AAA528: Computational Logic

Lecture 7 — Partial Correctness Proof

Hakjoo Oh
2026 Spring

Proving Partial Correctness

- A function is *partially correct* if when the function's precondition is satisfied on entry, its postcondition is satisfied when the function returns (if it ever does).
- Inductive assertion method for proving partial correctness
  - Derive verification conditions (VCs) from a function.
  - Check the validity of VCs by an SMT solver.
  - If all of VCs are valid, the function ia partially correct.

Deriving VCs

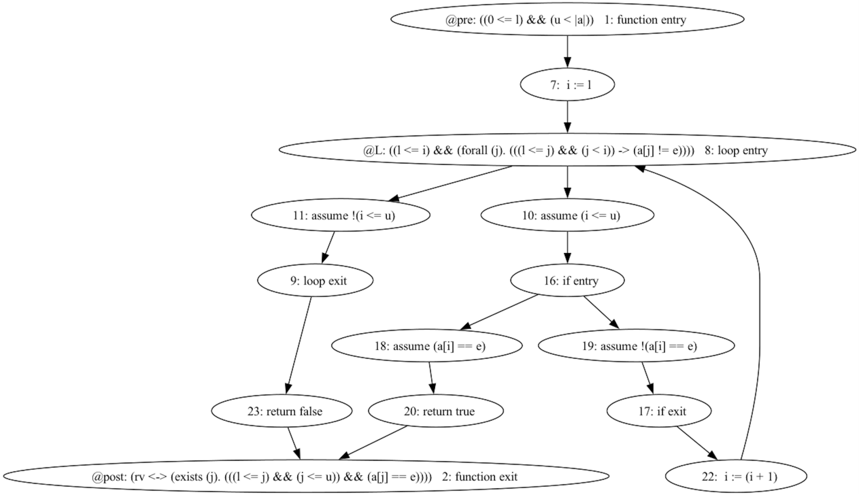
Done in two steps:

- The function is broken down into a finite set of *basic paths*.
- Each basic path generates a verification condition.
- Loops and recursive functions complicate proofs as they create unbounded number of paths. For loops, loop invariants cut the paths into a finite set of basic paths. For recursion, function specification cuts the paths.

Control-Flow Graph

```
@pre : $0 \le l \wedge u < |a|$
@post : $rv \leftrightarrow \exists i. l \le i \le u \wedge a[i] = e$
bool LinearSearch (int $a$[], int $l$, int $u$, int $e$) {
  int $i := l$;
  while @$L : l \le i \wedge (\forall j.\ l \le j < i \rightarrow a[j] \ne e)$
  ($i \le u$) {
    if ($a[i] = e$) return true
    $i := i + 1$;}
  return false }
```

Control-Flow Graph



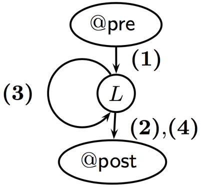
@pre: ((0 <= l) && (u < |a|))   1: function entry
7: i := l
@L: ((l <= i) && (forall (j). (((l <= j) && (j < i)) -> (a[j] != e))))   8: loop entry
11: assume !(i <= u)
10: assume (i <= u)
9: loop exit
16: if entry
18: assume (a[i] == e)
19: assume !(a[i] == e)
23: return false
20: return true
17: if exit
@post: (rv <-> (exists (j). (((l <= j) && (j <= u)) && (a[j] == e))))   2: function exit
22: i := (i + 1)

Basic Paths

- 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 (Basic paths do not cross loops).

Example: LinearSearch

```
@pre : $0 \le l \land u < |a|$
@post : $rv \leftrightarrow \exists i. l \le i \le u \land a[i] = e$
bool LinearSearch (int $a$[], int $l$, int $u$, int $e$) {
  int $i := l$;
  while
  @L : $l \le i \land (\forall j.\ l \le j < i \rightarrow a[j] \ne e)$
  ($i \le u$) {
    if ($a[i] = e$) return true
    $i := i + 1$;
  }
  return false
}
```



(1)

@pre : $0 \le l \land u < |a|$
$i := l;$
@L : $l \le i \land (\forall j.\ l \le j < i \rightarrow a[j] \ne e)$

(2)

@L : $l \le i \land (\forall j.\ l \le j < i \rightarrow a[j] \ne e)$
assume $i \le u$;
assume $a[i] = e$;
$rv :=$ true
@post : $rv \leftrightarrow \exists i. l \le i \le u \land a[i] = e$

(3)

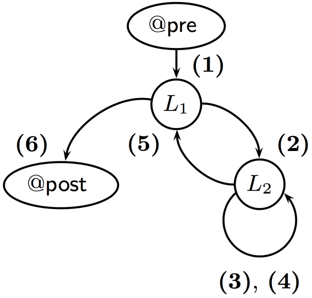
@L : $l \le i \land (\forall j.\ l \le j < i \rightarrow a[j] \ne e)$
assume $i \le u$;
assume $a[i] \ne e$
$i := i + 1;$
@L : $l \le i \land (\forall j.\ l \le j < i \rightarrow a[j] \ne e)$

(4)

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

Example: Bubble Sort

```
@pre : $|a_0| \geq 0$
@post : sorted$(rv, 0, |rv| - 1)$
int[] BubbleSort (int[] $a_0$) {
  int[] $a := a_0$
  @$L_1$ $\left[\begin{array}{l} -1 \leq i < |a| \\ \wedge \text{ partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge \text{ sorted}(a, i, |a|-1) \end{array}\right]$
  for (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
    @$L_2$ $\left[\begin{array}{l} 1 \leq i < |a| \ \wedge \ 0 \leq j \leq i \\ \wedge \text{ partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge \text{ partitioned}(a, 0, j-1, j, j) \\ \wedge \text{ sorted}(a, i, |a|-1) \end{array}\right]$
    for (int $j := 0$; $j < i$; $j := j + 1$) {
      if ($a[j] > a[j+1]$) {
        int $t := a[j]$;
        int $a[j] := a[j+1]$;
        int $a[j+1] := t$;
      }
    }
  }
  return $a$;
}
```


$$\begin{tikzcd}
 & @pre \arrow[d, "(1)"] & \\
 & L_1 \arrow[dl, bend right, "(6)"'] \arrow[dr, bend left, "(2)"] & \\
@post & & L_2 \arrow[ul, bend left, "(5)"] \arrow[loop below, "{(3), (4)}"]
\end{tikzcd}$$

Basic Paths

(1)

$@\text{pre} : |a| \geq 0$

$a := a_0;$

$i := |a| - 1;$

$@L_1 : -1 \leq i < |a| \wedge \text{partitioned}(a, 0, i, i+1, |a|-1) \wedge \text{sorted}(a, i, |a|-1)$

(2)

$@L_1 : -1 \leq i < |a| \wedge \text{partitioned}(a, 0, i, i+1, |a|-1) \wedge \text{sorted}(a, i, |a|-1)$

assume $i > 0;$

$j := 0;$

$$@L_2 : \left[\begin{array}{l} 1 \leq i < |a| \ \wedge \ 0 \leq j \leq i \wedge \ \text{partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge \ \text{partitioned}(a, 0, j-1, j, j) \wedge \ \text{sorted}(a, i, |a|-1) \end{array}\right]$$

(3)

$$@L_2 : \left[\begin{array}{l} 1 \leq i < |a| \ \wedge \ 0 \leq j \leq i \wedge \ \text{partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge \ \text{partitioned}(a, 0, j-1, j, j) \wedge \ \text{sorted}(a, i, |a|-1) \end{array}\right]$$

assume $j < i;$

assume $a[j] > a[j+1];$

$t := a[j];$

$a[j] := a[j+1];$

$a[j+1] := t;$

$j := j + 1;$

$$@L_2 : \left[\begin{array}{l} 1 \leq i < |a| \ \wedge \ 0 \leq j \leq i \wedge \ \text{partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge \ \text{partitioned}(a, 0, j-1, j, j) \wedge \ \text{sorted}(a, i, |a|-1) \end{array}\right]$$

Basic Paths

(4)

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

assume $j < i$;
assume $a[j] \le a[j+1]$;
$j := j + 1$;

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

(5)

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

assume $j \ge i$;
$i := i - 1$;

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

(6)

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

assume $i \le 0$;
$rv := a$;

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

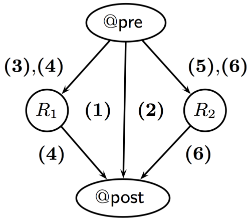
Basic Paths: Function Calls

- Both loops and (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.
- Observe that 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 replace 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.

Example: Binary Search

```
@pre : $0 \leq l \land u < |a| \land \textbf{sorted}(a, l, u)$
@post : $rv \leftrightarrow \exists i. l \leq i \leq u \land a[i] = e$
bool BinarySearch (int $a$[], int $l$, int $u$, int $e$) {
  if ($l > u$) return false;
  else {
    int $m$ := ($l + u$) div 2;
    if ($a[m] = e$) return true;
    else if ($a[m] < e$) {
      @$R_1$ : $0 \leq m + 1 \land u < |a| \land \textbf{sorted}(a, m + 1, u)$
      return BinarySearch ($a, m + 1, u, e$)
    } else {
      @$R_2$ : $0 \leq l \land m - 1 < |a| \land \textbf{sorted}(a, l, m - 1)$
      return BinarySearch ($a, l, m - 1, e$)
    }
  }
}
```

Basic Paths



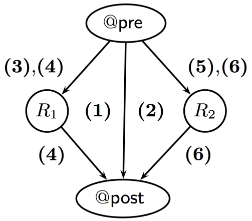
(1)
```
@pre : $0 \leq l \wedge u < |a| \wedge \text{sorted}(a, l, u)$
assume $l > u$;
$rv$ := false;
@post : $rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$
```

(2)
```
@pre : $0 \leq l \wedge u < |a| \wedge \text{sorted}(a, l, u)$
assume $l \leq u$;
$m := (l + u)$ div $2$;
assume $a[m] = e$;
$rv$ := true;
@post $rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$
```

(3)
```
@pre $0 \leq l \wedge u < |a| \wedge \text{sorted}(a, l, u)$
assume $l \leq u$;
$m := (l + u)$ div $2$;
assume $a[m] \neq e$;
assume $a[m] < e$;
@$R_1$ : $0 \leq m + 1 \wedge u < |a| \wedge \text{sorted}(a, m + 1, u)$
```

(5)
```
@pre $0 \leq l \wedge u < |a| \wedge \text{sorted}(a, l, u)$
assume $l \leq u$;
$m := (l + u)$ div $2$;
assume $a[m] \neq e$;
assume $a[m] \geq e$;
@$R_2$ : $0 \leq l \wedge m - 1 < |a| \wedge \text{sorted}(a, l, m - 1)$
```

Basic Paths



(4) @pre : $0 \leq l \wedge u < |a| \wedge \text{sorted}(a, l, u)$
assume $l \leq u$;
$m := (l + u)$ div 2;
assume $a[m] \neq e$;
assume $a[m] < e$;
assume $v_1 \leftrightarrow \exists i. m + 1 \leq i \leq u \wedge a[i] = e$
$rv := v_1$;
@post : $rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$

(6) @pre : $0 \leq l \wedge u < |a| \wedge \text{sorted}(a, l, u)$
assume $l \leq u$;
$m := (l + u)$ div 2;
assume $a[m] \neq e$;
assume $a[m] \geq e$;
assume $v_2 \leftrightarrow \exists i. l \leq i \leq m - 1 \wedge a[i] = e$;
$rv := v_2$;
@post : $rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$

Let $G[a, l, u, e, rv]$ be the post condition. Then, the function call

$$\text{return BinarySearch}(a, m + 1, u, e)$$

translates to

$$\text{assume } G[a, m + 1, u, e, v_1]; rv := v_1$$

Weakest Precondition

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

- { ? } `x:=x+1` { $x > 0$ }
- { ? } `y:=2*y` { $y < 6$ }
- { ? } `x:=x+y` { $y > x$ }
- { ? } `assume` $a \leq 5$ { $a \leq 5$ }
- { ? } `assume` $a \leq b$ { $a \leq 5$ }

The weakest precondition is the most general one among valid preconditions. Weakest precondition transformer:

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

Weakest Precondition Transformer

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

- Assumption: What must hold before statement **assume** c is executed to ensure that F holds afterwards? If $c \rightarrow F$ holds before, then satisfying c guarantees that F holds afterwards:

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

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

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

- For a sequence of statements $S_1; \ldots; S_n$,

$$\mathbf{wp}(F, S_1; \ldots, S_n) \Leftrightarrow \mathbf{wp}(\mathbf{wp}(F, S_n), S_1; \ldots, S_{n-1}).$$

Verification Condition

The verification condition of basic path

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

is

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

The verification condition is sometimes denoted by the Hoare triple

$$\{F\}S_1; \ldots; S_n\{G\}.$$

Example

The VC of the basic path

$$\begin{array}{l} @x \geq 0 \\ x := x + 1; \\ @x \geq 1 \end{array}$$

is

$$x \geq 0 \rightarrow \mathsf{wp}(x \geq 1, x := x + 1)$$

where

$$\mathsf{wp}(x \geq 1, x := x + 1) \iff x \geq 0$$

Example

Consider the basic path (2) in the LinearSearch example:

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

The VC is $F \rightarrow \mathrm{wp}(G, S_1; S_2; S_3)$, so compute

$$\begin{aligned}
&\mathrm{wp}(G,\ S_1; S_2; S_3) \\
\Leftrightarrow\ &\mathrm{wp}(\mathrm{wp}(rv \leftrightarrow \exists j.\ \ell \leq j \leq u \land a[j] = e,\ rv := \mathtt{true}),\ S_1; S_2) \\
\Leftrightarrow\ &\mathrm{wp}(\mathtt{true} \leftrightarrow \exists j.\ \ell \leq j \leq u \land a[j] = e,\ S_1; S_2) \\
\Leftrightarrow\ &\mathrm{wp}(\exists j.\ \ell \leq j \leq u \land a[j] = e,\ S_1; S_2) \\
\Leftrightarrow\ &\mathrm{wp}(\mathrm{wp}(\exists j.\ \ell \leq j \leq u \land a[j] = e,\ \mathtt{assume}\ a[i] = e),\ S_1) \\
\Leftrightarrow\ &\mathrm{wp}(a[i] = e \rightarrow \exists j.\ \ell \leq j \leq u \land a[j] = e,\ S_1) \\
\Leftrightarrow\ &\mathrm{wp}(a[i] = e \rightarrow \exists j.\ \ell \leq j \leq u \land a[j] = e,\ \mathtt{assume}\ i \leq u) \\
\Leftrightarrow\ &i \leq u \rightarrow (a[i] = e \rightarrow \exists j.\ \ell \leq j \leq u \land a[j] = e)
\end{aligned}$$

The resulting VC is $(T_{\mathbb{Z}} \cup T_A)$-valid:

$$\begin{aligned}
&l \leq i \land (\forall j.\, l \leq j < i \rightarrow a[j] \neq e) \\
&\qquad \rightarrow (i \leq u \rightarrow (a[i] = e \rightarrow \exists j. l \leq j \leq u \land a[j] = e))
\end{aligned}$$

Weakest Precondition Transformer for Array Assignment

$$\mathbf{wp}(F, x[e_1] := e_2) \iff F\{x \mapsto x\langle e_1 \lhd e_2\rangle\}$$

- $\{\ ?\ \}$ `x[i]:=y` $\{\ x[i] = z\ \}$
- $\{\ ?\ \}$ `x[i]:=y` $\{\ x[j] = z\ \}$
- $\{\ ?\ \}$ `y[j+1]:=x[i+1]` $\{\ y[j] = a \land y[j+1] = b\ \}$

Strongest Postcondition

Strongest postcondition transformer

$$\mathbf{sp} : \mathsf{FOL} \times \mathsf{stmts} \to \mathsf{FOL}$$

computes the most specific postcondition of a given precondition and program statement, e.g.,

- $\{ x > -1 \}$ `x:=x+1` $\{ ? \}$
- $\{ y < 3 \}$ `y:=2*y` $\{ ? \}$
- $\{ x < 0 \}$ `x:=x+y` $\{ ? \}$
- $\{ \top \}$ `assume` $a \leq 5$ $\{ ? \}$
- $\{ b \leq 5 \}$ `assume` $a \leq b$ $\{ ? \}$

Strongest Postcondition Transformer

- $\textsf{sp}(F, \textsf{assume } c) \iff c \wedge F$
- $\textsf{sp}(F[v], v := e[v]) \iff \exists v^0.\ v = e[v^0] \wedge F[v^0]$
- $\textsf{sp}(F, S_1; \ldots; S_n) \iff \textsf{sp}(\textsf{sp}(F, S_1), S_2; \ldots; S_n)$

$$\begin{aligned} & \textsf{sp}(i \geq n, i := i + k) \\ \iff & \exists i^0.\ i = i^0 + k \wedge i^0 \geq n \\ \iff & i - k \geq n \end{aligned}$$

$$\begin{aligned} & \textsf{sp}(i \geq n, \textsf{assume } k \geq 0;\ i := i + k) \\ \iff & \textsf{sp}(\textsf{sp}(i \geq n, \textsf{assume } k \geq 0), i := i + k) \\ \iff & \textsf{sp}(i \geq n \wedge k \geq 0, i := i + k) \\ \iff & \exists i^0.\ i = i^0 + k \wedge i^0 \geq n \wedge k \geq 0 \\ \iff & i - k \geq n \wedge k \geq 0 \end{aligned}$$

Verification Condition

The verification condition of basic path

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

is

$$\mathsf{sp}(F, S_1; \ldots; S_n) \to G$$

The verification condition is sometimes denoted by the Hoare triple

$$\{F\}S_1; \ldots; S_n\{G\}.$$

Partial Correctness

Theorem

If for every basic path

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

of program P, the verification condition

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

is valid, then the program obeys its specification.