AAA528: Computational Logic

Lecture 6 — Program Verification

Hakjoo Oh
2023 Spring

Program Verification

Techniques for specifying and verifying program properties:

- **Specification**: precise statement of program properties in first-order logic. Also called program annotations.
  - Partial correctness properties
  - Total correctness properties
- **Verification methods**: for proving partial/total correctness
  - Inductive assertion method
  - Ranking function method

Running Example 1: Linear Search

```
bool LinearSearch (int $a$[], int $l$, int $u$, int $e$) {
  int $i := l$;
  while ($i \leq u$) {
    if ($a[i] = e$) return true
    $i := i + 1$;
  }
  return false
}
```

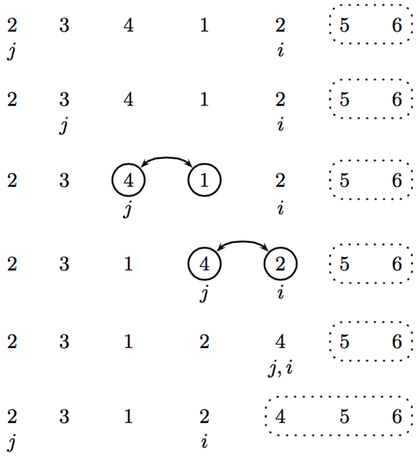
Running Example 2: Binary Search

```
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 ($a[m] < e$) return BinarySearch $(a, m + 1, u, e)$
    else return BinarySearch $(a, l, m - 1, e)$
  }
}
```

Running Example 3: Bubble Sort

```
bool BubbleSort (int $a$[]) {
  int[] $a := a_0$
  for (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
    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 $a$;
}
```

Running Example 3: Bubble Sort



Specification

- An annotation is a first-order logic formula F.
- An annotation F at location L expresses an *invariant* asserting that F is true whenever program control reaches L.
- Three types of annotations:
  - **Function specification**
  - **Loop invariant**
  - **Assertion**

Function Specifications

Formulas whose free variables include only the formal parameters and return variables.

- Precondition: Specification about what should be true upon entering the function.
- Postcondition: Specification about the expected output of the function. Postcondition relates the input and output of the function.

Example: Linear Search

The behavior of LinearSearch:

- It behaves correctly only when $l \geq 0$ and $u < |a|$.
- It returns true iff the array a contains the value e in the range $[l, u]$.

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

Our goal is to prove the *partial correctness* property: if the function precondition holds and the function halts, then the function postcondition holds upon return.

Example: Binary Search

- It behaves correctly only when $l \geq 0$, $u < |a|$, and a is sorted.
- It returns true iff the array a contains the value e in the range $[l, u]$.

```
@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 ($a[m] < e$) return BinarySearch $(a, m + 1, u, e)$
    else return BinarySearch $(a, l, m - 1, e)$
  }
}
```

$$\textbf{sorted}(a, l, u) \iff \forall i, j. l \leq i \leq j \leq u \rightarrow a[i] \leq a[j]$$

Example: Bubble Sort

- Any array can be given.
- The returned array is sorted.

```
@pre : $\top$
@post : sorted$(rv, 0, |rv| - 1)$
bool BubbleSort (int $a$[]) {
  int[] $a := a_0$
  for (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
    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 $a$;
}
```

Loop Invariants

For proving partial correctness, each loop must be annotated with a loop invariant $\boldsymbol{F}$:

```
while
    @F
    (⟨condition⟩) {
    ⟨body⟩
}
```

Loop invariant is a property that is preserved by executions of the loop body; $\boldsymbol{F}$ holds at the beginning of every iteration. Therefore,

- $\boldsymbol{F} \land \langle \boldsymbol{condition} \rangle$ holds on entering the body.
- $\boldsymbol{F} \land \neg\langle \boldsymbol{condition} \rangle$ holds when exiting the loop.

Loop Invariants

Find a loop invariant of the loop in LinearSearch:

```
@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
   @$L : l \leq i \wedge (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$
   $(i \leq u)$ {
      if $(a[i] = e)$ return true
      $i := i + 1$;
   }
   return false
}
```

Example: LinearSearch in Dafny

```
method LinearSearch (a: array<int>, l: int, u: int, e: int)
returns (b: bool)
  requires 0 <= l <= u && u < a.Length
  ensures (b <==> exists i :: l <= i <= u && a[i] == e);
{
  var i := l;
  b := false;
  while (i <= u)
    invariant l <= i <= a.Length
    invariant forall j :: l <= j < i ==> a[j] != e
  {
    if a[i] == e { b := true; break; }
    i := i + 1;
  }
}
```

Example: Bubble Sort

```
@pre : $\top$
@post : $\textbf{sorted}(rv, 0, |rv| - 1)$
bool BubbleSort (int $a$[]) {
  int[] $a := a_0$
  @$L_1$ $\left[\begin{array}{l} -1 \le i < |a| \\ \land\ \textbf{partitioned}(a, 0, i, i+1, |a|-1) \\ \land\ \textbf{sorted}(a, i, |a|-1) \end{array}\right]$
  for (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
    @$L_2$ $\left[\begin{array}{l} 1 \le i < |a|\ \land\ 0 \le j \le i \\ \land\ \textbf{partitioned}(a, 0, i, i+1, |a|-1) \\ \land\ \textbf{partitioned}(a, 0, j-1, j, j) \\ \land\ \textbf{sorted}(a, i, |a|-1) \end{array}\right]$
    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 $a$;
}
```

$$\textbf{partitioned}(a, l_1, u_1, l_2, u_2) \iff \forall i, j.\ l_1 \le i \le u_1 < l_2 \le j \le u_2 \rightarrow a[i] \le a[j].$$

Assertions

- Programmers' formal comments on the program behavior
- Runtime assertions: division by 0, array out of bounds, etc

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

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:

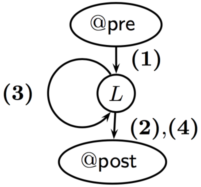
- The function is broken down into a finite set of *basic paths*.
- Each basic path generates a verification condition.
- Loops and recursive functions complicate proofs as they create unbounded number of paths. For loops, loop invariants cut the paths into a finite set of basic paths. For recursion,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

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 : $0 \le l \wedge u < |a|$
@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
}
```



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

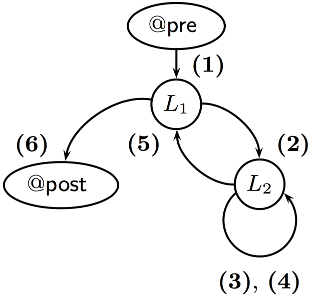
(2)
@L : $l \le i \wedge (\forall j.\ l \le j < i \rightarrow a[j] \ne e)$
assume $i \le u$;
assume $a[i] = e$;
$rv :=$ true
@post : $rv \leftrightarrow \exists i. l \le i \le u \wedge a[i] = e$

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

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

Example: Bubble Sort

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



Basic Paths

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

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

(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 $i \le 0$;
$rv := a$;
@$post$ $\text{sorted}(rv, 0, |rv|-1)$
```

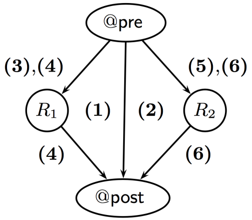
Basic Paths: Function Calls

- 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 : $0 \leq l \land u < |a| \land \mathsf{sorted}(a, l, u)$
@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 ($a[m] < e$) {
      @$R_1$ : $0 \leq m + 1 \land u < |a| \land \mathsf{sorted}(a, m + 1, u)$
      return BinarySearch $(a, m + 1, u, e)$
    } else {
      @$R_2$ : $0 \leq l \land m - 1 < |a| \land \mathsf{sorted}(a, l, m - 1)$
      return BinarySearch $(a, l, m - 1, e)$
    }
  }
}
```

Basic Paths



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

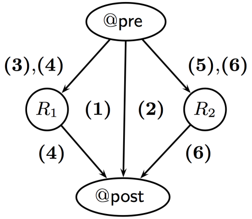
(2) @pre $0 \le l \wedge u < |a| \wedge \text{sorted}(a, l, u)$
assume $l \le u$;
$m := (l + u)$ div 2;
assume $a[m] = e$;
$rv := \text{true}$;
@post $rv \leftrightarrow \exists i. l \le i \le u \wedge a[i] = e$

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

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

Basic Paths

The basic paths that pass through function calls:



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

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

$G[a, l, u, e, rv]$ be the post condition. Then, the function call return BinarySearch$(a, m + 1, u, e)$ translates to 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?

- $\{ \ ?\ \boldsymbol{x > -1} \ \}$ `x:=x+1` $\{ \ \boldsymbol{x > 0} \ \}$
- $\{ \ ?\ \boldsymbol{y < 2.5} \ \}$ `y:=2*y` $\{ \ \boldsymbol{y < 5} \ \}$
- $\{ \ ?\ \boldsymbol{x < 0} \ \}$ `x:=x+y` $\{ \ \boldsymbol{y > x} \ \}$
- $\{ \ ?\ \top \ \}$ `assume` $\boldsymbol{a \leq 5}$ $\{ \ \boldsymbol{a \leq 5} \ \}$
- $\{ \ ?\ \boldsymbol{b \leq 5} \ \}$ `assume` $\boldsymbol{a \leq b}$ $\{ \ \boldsymbol{a \leq 5} \ \}$

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

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

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:

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

- 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:

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

- 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 Conditions

The verification condition of basic path

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

is

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

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

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

where

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

Example

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

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

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

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

The VC is $l \le i \land (\forall j.\ l \le j < i \to a[j] \ne e)$
$\to (i \le u \to (a[i] = e \to \exists j. l \le j \le u \land a[j] = e))$

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.

Total Correctness

- Total correctness = Partial correctness + Termination
- Total correctness of a function asserts that if the precondition holds on entry, then the function eventually halts and the postcondition holds.

Well-Founded Relations

- Termination proof is based on well-founded relations.
- A binary relation $\prec$ over a set S is well-founded iff there does not exist an infinite sequence $s_1, s_2, \ldots$ of elements of S such that

$$s_1 \succ s_2 \succ \cdots.$$

- For example, the relation $<$ is well-founded over the natural numbers, because any sequence of natural numbers decreasing according to $<$ is finite: e.g.,

$$1023 > 39 > 30 > 29 > 8 > 3 > 0.$$

However, the relation $<$ is not well-founded over the rationals or reals.

Lexicographic Relations

- A useful class of well-founded relations.
- From a set of pairs of sets and well-founded relations:

$$(S_1, \prec_1), \ldots, (S_m, \prec_m)$$

  construct the set

$$S = S_1 \times \cdots \times S_m$$

  and define the relation $\prec$:

$$(s_1, \ldots, s_m) \prec (t_1, \ldots, t_m) \iff \bigvee_{i=1}^{m} \left(s_i \prec_i t_i \wedge \bigwedge_{j=1}^{i-1} s_j = t_j\right)$$

- For example, let $S = \mathbb{N}^3$ and $<_3$ be triples of natural numbers and the natural lexicographic extension of $<$ to such triples, respectively:

$$(11, 9, 104) <_3 (11, 13, 3)$$

Proving Termination

- Define a set S with a well-founded relation $\prec$.
  - We usually choose as S the set of n-tuples of natural numbers and as $\prec_n$ the lexicographic extension $<_n$[1] of $<$, where n varies according to the application.
- Find a *ranking function* δ mapping program states to S such that δ decreases according to $\prec$ along every basic path.
- Then, since $\prec$ is well-founded, there cannot exist an infinite sequence of program states.

[1]When $n = 2$, $(a, b) <_2 (a', b') \iff a < a' \lor (a = a' \land b < b')$

Example: Bubble Sort

For each loop, annotate a ranking function:

```
@pre : $\top$
@post : $\top$
bool BubbleSort (int $a$[]) {
  int[] $a := a_0$
  @$L_1 : i + 1 \geq 0$
  $\downarrow (i + 1, i + 1)$
  for (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
    @$L_2 : i + 1 \geq 0 \land i - j \geq 0$
    $\downarrow (i + 1, i - j)$
    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 $a$;
}
```

Basic Paths

Prove that the ranking functions decrease along each basic paths.

```
(1)  $@L_1 : i + 1 \geq 0$
     $\downarrow L_1 : (i + 1, i + 1)$
     assume $i > 0$;
     $j := 0$;
     $\downarrow L_2 : (i + 1, i - j)$
(2)  $L_2 : i + 1 \geq 0 \wedge i - j \geq 0$
     $\downarrow L_2 : (i + 1, i - j)$
     assume $j < i$;
     assume $a[j] > a[j + 1]$;
     $t := a[j]$;
     $a[j] := a[j + 1]$;
     $a[j + 1] := t$;
     $j := j + 1$;
     $\downarrow L_2 : (i + 1, i - j)$
```

Basic Paths

$$
\begin{array}{ll}
(3) & L_2 : i+1 \geq 0 \land i-j \geq 0 \\
& \downarrow L_2 : (i+1, i-j) \\
& \textbf{assume } j < i; \\
& \textbf{assume } a[j] \leq a[j+1]; \\
& j := j+1; \\
& \downarrow L_2 : (i+1, i-j) \\
(4) & L_2 : i+1 \geq 0 \land i-j \geq 0 \\
& \downarrow L_2 : (i+1, i-j) \\
& \textbf{assume } j \geq i; \\
& i := i-1; \\
& \downarrow L_1 : (i+1, i+1)
\end{array}
$$

Other basic paths are not relevant to proving termination.

Verification Conditions

The verification condition of basic path

$$\begin{array}{l} @F \\ \downarrow \delta[\bar{x}] \\ S_1; \\ \vdots \\ S_n; \\ \downarrow \kappa[\bar{x}] \end{array}$$

is

$$F \rightarrow \mathsf{wp}(\kappa \prec \delta[\bar{x}_0], S_1; \ldots; S_n)\{\bar{x}_0 \mapsto \bar{x}\}$$

The value of κ after executing the statements is less than the value of δ before executing the statements. The annotation F can provide extra invariant to prove the relation.

Example

To derive the VC for the path

$$\begin{array}{ll}(4) & L_2 : i+1 \geq 0 \land i-j \geq 0 \\ & \downarrow L_2 : (i+1, i-j) \\ & \textbf{assume } j \geq i; \\ & i := i-1; \\ & \downarrow L_1 : (i+1, i+1)\end{array}$$

compute

$$\begin{array}{l}\textbf{wp}((i+1, i+1) \prec_2 (i_0+1, i_0-j_0), \textbf{assume } j \geq i; i := i-1) \\ \iff \textbf{wp}(((i_0-1)+1, (i_0-1)+1) <_2 (i_0+1, i_0-j_0), \textbf{assume } j \geq i) \\ \iff j \geq i \rightarrow (i, i) <_2 (i_0+1, i_0-j_0)\end{array}$$

Then, replace the variables:

$$j \geq i \rightarrow (i, i) <_2 (i+1, i-j).$$

The VC:

$$i+1 \geq 0 \land i-j \geq 0 \land j \geq i \rightarrow (i, i) <_2 (i+1, i-j).$$

Exercise

Compute the verification conditions for the basic paths (1)–(3).

Example: Binary Search

```
@pre : $u - l + 1 \geq 0$
@post : $\top$
$\downarrow u - l + 1$
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 ($a[m] < e$) return BinarySearch $(a, m + 1, u, e)$
    else return BinarySearch $(a, l, m - 1, e)$
  }
}
```

Basic Paths

```
(1)  pre : $u - l + 1 \geq 0$
     $\downarrow u - l + 1$
     assume $l \leq u$;
     $m := (l + u)$ div $2$;
     assume $a[m] \neq e$
     assume $a[m] < e$
     $\downarrow u - (m + 1) + 1$
```

VC:

$$u - l + 1 \geq 0 \land l \leq u \land \cdots \rightarrow u - (((l + u) \text{ div } 2) + 1) + 1 < u - l + 1$$

Basic Paths

```
(2) pre : $u - l + 1 \geq 0$
    assume $l \leq u$;
    $m := (l + u)$ div $2$;
    assume $a[m] \neq e$
    assume $a[m] \geq e$
    $\downarrow (m - 1) - l + 1$
```

VC:

$$u - l + 1 \geq 0 \wedge l \leq u \wedge \cdots \rightarrow (((l + u) \text{ div } 2) - 1) - l + 1 < u - l + 1$$

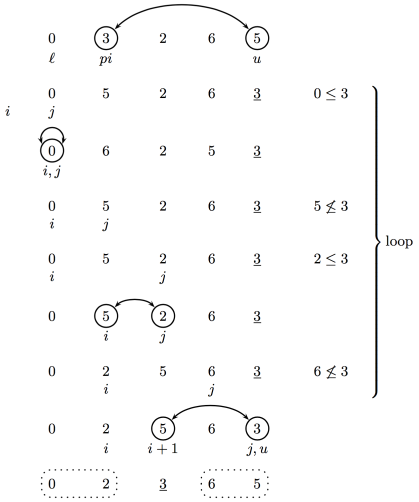
Example: QuickSort

Prove that QuickSort returns a sorted array and always halts.

```
typedef struct qs {
  int $pivot$;
  int[] $array$;
} qs;

@pre $\top$
@post sorted($rv, 0, |rv| - 1$)
int[] QuickSort(int[] $a$) {
  return qsort($a, 0, |a| - 1$);
}

@pre $\top$
@post $\top$
int[] qsort(int[] $a_0$, int $\ell$, int $u$) {
  int[] $a := a_0$;
  if ($\ell \geq u$) return $a$;
  else {
    qs $p$ := partition($a, \ell, u$);
    $a := p.array$;
    $a$ := qsort($a, \ell, p.pivot - 1$);
    $a$ := qsort($a, p.pivot + 1, u$);
    return $a$;
  }
}
```

```
@pre $\top$
@post $\top$
qs partition(int[] $a_0$, int $\ell$, int $u$) {
  int[] $a := a_0$;
  int $pi$ := random($\ell, u$);
  int $pv := a[pi]$;
  $a[pi] := a[u]$;
  $a[u] := pv$;

  int $i := \ell - 1$;
  for @ $\top$
    (int $j := \ell$; $j < u$; $j := j + 1$) {
    if ($a[j] \leq pv$) {
      $i := i + 1$;
      $t := a[i]$;
      $a[i] := a[j]$;
      $a[j] := t$;
    }
  }

  $t := a[i + 1]$;
  $a[i + 1] := a[u]$;
  $a[u] := t$;
  return
    { $pivot = i + 1$;
      $a = a$;
    };
}
```

QuickSort



0 3 2 6 5
ℓ pi u
0 5 2 6 3 0 ≤ 3
i j
0 6 2 5 3
i, j
0 5 2 6 3 5 ≰ 3
i j
0 5 2 6 3 2 ≤ 3
i j
0 5 2 6 3
i j
0 2 5 6 3 6 ≰ 3
i j
0 2 5 6 3
i i + 1 j, u
0 2 3 6 5
loop

Function Specification

$$
\begin{array}{l}
@\mathsf{pre}\ \left[\begin{array}{l}
0 \le \ell \ \land \ u < |a_0| \\
\ \land\ \mathsf{partitioned}(a_0, 0, \ell-1, \ell, u) \\
\ \land\ \mathsf{partitioned}(a_0, \ell, u, u+1, |a_0|-1)
\end{array}\right] \\
@\mathsf{post}\ \left[\begin{array}{l}
|rv| = |a_0| \ \land\ \mathsf{beq}(rv, a_0, 0, \ell-1) \ \land\ \mathsf{beq}(rv, a_0, u+1, |a_0|-1) \\
\ \land\ \mathsf{partitioned}(rv, 0, \ell-1, \ell, u) \\
\ \land\ \mathsf{partitioned}(rv, \ell, u, u+1, |rv|-1) \\
\ \land\ \mathsf{sorted}(rv, \ell, u)
\end{array}\right] \\
\texttt{int}[]\ \mathsf{qsort}(\texttt{int}[]\ a_0,\ \texttt{int}\ \ell,\ \texttt{int}\ u)
\end{array}
$$

$$
\begin{array}{l}
@\mathsf{pre}\ \left[\begin{array}{l}
0 \le \ell \ \land \ u < |a_0| \\
\ \land\ \mathsf{partitioned}(a_0, 0, \ell-1, \ell, u) \\
\ \land\ \mathsf{partitioned}(a_0, \ell, u, u+1, |a_0|-1)
\end{array}\right] \\
@\mathsf{post}\ \left[\begin{array}{l}
|rv.array| = |a_0| \ \land\ \mathsf{beq}(rv.array, a_0, 0, \ell-1) \\
\ \land\ \mathsf{beq}(rv.array, a_0, u+1, |a_0|-1) \\
\ \land\ \mathsf{partitioned}(rv.array, 0, \ell-1, \ell, u) \\
\ \land\ \mathsf{partitioned}(rv.array, \ell, u, u+1, |rv.array|-1) \\
\ \land\ \ell \le rv.pivot \le u \\
\ \land\ \mathsf{partitioned}(rv.array, \ell, rv.pivot-1, rv.pivot, rv.pivot) \\
\ \land\ \mathsf{partitioned}(rv.array, rv.pivot, rv.pivot, rv.pivot+1, u)
\end{array}\right] \\
\mathsf{qs}\ \mathsf{partition}(\texttt{int}[]\ a_0,\ \texttt{int}\ \ell,\ \texttt{int}\ u)
\end{array}
$$

Termination Argument

```
@pre $u - \ell + 1 \geq 0$
@post $\top$
$\downarrow \delta_2 :\ u - \ell + 1$
int[] qsort(int[] $a_0$, int $\ell$, int $u$) {
  int[] $a := a_0$;
  if $(\ell \geq u)$ return $a$;
  else {
    qs $p :=$ partition$(a, \ell, u)$;
    $a := p.array$;
    $a :=$ qsort$(a, \ell, p.pivot - 1)$;
    $a :=$ qsort$(a, p.pivot + 1, u)$;
    return $a$;
  }
}

@pre $\ell \leq u$
@post $\ell \leq rv.pivot \ \land\ rv.pivot \leq u$
qs partition(int[] $a_0$, int $\ell$, int $u$) {
  ⋮
  int $i := \ell - 1$;
  for
    @$L_1 :\ \ell \leq j \ \land\ j \leq u \ \land\ \ell - 1 \leq i \ \land\ i < j$
    $\downarrow \delta_1 :\ u - j$
    (int $j := \ell$; $j < u$; $j := j + 1$) {
    ⋮
  }
  ⋮
  return
    { $pivot \ = \ i + 1$;
      $a \ = \ a$;
    };
}
```

Exercise 1: Absolute Value

Prove the partial correctness of the function:

```
@pre $\top$
@post $\forall i.\ 0 \le i < |rv| \ \rightarrow\ rv[i] \ge 0$
int[] abs(int[] $a_0$) {
  int[] $a := a_0$;
  for @ $\top$
    (int $i := 0$; $i < |a|$; $i := i + 1$) {
    if ($a[i] < 0$) {
      $a[i] := -a[i]$;
    }
  }
  return $a$;
}
```

That is, annotate the function; list basic paths and verification conditions; and argue that the VC's are valid.

Exercise 2: Insertion Sort

Prove the partial correctness of the function:

```
@pre $\top$
@post sorted$(rv, 0, |rv| - 1)$
int[] InsertionSort(int[] $a_0$) {
  int[] $a := a_0$;
  for @ $\top$
    (int $i := 1$; $i < |a|$; $i := i + 1$) {
    int $t := a[i]$;
    for @ $\top$
      (int $j := i - 1$; $j \geq 0$; $j := j - 1$) {
      if $(a[j] \leq t)$ break;
      $a[j + 1] := a[j]$;
    }
    $a[j + 1] := t$;
  }
  return $a$;
}
```

That is, annotate the function; list basic paths and verification conditions; and argue that the VC's are valid.

Summary

Inductive assertion method for proving partial correctness:

1. Derive verification conditions (VCs) from a function.
2. Check the validity of VCs by an SMT solver.
3. If all of VCs are valid, the function ia 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.