The Calculus of Computation
Zohar Manna
Spring 2009

Chapter 5: Program Correctness: Mechanics

Program A: <u>LinearSearch</u> with function specification

```
@pre 0 ≤ ℓ ∧ u < |a|
@post rv ↔ ∃i. ℓ ≤ i ≤ u ∧ a[i] = e
bool LinearSearch(int[] a, int ℓ, int u, int e) {
  for @ ⊤
    (int i := ℓ; i ≤ u; i := i + 1) {
    if (a[i] = e) return true;
  }
  return false;
}
```

Function <u>LinearSearch</u> searches subarray of array a of integers for specified value e.

<u>Function specifications</u>

- Function precondition (@*pre*)
  It behaves correctly only if $0 \leq \ell$ and $u < |a|$
- Function postcondition (@*post*)
  It returns <u>true</u> iff a contains the value e in the range $[\ell, u]$

<u>for loop</u>: initially set i to be ℓ,

    execute the body and increment i by 1

    as long as $i \leq u$

@ - program annotation

Program B: BinarySearch with function specification

```
@pre 0 ≤ ℓ ∧ u < |a| ∧ sorted(a, ℓ, u)
@post rv ↔ ∃i. ℓ ≤ i ≤ u ∧ 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);
  }
}
```

The recursive function BinarySearch searches sorted subarray a of integers for specified value e.

sorted: weakly increasing order, i.e.

$$\text{sorted}(a, \ell, u) \Leftrightarrow \forall i, j.\ \ell \le i \le j \le u \rightarrow a[i] \le a[j]$$

Defined in the combined theory of integers and arrays, $T_{\mathbb{Z} \cup A}$

Function specifications

► Function precondition (@pre)
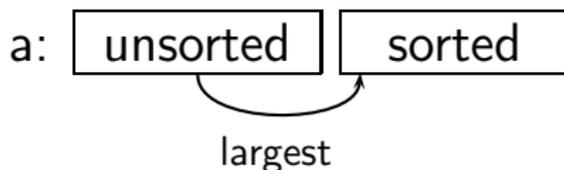 It behaves correctly only if
    $0 \le \ell$ and $u < |a|$ and
    sorted(a, ℓ, u).

► Function postcondition (@post)
  It returns <u>true</u> iff a contains the value e in the range $[\ell, u]$

Program C: <u>BubbleSort</u> with function specification

```
@pre ⊤
@post sorted(rv, 0, |rv| − 1)
int[] BubbleSort(int[] a₀) {
  int[] a := a₀;
  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];
        a[j] := a[j + 1];
        a[j + 1] := t;
      }
    }
  }
  return a;
}
```

Function <u>BubbleSort</u> sorts integer array *a*

by "bubbling" the largest element of the left unsorted region of *a* toward the sorted region on the right.

Each iteration of the outer loop expands the sorted region by one cell.[1]

<u>Function specification</u>

- Function postcondition (@post):

  <u>BubbleSort</u> returns array *rv* sorted on the range $[0, |rv| - 1]$.

---

[1]Except the last iteration, which expands the sorted region by two cells, so that an entire array of length *n* is sorted in $n - 1$ iterations.

Sample execution of BubbleSort

```
2    3    4    1    2    5    6
j                   i

2    3    4    1    2   [5    6]
     j              i

2    3   (4)  (1)   2   [5    6]
          j          i

2    3    1   (4)  (2)  [5    6]
               j    i

2    3    1    2    4   [5    6]
                   j, i

2    3    1    2   [4    5    6]
j                  i
```

Program Annotation

- ▶ Function Specifications
    function precondition (@pre)
    function postcondition (@post)

- ▶ Runtime Assertions
    e.g., $@\ 0 \le j < |a| \ \wedge \ 0 \le j + 1 < |a|$
    $a[j] := a[j + 1]$

- ▶ Loop Invariants
    e.g., $@\ L : \ell \le i \ \wedge \ \forall j.\ \ell \le j < i \ \rightarrow \ a[j] \ne e$

    The $L :$ gives a name to the formula, just like the $F :$ we've used in other formulae.

Program A: <u>LinearSearch</u> with runtime assertions

```
@pre 0 ≤ ℓ ∧ u < |a|
@post rv ↔ ∃i. ℓ ≤ i ≤ u ∧ a[i] = e
bool LinearSearch(int[] a, int ℓ, int u, int e) {
  for
    @ L : ⊤
    (int i := ℓ; i ≤ u; i := i + 1) {
    @ 0 ≤ i < |a|;
    if (a[i] = e) return true;
  }
  return false;
}
```

Program B: BinarySearch with runtime assertions

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

Program C: BubbleSort with runtime assertions

```
@pre ⊤
@post sorted(rv, 0, |rv| − 1)
int[] BubbleSort(int[] a₀) {
  int[] a := a₀;
  for
    @ L₁ : ⊤
    (int i := |a| − 1; i > 0; i := i − 1) {
    for
      @ L₂ : ⊤
      (int j := 0; j < i; j := j + 1) {
      @ 0 ≤ j < |a| ∧ 0 ≤ j + 1 < |a|;
      if (a[j] > a[j + 1]) {
        int t := a[j];
        a[j] := a[j + 1];
        a[j + 1] := t;
      }
    }
  }
  return a;
}
```

Loop Invariants

```
while
  @ F
  ⟨cond⟩ { ⟨body⟩ }
```

- ▶ apply ⟨body⟩ as long as ⟨cond⟩ holds
- ▶ assertion F holds at the beginning of every iteration
  evaluated <u>before</u> ⟨cond⟩ is checked

`for`
  `@ F`
  `(`⟨init⟩`;` ⟨cond⟩`;` ⟨incr⟩`)`
  `{`⟨body⟩`}`
	⟨init⟩`;`
	`while`
	  `@ F`
	  ⟨cond⟩ `{` ⟨body⟩`;` ⟨incr⟩ `}`

Program A: <u>LinearSearch</u> with loop invariants

```
@pre 0 ≤ ℓ ∧ u < |a|
@post rv ↔ ∃j. ℓ ≤ j ≤ u ∧ a[j] = e
bool LinearSearch(int[] a, int ℓ, int u, int e) {
  for
    @L : ℓ ≤ i ∧ (∀j. ℓ ≤ j < i → a[j] ≠ e)
    (int i := ℓ; i ≤ u; i := i + 1) {
    if (a[i] = e) return true;
  }
  return false;
}
```

Proving Partial Correctness

A function is <u>partially correct</u> if
when the program's precondition is satisfied on entry,
its postcondition is satisfied <u>when</u> the program halts/exits.

- ▶ A program + annotation is reduced to finite set of
  <u>verification conditions</u> (VCs), FOL formulae
- ▶ If all VCs are T-valid, then the program obeys its specification
  (partially correct)

Basic Paths: Loops

To handle loops, we break the program into basic paths

@ ← precondition or loop invariant

sequence of instructions
(with no loop invariants)

@ ← loop invariant, runtime assertion, or postcondition

Program A: LinearSearch I

Basic Paths of LinearSearch

(1)

@pre $0 \leq \ell \wedge u < |a|$

$i := \ell$;

@L : $\ell \leq i \wedge \forall j. \ \ell \leq j < i \ \rightarrow \ a[j] \neq e$

---

(2)

@L : $\ell \leq i \wedge \forall j. \ \ell \leq j < i \ \rightarrow \ a[j] \neq e$

assume $i \leq u$;

assume $a[i] = e$;

$rv := \text{true}$;

@post $rv \ \leftrightarrow \ \exists j. \ \ell \leq j \leq u \wedge a[j] = e$

Program A: LinearSearch II

---
(3)

$@L: \ \ell \leq i \land \forall j. \ \ell \leq j < i \ \rightarrow \ a[j] \neq e$

`assume` $i \leq u$;

`assume` $a[i] \neq e$;

$i := i + 1$;

$@L: \ \ell \leq i \land \forall j. \ \ell \leq j < i \ \rightarrow \ a[j] \neq e$

---
(4)

$@L: \ \ell \leq i \land \forall j. \ \ell \leq j < i \ \rightarrow \ a[j] \neq e$

`assume` $i > u$;

$rv :=$ `false`;

$@$post $rv \ \leftrightarrow \ \exists j. \ \ell \leq j \leq u \land a[j] = e$

Visualization of basic paths of LinearSearch
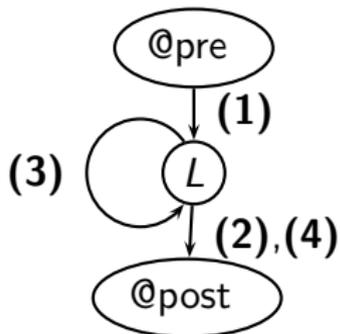
Program C: BubbleSort with loop invariants

```
@pre |a_0| > 0
@post sorted(rv, 0, |rv| - 1)
int[] BubbleSort(int[] a_0) {
    int[] a := a_0;
    for
        @L_1 : ⎡0 ≤ i < |a|                              ⎤
               ⎢∧partitioned(a, 0, i, i + 1, |a| - 1)⎥
               ⎣∧sorted(a, i, |a| - 1)                  ⎦
        (int i := |a| - 1;  i > 0;  i := i - 1) {
```

```
for
        ⎡1 ≤ i < |a| ∧ 0 ≤ j ≤ i                  ⎤
  @L₂ : ⎢∧partitioned(a, 0, i, i + 1, |a| − 1)     ⎥
        ⎢∧partitioned(a, 0, j − 1, j, j)           ⎥
        ⎣∧sorted(a, i, |a| − 1)                    ⎦
  (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;
}
```

Partition

$$\text{partitioned}(a, \ell_1, u_1, \ell_2, u_2)$$
$$\Leftrightarrow \forall i, j. \ \ell_1 \leq i \leq u_1 < \ell_2 \leq j \leq u_2 \ \rightarrow \ a[i] \leq a[j]$$

in $T_{\mathbb{Z}} \cup T_A$.

That is, each element of a in the range $[\ell_1, u_1]$ is $\leq$ each element in the range $[\ell_2, u_2]$.

Basic Paths of BubbleSort

---
(1)

$@\text{pre} \ |a_0| > 0$

$a := a_0;$

$i := |a| - 1;$

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

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

```
assume i > 0;
j := 0;
```

$$@L_2 : \begin{bmatrix} 1 \le i < |a| \land 0 \le j \le i \land \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \land \mathsf{partitioned}(a, 0, j-1, j, j) \land \mathsf{sorted}(a, i, |a|-1) \end{bmatrix}$$

$$@L_2 : \begin{bmatrix} 1 \le i < |a| \land 0 \le j \le i \land \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \land \ \mathsf{partitioned}(a, 0, j-1, j, j) \land \mathsf{sorted}(a, i, |a|-1) \end{bmatrix}$$

```
assume j < i;
assume a[j] > a[j + 1];
t := a[j];
a[j] := a[j + 1];
a[j + 1] := t;
j := j + 1;
```

$$@L_2 : \begin{bmatrix} 1 \le i < |a| \land 0 \le j \le i \land \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \land \ \mathsf{partitioned}(a, 0, j-1, j, j) \land \mathsf{sorted}(a, i, |a|-1) \end{bmatrix}$$

$$\frac{\qquad \textbf{(4)} \qquad}{}$$

$@L_2 : \begin{bmatrix} 1 \le i < |a| \land 0 \le j \le i \land \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \land\ \mathsf{partitioned}(a, 0, j-1, j, j) \land \mathsf{sorted}(a, i, |a|-1) \end{bmatrix}$

`assume` $j < i$;

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

$j := j + 1$;

$@L_2 : \begin{bmatrix} 1 \le i < |a| \land 0 \le j \le i \land \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \land\ \mathsf{partitioned}(a, 0, j-1, j, j) \land \mathsf{sorted}(a, i, |a|-1) \end{bmatrix}$

$$@L_2 : \begin{bmatrix} 1 \leq i < |a| \land 0 \leq j \leq i \land \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \land \; \mathsf{partitioned}(a, 0, j-1, j, j) \land \mathsf{sorted}(a, i, |a|-1) \end{bmatrix}$$

$\texttt{assume } j \geq i;$

$i := i - 1;$

$$@L_1 : \begin{bmatrix} 0 \leq i < |a| \land \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \land \; \mathsf{sorted}(a, i, |a|-1) \end{bmatrix}$$

$$
\begin{array}{l}
\textbf{(6)} \\
@L_1 : \begin{bmatrix} 0 \leq i < |a| \wedge \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge\ \mathsf{sorted}(a, i, |a|-1) \end{bmatrix} \\[1em]
\texttt{assume}\ i \leq 0; \\
rv := a; \\
@\mathsf{post}\ \mathsf{sorted}(rv, 0, |rv|-1)
\end{array}
$$

Visualization of basic paths of BubbleSort
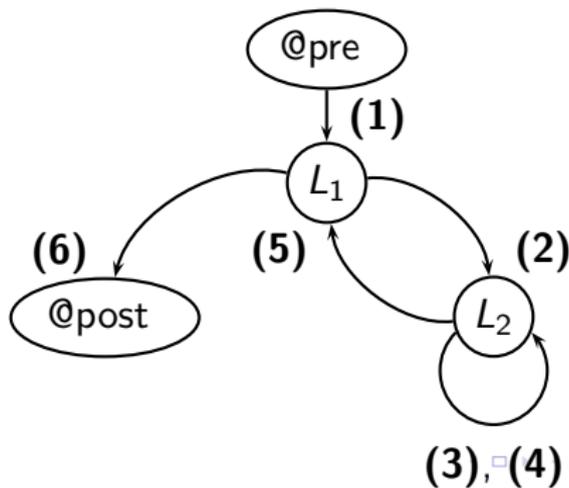
Basic Paths: Function Calls

- Loops produce unbounded number of paths
  loop invariants cut loops to produce
  finite number of basic paths
- Reursive calls produce unbounded number of paths
  function specifications cut function calls

In BinarySearch

@pre $0 \leq \ell \wedge u < |a| \wedge$ sorted(a, ℓ, u) ... $F[a, \ell, u, e]$

      ⋮

     @R_1 : $0 \leq m + 1 \wedge u < |a| \wedge$ sorted$(a, m+1, u)$ ... $F[a, m+1, u, e]$
     return BinarySearch$(a, m+1, u, e)$

      ⋮

     @R_2 : $0 \leq \ell \wedge m - 1 < |a| \wedge$ sorted$(a, \ell, m-1)$ ... $F[a, \ell, m-1, e]$
     return BinarySearch$(a, \ell, m-1, e)$

Program B: BinarySearch with function call assertions

```
@pre 0 ≤ ℓ ∧ u < |a| ∧ sorted(a, ℓ, u)
@post rv ↔ ∃i. ℓ ≤ i ≤ u ∧ 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) {
      @R₁ :  0 ≤ m + 1 ∧ u < |a| ∧ sorted(a, m + 1, u);
      return BinarySearch(a, m + 1, u, e);
    } else {
      @R₂ :  0 ≤ ℓ ∧ m − 1 < |a| ∧ sorted(a, ℓ, m − 1);
      return BinarySearch(a, ℓ, m − 1, e);
    }
  }
}
```

---
(1)

@pre $0 \leq \ell \wedge u < |a| \wedge \text{sorted}(a, \ell, u)$

`assume` $\ell > u$;

$rv := \texttt{false}$;

@post $rv \leftrightarrow \exists i.\ \ell \leq i \leq u \wedge a[i] = e$

---
(2)

@pre $0 \leq \ell \wedge u < |a| \wedge \text{sorted}(a, \ell, u)$

`assume` $\ell \leq u$;

$m := (\ell + u) \texttt{ div } 2$;

`assume` $a[m] = e$;

$rv := \texttt{true}$;

@post $rv \leftrightarrow \exists i.\ \ell \leq i \leq u \wedge a[i] = e$

@pre $0 \leq \ell \wedge u < |a| \wedge \text{sorted}(a, \ell, u)$

assume $\ell \leq u$;

$m := (\ell + u)$ div 2;

assume $a[m] \neq e$;

assume $a[m] < e$;

@$R_1 : 0 \leq m + 1 \wedge u < |a| \wedge \text{sorted}(a, m + 1, u)$

@pre $0 \leq \ell \wedge u < |a| \wedge \text{sorted}(a, \ell, u)$

assume $\ell \leq u$;

$m := (\ell + u) \text{ div } 2$;

assume $a[m] \neq e$;

assume $a[m] \geq e$;

@$R_2 : 0 \leq \ell \wedge m - 1 < |a| \wedge \text{sorted}(a, \ell, m - 1)$

@pre $0 \leq \ell \wedge u < |a| \wedge \mathsf{sorted}(a, \ell, u)$

assume $\ell \leq u$;

$m := (\ell + u)$ 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 $rv \leftrightarrow \exists i.\ \ell \leq i \leq u \wedge a[i] = e$

@pre $0 \leq \ell \wedge u < |a| \wedge \text{sorted}(a, \ell, u)$

assume $\ell \leq u$;

$m := (\ell + u) \text{ div } 2$;

assume $a[m] \neq e$;

assume $a[m] \geq e$;

assume $v_2 \leftrightarrow \exists i. \ell \leq i \leq m - 1 \wedge a[i] = e$;

$rv := v_2$;

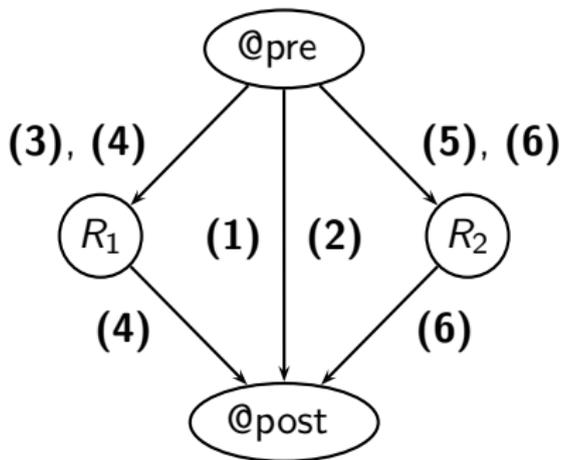@post $rv \leftrightarrow \exists i. \ell \leq i \leq u \wedge a[i] = e$

Figure: Visualization of basic paths of BinarySearch

Program States

Program counter pc holds current location of control

State s of P assignment of values to all variables
(proper types) of P

Example:

$$s : \left\{ \begin{array}{l} pc \mapsto L_2, \ a \mapsto [0; 1; 2], \\ i \mapsto 3, \ j \mapsto 0 \end{array} \right\}$$

is a state of BubbleSort.

Reachable state s of P a state that can be reached during
some computation of P

Example:

$$s : \left\{ \begin{array}{l} pc \mapsto L_2, \ a \mapsto [0; 1; 2], \\ i \mapsto 2, \ j \mapsto 0 \end{array} \right\}$$
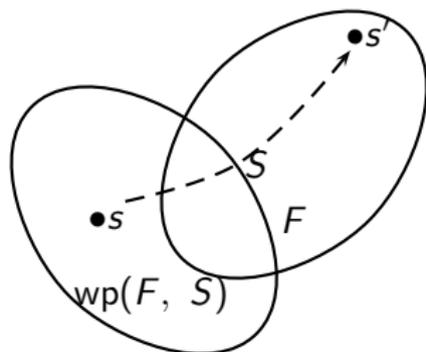
is a reachable state of BubbleSort.

Weakest Precondition wp(F, S)

For FOL formula F, program statement S,
$s \models \text{wp}(F, S)$ iff
   statement S is executed on state s to produce state s',
   and $s' \models F$:

- wp(F, assume c) $\Leftrightarrow$ $c \rightarrow F$
- wp($F[v]$, $v := e$) $\Leftrightarrow$ $F[e]$
- For $S_1; \dots; S_n$,
  wp(F, $S_1; \dots; S_n$) $\Leftrightarrow$ wp(wp(F, S_n), $S_1; \dots; S_{n-1}$)

Verification Conditions

Verification Condition of basic path

     @ F

     S_1;

     . . .

     S_n;

     @ G

  is

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

  Also denoted by

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

That is, for every state s,

    if $s \models F$

    then $s' \models G$ (after the path $S_1; S_2; \ldots; S_n$ is executed)

Example: Basic path

$$
\begin{array}{l}
\hline
\text{@ } F : \ x \geq 0 \\
S_1 : \ x := x + 1; \\
\text{@ } G : \ x \geq 1 \\
\hline
\end{array}
$$

(1)

The VC is $\quad F \ \rightarrow \ \text{wp}(G, \ S_1)$. That is,

$$
\begin{aligned}
& \text{wp}(G, \ S_1) \\
\Leftrightarrow \quad & \text{wp}(x \geq 1, \ x := x + 1) \\
\Leftrightarrow \quad & (x \geq 1)\{x \ \mapsto \ x + 1\} \\
\Leftrightarrow \quad & x + 1 \geq 1 \\
\Leftrightarrow \quad & x \geq 0
\end{aligned}
$$

Therefore the VC of path (1) is

$$
x \geq 0 \ \rightarrow \ x \geq 0 \ ,
$$

which is $T_{\mathbb{Z}}$-valid.

Example 1: Shortcut (backward substitution)

$$
\text{VC:} \qquad \boxed{\underbrace{x \geq 0}_{F} \rightarrow \underbrace{x \geq 0}_{\text{wp}(G,S_1)}}
$$

$@F: \quad x \geq 0$

$\qquad\qquad\quad x + 1 \geq 1 \quad i.e. \quad x \geq 0$

$S_1: \quad x := x + 1;$

$\qquad\qquad\qquad x \geq 1$

$@G: \quad x \geq 1$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Uparrow$

Example: Basic path (2) of LinearSearch

(2)

$@L: \ F: \ \ell \leq i \wedge \forall j. \ \ell \leq j < i \ \rightarrow \ a[j] \neq e$

$S_1: \ \texttt{assume } i \leq u;$

$S_2: \ \texttt{assume } a[i] = e;$

$S_3: \ rv := \texttt{true};$

$@\text{post } G: \ rv \ \leftrightarrow \ \exists j. \ \ell \leq j \leq u \wedge a[j] = e$

The VC is $\quad F \ \rightarrow \ \text{wp}(G, \ S_1; S_2; S_3)$. That is,

$$\text{wp}(G, \ S_1; S_2; S_3)$$

$\Leftrightarrow \quad \text{wp}(\text{wp}(rv \ \leftrightarrow \ \exists j. \ \ell \leq j \leq u \wedge a[j] = e, \ rv := \texttt{true}), \ S_1; S_2)$

$\Leftrightarrow \quad \text{wp}(\texttt{true} \ \leftrightarrow \ \exists j. \ \ell \leq j \leq u \wedge a[j] = e, \ S_1; S_2)$

$\Leftrightarrow \quad \text{wp}(\exists j. \ \ell \leq j \leq u \wedge a[j] = e, \ S_1; S_2)$

$\Leftrightarrow \quad \text{wp}(\text{wp}(\exists j. \ \ell \leq j \leq u \wedge a[j] = e, \ \texttt{assume } a[i] = e), \ S_1)$

$\Leftrightarrow \quad \text{wp}(a[i] = e \ \rightarrow \ \exists j. \ \ell \leq j \leq u \wedge a[j] = e, \ S_1)$

$\Leftrightarrow \quad \text{wp}(a[i] = e \ \rightarrow \ \exists j. \ \ell \leq j \leq u \wedge a[j] = e, \ \texttt{assume } i \leq u)$

$\Leftrightarrow \quad i \leq u \ \rightarrow \ (a[i] = e \ \rightarrow \ \exists j. \ \ell \leq j \leq u \wedge a[j] = e)$

Therefore the VC of path (2) is

$$\ell \le i \ \wedge \ (\forall j. \ \ell \le j < i \ \to \ a[j] \ne e) \tag{1}$$
$$\to \ (i \le u \ \to \ (a[i] = e \ \to \ \exists j. \ \ell \le j \le u \wedge a[j] = e))$$

or, equivalently,

$$\ell \le i \ \wedge \ (\forall j. \ \ell \le j < i \ \to \ a[j] \ne e) \ \wedge \ i \le u \ \wedge \ a[i] = e \tag{2}$$
$$\to \ \exists j. \ \ell \le j \le u \ \wedge \ a[j] = e$$

according to the equivalence

$$F_1 \wedge F_2 \ \to \ (F_3 \ \to \ (F_4 \ \to \ F_5))$$
$$\Leftrightarrow \ (F_1 \wedge F_2 \wedge F_3 \wedge F_4) \ \to \ F_5 \ .$$

This formula (2) is ($T_{\mathbb{Z}} \cup T_A$)-valid.

Example 2: Shortcut (backward substitution)

VC: $$\boxed{\underbrace{1 \leq i \wedge (\forall j.A[j])}_{F} \wedge i \leq u \wedge a[i] = e \rightarrow (\exists j.B[j])}$$

$@L:$ $F:$ $1 \leq i \wedge \forall j. \underbrace{1 \leq j < i \rightarrow a[j] \neq e}_{A[j]}$

$$i \leq u \wedge a[i] = e \rightarrow (\exists j.B[j])$$

$S_1:$ assume $i \leq u$;

$$a[i] = e \rightarrow (\exists j.B[j])$$

⇑

Example 2: Shortcut (backward substitution), cont.

S_1 : assume $i \leq u$;
$$a[i] = e \rightarrow (\exists j.B[j])$$

S_2 : assume $a[i] = e$;
$$\text{true} \leftrightarrow (\exists j.B[j]) \qquad i.e. \qquad (\exists j.B[j]))$$

S_3 : $rv := \text{true}$;
$$rv \leftrightarrow (\exists j.B[j])$$

@post G : $rv \leftrightarrow \exists j.\underbrace{1 \leq j \leq u \ \wedge \ a[j] = e}_{B[j]}$

$\Uparrow$

P-invariant and P-inductive I

Consider program P with function f s.t.

  function precondition F_{pre} and

  initial location L_0.

A P-computation is a sequence of states

  $s_0, s_1, s_2, \ldots$

  such that

  ► $s_0[pc] = L_0$ and $s_0 \models F_{pre}$, and

  ► for each i, s_{i+1} is the result of executing the instruction at
    $s_i[pc]$ on state s_i.

  where $s_i[pc] = $ value of pc given by state s_i.

P-invariant and P-inductive II

A formula F annotating location L of program P is <u>P-invariant</u> if for all P-computations $s_0, s_1, s_2, \ldots$ and for each index i,

$$s_i[pc] = L \quad \Rightarrow \quad s_i \models F$$

Annotations of P are <u>P-invariant</u> iff each annotation of P is P-invariant at its location.

<u>Not Implementable</u>: checking if F is P-invariant requires an infinite number of P-computations in general.

Annotations of P are <u>P-inductive</u> iff all VCs generated from the basic paths of program P are T-valid

$$P\text{-inductive} \quad \Rightarrow \quad P\text{-invariant}$$

<u>In Practice</u>: we check if the annotations are P-inductive.

Theorem (Verification Conditions)

If for every basic path

$@L_1:\ F$
$S_1;$
$\vdots$
$S_n;$
$@L_j:\ G$

of program P, the verification condition

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

is T-valid, then the annotations are P-inductive, and therefore P-invariant.

<u>Partial Correctness</u>: For program P, if there is a P-invariant annotation, then P is partially correct.

Total Correctness

Total Correctness = Partial Correctness + Termination

For every input that satisfies F_{pre}, the program eventually halts and produces output that satisfies F_{post}.

Proving function termination:

- Choose set W with well-founded relation $\prec$
  Usually set of n-tuples of natural numbers with the lexicographic relation $<_n$

- Find function δ (ranking function)
  mapping

  program states $\rightarrow$ W

  such that δ decreases according to $\prec$ along every basic path.

Since $\prec$ is well-founded, there cannot exist an infinite sequence of program states. The program must terminate.

Showing decrease of ranking function

For basic path with ranking function

$$\begin{aligned}
&@\ F \\
&\downarrow\ \delta[\overline{x}] \qquad \text{... ranking function} \\
&S_1; \\
&\vdots \\
&S_k; \\
&\downarrow\ \kappa[\overline{x}] \qquad \text{... ranking function}
\end{aligned}$$

We must prove that

the value of $\kappa \in W$ after executing $S_1; \cdots ; S_n$

is less than

the value of $\delta \in W$ before executing the statements

Thus, we show the verification condition

$$F\ \rightarrow\ \text{wp}(\kappa \prec \delta[\overline{x}_0],\ S_1; \cdots ; S_k)\{\overline{x}_0\ \mapsto\ \overline{x}\}\ .$$

Example: BubbleSort — loops

Choose $(\mathbb{N}^2, <_2)$ as well-founded set

```
@pre ⊤
@post ⊤
int[] BubbleSort(int[] a₀) {
  int[] a := a₀;
  for
    @L₁ :  i + 1 ≥ 0
    ↓ (i + 1,  i + 1)          ... ranking function δ₁
    (int i := |a| − 1;  i > 0;  i := i − 1) {
```

```
      for
        @L_2 :  i + 1 ≥ 0 ∧ i − j ≥ 0
        ↓ (i + 1,  i − j)              . . . ranking function δ_2
        (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;
}
```

We have to prove

- loop invariants are inductive (we don't show here)
- function decreases along each basic path.

The relevant basic paths:

(1)

$@L_1: \ i + 1 \geq 0$

$\downarrow L_1: \ (i + 1, i + 1)$

assume $i > 0$;

$j := 0$;

$\downarrow L_2: \ (i + 1, i - j)$

Path **(1)**:

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

$$\text{(2, 3)}$$

$@L_2: \ i + 1 \geq 0 \wedge i - j \geq 0$

$\downarrow L_2: \ (i+1, i-j)$

`assume` $j < i;$

$\ldots$

$j := j + 1;$

$\downarrow L_2: \ (i+1, i-j)$

Paths **(2)** and **(3)**:

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

(4)

$$@L_2 : \ i+1 \geq 0 \land i-j \geq 0$$
$$\downarrow L_2 : \ (i+1, i-j)$$
$$\texttt{assume } j \geq i;$$
$$i := i-1;$$
$$\downarrow L_1 : \ (i+1, i+1)$$

Path **(4)**:

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

All VCs are valid. Hence, BubbleSort always halts.

Construction of last VC

The verification condition for Path (4) is generated as follows:

$$wp((i+1, i+1) <_2 (i_0+1, i_0-j_0), \texttt{assume } j \geq i; i := i-1)$$
$$\Leftrightarrow \quad wp(((i-1)+1, (i-1)+1) <_2 (i_0+1, i_0-j_0), \texttt{assume } j \geq i)$$
$$\Leftrightarrow \quad j \geq i \;\rightarrow\; (i, i) <_2 (i_0+1, i_0-j_0)$$

Replace back $(i_0, j_0) \;\rightarrow\; (i, j)$:

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

producing the VC

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

Example 3: Shortcut (backward substitution)

VC: $\boxed{i + 1 \geq 0 \land i - j \geq 0 \land j \geq i \rightarrow (i, i) <_2 (i + 1, i - j)}$

$i + 1 \geq 0 \land i - j \geq 0 \land j \geq i \rightarrow (i, i) <_2 (i_0 + 1, i_0 - j_0)$

$@L_2 : \ i + 1 \geq 0 \land i - j \geq 0$

$$j \geq i \rightarrow (i, i) <_2 (i_0 + 1, i_0 - j_0)$$

$\downarrow L_2 : \ (i + 1, i - j)$

$$j \geq i \rightarrow (i, i) <_2 (i_0 + 1, i_0 - j_0)$$

assume $j \geq i$;

$$(i, i) <_2 (i_0 + 1, i_0 - j_0)$$

$i := i - 1$;

$$(i + 1, i + 1) <_2 (i_0 + 1, i_0 - j_0)$$

$\downarrow L_1 : \ (i + 1, i + 1)$ $\Uparrow$

Example 3: Shortcut (backward substitution)

VC: $\boxed{i + 1 \geq 0 \land i - j \geq 0 \land j \geq i \rightarrow (i, i) <_2 (i + 1, i - j)}$

$@L_2: \quad i + 1 \geq 0 \land i - j \geq 0$
$$j \geq i \rightarrow (i, i) <_2 (i + 1, i - j)$$

$\downarrow L_2: \quad (i + 1, i - j)$
$$j \geq i \rightarrow (i, i) <_2 \ ?$$

assume $j \geq i$;
$$(i, i) <_2 \ ?$$

$i := i - 1$;
$$(i + 1, i + 1) <_2 \ ?$$

$\downarrow L_1: \quad (i + 1, i + 1)$ $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \Uparrow$

Example: Binary Search — recursive calls

Choose $(\mathbb{N}, <)$ as well-founded set and ranking function $\delta : u - \ell + 1$

```
@pre u − ℓ + 1 ≥ 0
@post ⊤
↓ u − ℓ + 1        . . . ranking function δ
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
      @R₁ : u − (m + 1) + 1 ≥ 0
      BinarySearch(a, m + 1, u, e);
    else return
      @R₂ : (m − 1) − ℓ + 1 ≥ 0
      BinarySearch(a, ℓ, m − 1, e);
  }
}
```

Show @R_1 and @R_2 are P-invariant

Show decrease in $u - \ell + 1$:

(1)

@pre $u - \ell + 1 \geq 0$

$\downarrow u - \ell + 1$

assume $\ell \leq u$;

$m := (\ell + u)$ div 2;

assume $a[m] \neq e$;

assume $a[m] < e$;

$\downarrow u - (m + 1) + 1$

Verification condition:

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

Show decrease in $u - \ell + 1$:

(2)

@pre $u - \ell + 1 \geq 0$

$\downarrow u - \ell + 1$

assume $\ell \leq u$;

$m := (\ell + u)$ div 2;

assume $a[m] \neq e$;

assume $a[m] \geq e$;

$\downarrow (m - 1) - \ell + 1$

Verification condition:

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

<u>Note</u>: two other basic paths (... `return false` and
... `return true`) are irrelevant to the termination argument
(recursion ends at each).

Both VCs are $T_{\mathbb{Z}}$-valid. Thus BinarySearch halts on all input in
which ℓ is initially at most $u + 1$.