbf{wp}(F, S_1; \cdots ; S_n) \iff \mathbf{wp}(\mathbf{wp}(F, S_n), S_1; \cdots , S_{n-1})$$

Verification Condition

The verification condition of a basic path

$$
\begin{array}{c}
@F \\
S_1; \\
\vdots \\
S_n; \\
@G
\end{array}
$$

is

$$F \rightarrow \mathbf{wp}(G, S_1; \cdots ; S_n)$$

The verification condition is sometimes denoted by the Hoare triple

$$\{F\} S_1; \cdots ; S_n \{G\}$$

Example 1: Verification Condition

Consider the basic path

$$
\begin{aligned}
&@x \geq 0 \\
&x := x + 1; \\
&@x \geq 1
\end{aligned}
$$

and its corresponding Hoare triple $\{x \geq 0\}\ x := x + 1\ \{x \geq 1\}$. The verification proceeds as follows:

$$
\begin{aligned}
& x \geq 0 \rightarrow \mathbf{wp}(x \geq 1, x := x + 1) \\
\iff\ & x \geq 0 \rightarrow x + 1 \geq 1 \\
\iff\ & \mathit{true}
\end{aligned}
$$

which is valid.

Example 2: Verification Condition

Consider the basic path (2) from the LinearSearch example.

$$@L : F : l \leq i \wedge (\forall j.l \leq j < i \rightarrow a[j] \neq e)$$
$$S_1 : \texttt{assume } i \leq u;$$
$$S_2 : \texttt{assume } a[i] = e;$$
$$S_3 : rv := \text{true}$$
$$@\text{post } G : rv \leftrightarrow \exists i.l \leq i \leq u \wedge a[i] = e$$

Its verification condition is

$$\begin{aligned}
& F \rightarrow \mathbf{wp}(G, S_1; S_2; S_3) \\
\Leftrightarrow\ & F \rightarrow \mathbf{wp}(\mathbf{wp}(rv \leftrightarrow \exists i.l \leq i \leq u \wedge a[i] = e, rv := \text{true}), S_1; S_2) \\
\Leftrightarrow\ & F \rightarrow \mathbf{wp}(true \leftrightarrow \exists i.l \leq i \leq u \wedge a[i] = e, S_1; S_2) \\
\Leftrightarrow\ & F \rightarrow \mathbf{wp}(\exists i.l \leq i \leq u \wedge a[i] = e, S_1; S_2) \\
\Leftrightarrow\ & F \rightarrow \mathbf{wp}(\mathbf{wp}(\exists i.l \leq i \leq u \wedge a[i] = e, \texttt{assume } a[i] = e), S_1) \\
\Leftrightarrow\ & F \rightarrow \mathbf{wp}(a[i] = e \rightarrow \exists i.l \leq i \leq u \wedge a[i] = e, S_1) \\
\Leftrightarrow\ & F \rightarrow (i \leq u \rightarrow (a[i] = e \rightarrow \exists i.l \leq i \leq u \wedge a[i] = e))
\end{aligned}$$

Partial Correctness

- So far: how to verify the specification for each basic path.
- If the verification succeeds in every basic path, the program is partially correct.

Theorem

Given a program P, if for every basic path

$$
\begin{gathered}
@F \\
S_1; \\
\vdots; \\
S_n; \\
@G
\end{gathered}
$$

the verification condition $\{F\}S_1; \cdots ; S_n\{G\}$ is valid, then the program obeys its specification.

Summary

We have learned the **inductive assertion method** for proving **partial correctness**.

1. Break a function down into a set of basic paths.
2. Derive verification conditions (VCs) from every basic path.
3. Check the validity of VCs by an SMT solver.
4. If all of the VCs are valid, the function is partially correct.

The method can be automated, if proper loop invariants are given. Automatically generating loop invariants, however, is not an easy task and remains an active research area.