

CS156: The Calculus of Computation

Zohar Manna
Winter 2010

Chapter 5: Program Correctness: Mechanics

Program A: LinearSearch with function specification

@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) {  
  for @ T  
    (int i := l; i ≤ u; i := i + 1) {  
      if (a[i] = e) return true;  
    }  
  return false;  
}
```

Function LinearSearch searches subarray of array a of integers for specified value e .

Function specifications

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

for loop: 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 $0 \leq \ell \wedge u < |a| \wedge \text{sorted}(a, \ell, u)$

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

```
bool BinarySearch(int[] a, int  $\ell$ , int  $u$ , int  $e$ ) {  
    if ( $\ell > u$ ) return false;  
    else {  
        int  $m := (\ell + u) \text{ div } 2$ ;  
        if ( $a[m] = e$ ) return true;  
        else if ( $a[m] < e$ ) return BinarySearch( $a, m + 1, u, e$ );  
        else return BinarySearch( $a, \ell, m - 1, e$ );  
    }  
}
```

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

sorted: weakly increasing order, i.e.

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

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

Function specifications

- ▶ Function precondition (*@pre*)

It behaves correctly only if

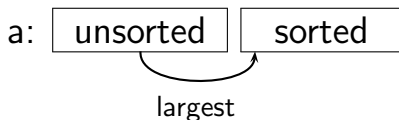
$$0 \leq \ell \text{ and } u < |a| \text{ and} \\ \text{sorted}(a, \ell, u).$$

- ▶ Function postcondition (*@post*)

It returns true iff a contains the value e in the range $[\ell, u]$

```
@pre T
@post sorted(rv, 0, |rv| - 1)
int[] BubbleSort(int[] a0) {
  int[] a := a0;
  for @ T
    (int i := |a| - 1; i > 0; i := i - 1) {
      for @ T
        (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 BubbleSort 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

Function specification

- ▶ Function postcondition (@post):

BubbleSort returns array rv sorted on the range $[0, |rv| - 1]$.


¹Except 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
<i>j</i>				<i>i</i>		

2	3	4	1	2	5	6
	<i>j</i>			<i>i</i>		

2	3	4	1	2	5	6
		<i>j</i>		<i>i</i>		



2	3	1	4	2	5	6
			<i>j</i>	<i>i</i>		



2	3	1	2	4	5	6
				<i>j, i</i>		

2	3	1	2	4	5	6
<i>j</i>			<i>i</i>			

Program Annotation

▶ Function Specifications

function precondition (@pre)

function postcondition (@post)

▶ Runtime Assertions

e.g., $@ 0 \leq j < |a| \wedge 0 \leq j + 1 < |a|$
 $a[j] := a[j + 1]$

▶ Loop Invariants

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

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

Program A: LinearSearch with runtime assertions

@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) {
```

```
  for
```

```
    @ L : T
```

```
    (int i := l; i ≤ u; i := i + 1) {
```

```
      @  $0 \leq i < |a|$ ;
```

```
      if (a[i] = e) return true;
```

```
    }
```

```
  return false;
```

```
}
```

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

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

```
bool BinarySearch(int[] a, int  $\ell$ , int  $u$ , int  $e$ ) {  
    if ( $\ell > u$ ) return false;  
    else {  
        @  $2 \neq 0$ ;  
        int  $m := (\ell + u) \text{ div } 2$ ;  
        @  $0 \leq m < |a|$ ;  
        if ( $a[m] = e$ ) return true;  
        else {  
            @  $0 \leq m < |a|$ ;  
            if ( $a[m] < e$ ) return BinarySearch( $a, m + 1, u, e$ );  
            else return BinarySearch( $a, \ell, m - 1, e$ );  
        }  
    }  
}
```

```
@pre T
@post sorted(rv, 0, |rv| - 1)
int[] BubbleSort(int[] a0) {
    int[] a := a0;
    for
        @ L1 : T
        (int i := |a| - 1; i > 0; i := i - 1) {
            for
                @ L2 : T
                (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
  @  $F$ 
  {  $\langle cond \rangle$  {  $\langle body \rangle$  }
```

- ▶ apply $\langle body \rangle$ as long as $\langle cond \rangle$ holds
- ▶ assertion F holds at the beginning of every iteration evaluated before $\langle cond \rangle$ is checked

for		$\langle init \rangle$;
@ F		while
($\langle init \rangle$; $\langle cond \rangle$; $\langle incr \rangle$)		@ F
{ $\langle body \rangle$ }		$\langle cond \rangle$ { $\langle body \rangle$; $\langle incr \rangle$ }

Program A: LinearSearch with loop invariants

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

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

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

Proving Partial Correctness

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

- ▶ A program + annotation is reduced to finite set of verification conditions (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: $l \leq i \wedge \forall j. l \leq j < i \rightarrow a[j] \neq e$

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)

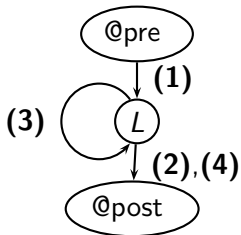
@L: $l \leq i \wedge \forall j. l \leq j < i \rightarrow a[j] \neq e$

assume $i > u$;

$rv := \text{false}$;

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

Visualization of basic paths of LinearSearch



L is a cutpoint.

Program C: BubbleSort with loop invariants

```
@pre |a0| > 0
@post sorted(rv, 0, |rv| - 1)
int[] BubbleSort(int[] a0) {
    int[] a := a0;
    for
        @L1 :  $\left[ \begin{array}{l} 0 \leq i < |a| \\ \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge \text{sorted}(a, i, |a| - 1) \end{array} \right]$ 
        (int i := |a| - 1; i > 0; i := i - 1) {
```

```

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

    (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

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)

@pre $|a_0| > 0$

$a := a_0;$

$i := |a| - 1;$

@L₁ : $\left[\begin{array}{l} 0 \leq i < |a| \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge \text{sorted}(a, i, |a| - 1) \end{array} \right]$

(2)

$@L_1 : \left[\begin{array}{l} 0 \leq i < |a| \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge \text{sorted}(a, i, |a| - 1) \end{array} \right]$

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

$t := a[j]$;

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

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

$j := j + 1$;

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

(4)

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

assume $j < i$;

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

$j := j + 1$;

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

(5)

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

assume $j \geq i$;

$i := i - 1$;

$@L_1 : \left[\begin{array}{l} 0 \leq i < |a| \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge \text{sorted}(a, i, |a| - 1) \end{array} \right]$

(6)

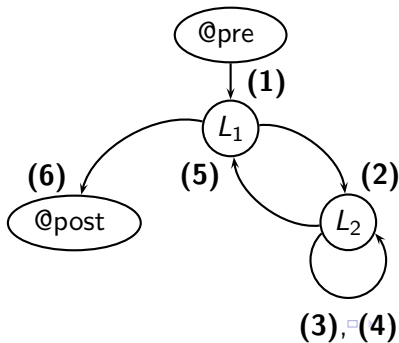
$@L_1 : \left[\begin{array}{l} 0 \leq i < |a| \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge \text{sorted}(a, i, |a| - 1) \end{array} \right]$

assume $i \leq 0$;

$rv := a$;

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

Visualization of basic paths of BubbleSort



L_1 and L_2 are cut points

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  $0 \leq \ell \wedge u < |a| \wedge \text{sorted}(a, \ell, u)$  ...  $F[a, \ell, u, e]$ 
    :
@R1 :  $0 \leq m + 1 \wedge u < |a| \wedge \text{sorted}(a, m + 1, u)$  ...  $F[a, m + 1, u, e]$ 
return BinarySearch( $a, m + 1, u, e$ )
    :
@R2 :  $0 \leq \ell \wedge m - 1 < |a| \wedge \text{sorted}(a, \ell, m - 1)$  ...  $F[a, \ell, m - 1, e]$ 
return BinarySearch( $a, \ell, m - 1, e$ )
```

```
@pre  $0 \leq \ell \wedge u < |a| \wedge \text{sorted}(a, \ell, u)$ 
@post  $rv \leftrightarrow \exists i. \ell \leq i \leq u \wedge a[i] = e$ 
bool BinarySearch(int[] a, int  $\ell$ , int  $u$ , int  $e$ ) {
    if ( $\ell > u$ ) return false;
    else {
        int  $m := (\ell + u) \text{ div } 2$ ;
        if ( $a[m] = e$ ) return true;
        else if ( $a[m] < e$ ) {
            @R1 :  $0 \leq m + 1 \wedge u < |a| \wedge \text{sorted}(a, m + 1, u)$ ;
            return BinarySearch(a,  $m + 1$ ,  $u$ ,  $e$ );
        } else {
            @R2 :  $0 \leq \ell \wedge m - 1 < |a| \wedge \text{sorted}(a, \ell, m - 1)$ ;
            return BinarySearch(a,  $\ell$ ,  $m - 1$ ,  $e$ );
        }
    }
}
```

(1)

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

assume $l > u$;

$rv := \text{false}$;

@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 $l \leq u$;

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

assume $a[m] = e$;

$rv := \text{true}$;

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

(3)

@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 $a[m] < e$;

@R₁ : $0 \leq m + 1 \wedge u < |a| \wedge \text{sorted}(a, m + 1, u)$

(5)

@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 $a[m] \geq e$;

@R₂ : $0 \leq \ell \wedge m - 1 < |a| \wedge \text{sorted}(a, \ell, m - 1)$

(4)

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

assume $l \leq u$;

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

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 $l \leq u$;

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

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 $rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$

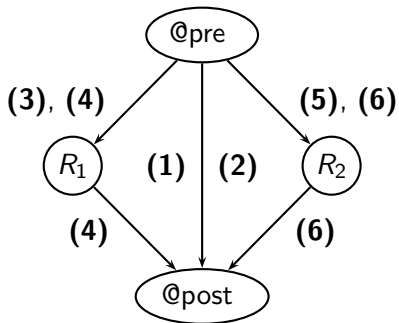


Figure: Visualization of basic paths of BinarySearch

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.

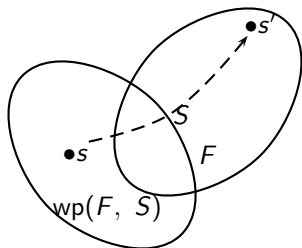
Weakest Precondition $wp(F, S)$

For FOL formula F , program statement S ,

$s \models wp(F, S)$ iff

statement S is executed on state s to produce state s' ,

and $s' \models F$:



▶ $wp(F, \text{assume } c) \Leftrightarrow c \rightarrow F$

▶ $wp(F[v], v := e) \Leftrightarrow F[e]$

▶ 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

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

Also denoted by

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

That is, for every state s ,

if $s \models F$

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

Example: Basic path

(1)

@ $F : x \geq 0$

$S_1 : x := x + 1;$

@ $G : x \geq 1$

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

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

Therefore the VC of path (1) is

$$\underbrace{x \geq 0}_F \rightarrow \underbrace{x \geq 0}_{\text{wp}(G, S_1)},$$

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

Example 1: Shortcut (backward substitution)

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

$$\textcircled{F} : x \geq 0$$

$$x + 1 \geq 1 \quad \text{i.e.} \quad \underbrace{x \geq 0}_{\text{wp}(G, S_1)}$$

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

$$x \geq 1$$

$$\textcircled{G} : x \geq 1$$



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 : assume $i \leq u$;

S_2 : assume $a[i] = e$;

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

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

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

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

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

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

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

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

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

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

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

Therefore the VC of path (2) is

$$\begin{aligned} & \ell \leq i \wedge (\forall j. \ell \leq j < i \rightarrow a[j] \neq e) \\ & \rightarrow (i \leq u \rightarrow (a[i] = e \rightarrow \exists j. \ell \leq j \leq u \wedge a[j] = e)) \end{aligned} \quad (1)$$

or, equivalently,

$$\begin{aligned} & \ell \leq i \wedge (\forall j. \ell \leq j < i \rightarrow a[j] \neq e) \wedge i \leq u \wedge a[i] = e \\ & \rightarrow \exists j. \ell \leq j \leq u \wedge a[j] = e \end{aligned} \quad (2)$$

according to the equivalence

$$\begin{aligned} & F_1 \wedge F_2 \rightarrow (F_3 \rightarrow (F_4 \rightarrow F_5)) \\ \Leftrightarrow & (F_1 \wedge F_2 \wedge F_3 \wedge F_4) \rightarrow F_5 . \end{aligned}$$

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

Example 2: Shortcut (backward substitution)

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

$$\textcircled{L}: F : l \leq i \wedge \underbrace{\forall j. 1 \leq j < i \rightarrow a[j] \neq e}_{A[j]} \\ i \leq u \wedge a[i] = e \rightarrow (\exists j. B[j])$$

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

↑

Example 2: Shortcut (backward substitution), cont.

S_1 : assume $i \leq u$;

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

S_2 : assume $a[i] = e$;

$$\text{true} \leftrightarrow (\exists j. B[j]) \quad \text{i.e.} \quad (\exists j. B[j])$$

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

$$rv \leftrightarrow (\exists j. B[j])$$

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

↑

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, \dots

such that

- ▶ $s_0[pc] = L_0$ and $s_0 \models F_{\text{pre}}$, 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 P -invariant if for all P -computations s_0, s_1, s_2, \dots and for each index i ,

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

Annotations of P are P -invariant iff each annotation of P is P -invariant at its location.

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

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

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

In Practice: we check if the annotations are P -inductive.

Theorem (Verification Conditions)

If for every basic path

@ L_1 : F

S_1 ;

⋮

S_n ;

@ L_j : G

of program P , the verification condition

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

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

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

Total Correctness

$$\underline{\text{Total Correctness}} = \underline{\text{Partial Correctness}} + \underline{\text{Termination}}$$

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

@ F

$\downarrow \delta[\bar{x}]$... ranking function

$S_1;$

\vdots

$S_k;$

$\downarrow \kappa[\bar{x}]$... ranking function

We must prove that

the value of $\kappa \in W$ after executing $S_1; \dots ; S_n$
is less than

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

Thus, we show the verification condition

$$F \rightarrow \text{wp}(\kappa < \delta[\bar{x}_0], S_1; \dots ; S_k) \{ \bar{x}_0 \mapsto \bar{x} \} .$$

Example: BubbleSort — loops

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

@pre \top

@post \top

int[] BubbleSort(int[] a_0) {

 int[] $a := a_0$;

 for

 @ $L_1 : i + 1 \geq 0$

$\downarrow (i + 1, i + 1)$... ranking function δ_1

 (int $i := |a| - 1; i > 0; i := i - 1$) {

```

for
  @L2 :  $i + 1 \geq 0 \wedge i - j \geq 0$ 
  ↓ ( $i + 1, i - j$ )          ... ranking function  $\delta_2$ 
  (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):

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

(2, 3)

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

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

assume $j < i$;

...

$j := j + 1$;

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

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

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

assume $j \geq i$;

$i := i - 1$;

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

Path (4):

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

All VCs are valid. Hence, BubbleSort always halts.

Construction of last VC

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

$$\begin{aligned} & wp((i + 1, i + 1) <_2 (i_0 + 1, i_0 - j_0), \text{assume } j \geq i; i := i - 1) \\ \Leftrightarrow & wp(((i - 1) + 1, (i - 1) + 1) <_2 (i_0 + 1, i_0 - j_0), \text{assume } j \geq i) \\ \Leftrightarrow & j \geq i \rightarrow (i, i) <_2 (i_0 + 1, i_0 - j_0) \end{aligned}$$

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)

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

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

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

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

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

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

assume $j \geq i$;

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

$i := i - 1$;

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

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

↑

Example 3: Shortcut (backward substitution)

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

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

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

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

$$j \geq i \rightarrow (i, i) <_2 ?$$

assume $j \geq i$;

$$(i, i) <_2 ?$$

$i := i - 1$;

$$(i + 1, i + 1) <_2 ?$$

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



Example: Binary Search — recursive calls

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

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

@post \top

$\downarrow u - \ell + 1$... ranking function δ

```
bool BinarySearch(int[] a, int  $\ell$ , int  $u$ , int  $e$ ) {
```

```
    if ( $\ell > u$ ) return false;
```

```
    else {
```

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

```
        if ( $a[m] = e$ ) return true;
```

```
        else if ( $a[m] < e$ ) return
```

```
            @ $R_1 : u - (m + 1) + 1 \geq 0$ 
```

```
            BinarySearch( $a, m + 1, u, e$ );
```

```
        else return
```

```
            @ $R_2 : (m - 1) - \ell + 1 \geq 0$ 
```

```
            BinarySearch( $a, \ell, m - 1, e$ );
```

```
    }
```

```
}
```

Show $@R_1$ and $@R_2$ are P -invariant

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

$$\text{(1)} \quad \frac{\begin{array}{l} @pre \ u - \ell + 1 \geq 0 \\ \downarrow \ u - \ell + 1 \\ \text{assume } \ell \leq u; \\ m := (\ell + u) \text{ div } 2; \\ \text{assume } a[m] \neq e; \\ \text{assume } a[m] < e; \\ \downarrow \ u - (m + 1) + 1 \end{array}}{\quad}$$

Verification condition:

$$\begin{aligned} & u - \ell + 1 \geq 0 \wedge \ell \leq u \wedge \dots \\ \rightarrow & u - (((\ell + u) \text{ div } 2) + 1) + 1 < u - \ell + 1 \end{aligned}$$

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

(2)

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

↓ $u - \ell + 1$

assume $l \leq u$;

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

assume $a[m] \neq e$;

assume $a[m] \geq e$;

↓ $(m - 1) - \ell + 1$

Verification condition:

$$\begin{aligned} & u - \ell + 1 \geq 0 \wedge l \leq u \wedge \dots \\ \rightarrow & (((l + u) \text{ div } 2) - 1) - \ell + 1 < u - \ell + 1 \end{aligned}$$

Note: 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 $u + 1$.