

THE CALCULUS OF COMPUTATION:  
Decision Procedures with  
Applications to Verification

by  
Aaron Bradley  
Zohar Manna

Springer 2007

## 5. Program Correctness: Mechanics

## Program A: LinearSearch with function specification

---

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

---

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

### Function specifications

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

for loop: initially set  $i$  to be  $\ell$ ,  
execute the body and increment  $i$  by 1  
as long as  $i \leq n$

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

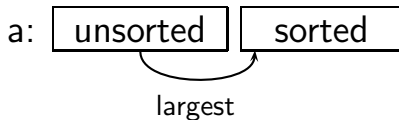
## Program C: BubbleSort with function specification

---

```
@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.



# Program Annotation

- ▶ Function Specifications

function postcondition ( $@post$ )

function precondition ( $@pre$ )

- ▶ Runtime Assertions

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

- ▶ Loop Invariants

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

## Program A: LinearSearch with runtime assertions

---

```
@pre T
@post T
bool LinearSearch(int[] a, int l, int u, int e) {
  for @ T
    (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 T
@post T
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  $\top$ 
@post  $\top$ 
int[] BubbleSort(int[] a0) {
    int[] a := a0;
    for @  $\top$ 
        (int i := |a| - 1; i > 0; i := i - 1) {
            for @  $\top$ 
                (int j := 0; j < i; j := j + 1) {
                    @  $0 \leq j < |a| \wedge 0 \leq j + 1 < |a|$ ;
                    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
  @  $F$ 
  ( $\langle init \rangle$ ;  $\langle cond \rangle$ ;  $\langle incr \rangle$ ) {  $\langle body \rangle$  }
```

$\Rightarrow$

```
 $\langle init \rangle$ ;  
while  
  @  $F$   
   $\langle cond \rangle$  {  $\langle body \rangle$   $\langle incr \rangle$  }
```

## Program A: LinearSearch with loop invariants

---

```
@pre  $0 \leq \ell \wedge u < |a|$ 
@post  $rv \leftrightarrow \exists i. \ell \leq i \leq u \wedge a[i] = e$ 
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 function's precondition is satisfied on entry, its postcondition is satisfied when the function halts.
- ▶ A function + annotation is reduced to finite set of verification conditions (VCs), FOL formulae
  - ▶ If all VCs are valid, then the function obeys its specification (partially correct)

## Basic Paths: Loops

To handle loops, we break the function into basic paths

@ ← precondition or loop invariant

sequence of instructions  
(with no loop invariants)

@ ← loop invariant, assertion, or postcondition

# Program A: LinearSearch

## Basic Paths of LinearSearch

---

(1)

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

$i := l$ ;

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

---

(2)

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

assume  $i \leq u$ ;

assume  $a[i] = e$ ;

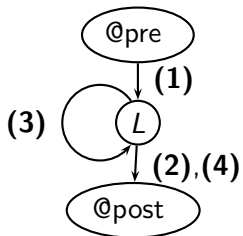
$rv := \text{true}$ ;

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

---



## Visualization of basic paths of LinearSearch



## Program C: BubbleSort with loop invariants

---

@pre  $\top$

@post sorted( $rv, 0, |rv| - 1$ )

int[] BubbleSort(int[]  $a_0$ ) {

  int[]  $a := a_0$ ;

  for

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

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

```

for
    @L2 : 
$$\left[ \begin{array}{l} 1 \leq i < |a| \wedge 0 \leq j \leq i \\ \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge \text{partitioned}(a, 0, j - 1, j, j) \\ \wedge \text{sorted}(a, i, |a| - 1) \end{array} \right]$$

    (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  $\top$ ;

$a := a_0$ ;

$i := |a| - 1$ ;

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

$\wedge$  sorted( $a, i, |a| - 1$ )

---

---

(2)

---

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

assume  $i > 0$ ;

$j := 0$ ;

@L<sub>2</sub> :  $\left[ \begin{array}{l} 1 \leq i < |a| \wedge 0 \leq j \leq i \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge \text{partitioned}(a, 0, j - 1, j, j) \wedge \text{sorted}(a, i, |a| - 1) \end{array} \right]$

---

(3)

---

@L<sub>2</sub> :  $\left[ \begin{array}{l} 1 \leq i < |a| \wedge 0 \leq j \leq i \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge \text{partitioned}(a, 0, j - 1, j, j) \wedge \text{sorted}(a, i, |a| - 1) \end{array} \right]$

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<sub>2</sub> :  $\left[ \begin{array}{l} 1 \leq i < |a| \wedge 0 \leq j \leq i \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge \text{partitioned}(a, 0, j - 1, j, j) \wedge \text{sorted}(a, i, |a| - 1) \end{array} \right]$

---

---

(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  $j \geq i$ ;

$i := i - 1$ ;

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

---

---

(6)

---

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

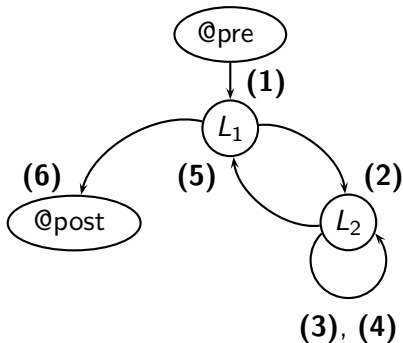
assume  $i \leq 0$ ;

$rv := a$ ;

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

---

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  $0 \leq \ell \wedge u < |a| \wedge \text{sorted}(a, \ell, u)$       ...  $F[a, \ell, u, e]$   
  :  
@R<sub>1</sub> :  $0 \leq m + 1 \wedge u < |a| \wedge \text{sorted}(a, m + 1, u)$     ...  $F[a, m + 1, u, e]$   
  return BinarySearch( $a, m + 1, u, e$ )  
  :  
@R<sub>2</sub> :  $0 \leq \ell \wedge m - 1 < |a| \wedge \text{sorted}(a, \ell, m - 1)$     ...  $F[a, \ell, m - 1, e]$   
  return BinarySearch( $a, \ell, m - 1, e$ )

## Program B: BinarySearch with function call assertions

---

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

---

# Verification Conditions

- ▶ Program counter  $pc$  — holds current location of control
- ▶ State  $s$  — assignment of values to all variables

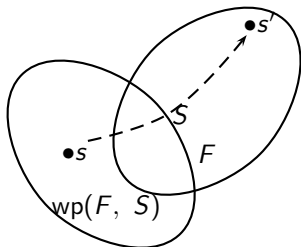
Example: Control resides at  $L_1$  of BubbleSort

$$s : \{pc \mapsto L_1, a \mapsto [2; 0; 1], i \mapsto 2, j \mapsto 0, \\ t \mapsto 2, rv \mapsto []\}$$

- ▶ Weakest precondition  $wp(F, S)$

For FOL formula  $F$ , program statement  $S$ ,

If  $s \models wp(F, S)$  and if statement  $S$  is executed on state  $s$  to produce state  $s'$ , then  $s' \models F$



## Weakest Precondition $wp(F, S)$

- ▶  $wp(F, \text{assume } c) \Leftrightarrow c \rightarrow F$
- ▶  $wp(F[v], v := e) \Leftrightarrow F[e]$
- ▶ 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 Condition of basic path

@  $F$

$S_1$ ;

...

$S_n$ ;

@  $G$

is

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

Also denoted by

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

## Example: Basic path

(1)

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

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

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

The VC is

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

That is,

$$\text{wp}(G, S_1)$$

$$\Leftrightarrow \text{wp}(x \geq 1, x := x + 1)$$

$$\Leftrightarrow (x \geq 1)\{x \mapsto x + 1\}$$

$$\Leftrightarrow x + 1 \geq 1$$

$$\Leftrightarrow x \geq 0$$

Therefore the VC of path (1)

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

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

## Example: Basic path (2) of LinearSearch

(2)

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

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

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

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

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

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)

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

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

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

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

## $P$ -invariant and $P$ -inductive

Consider program  $P$  with function  $f$  s.t.  
function precondition  $F_0$  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  $s_0 \models F_0$ , 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  $s_i$

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 (invariant) iff each annotation of  $P$  is  $P$ -invariant at its location.

Annotations of  $P$  are  $P$ -inductive (inductive) iff all VCs generated from program  $P$  are  $T$ -valid

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

# Total Correctness

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

Given that the input satisfies the function precondition, the function eventually halts and produces output that satisfies the function postcondition.

Proving function termination:

- ▶ Choose set  $S$  with well-founded relation  $\prec$   
Usually set of  $n$ -tuples of natural numbers with the lexicographic extension  $<_n$
- ▶ Find function  $\delta$  (ranking function)  
mapping  
program states  $\rightarrow S$   
such that  $\delta$  decreases according to  $\prec$  along every basic path.

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

## Choosing well-founded relation and ranking function

Example: Ackermann function — recursive calls

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

---

@pre  $x \geq 0 \wedge y \geq 0$

@post  $rv \geq 0$

↓  $(x, y)$                     ... ranking function  $\delta : (x, y)$

```
int Ack(int x, int y) {  
    if (x = 0) {  
        return y + 1;  
    }  
    else if (y = 0) {  
        return Ack(x - 1, 1);  
    }  
    else {  
        int z := Ack(x, y - 1);  
        return Ack(x - 1, z);  
    }  
}
```

---

- ▶ Show  $\delta : (x, y)$  maps into  $\mathbb{N}^2$ , i.e.,  
 $x \geq 0$  and  $y \geq 0$  are invariants
- ▶ Show  $\delta : (x, y)$  decreases from function entry to each recursive call. We show this.

The basic paths are:

---

(1)

@pre  $x \geq 0 \wedge y \geq 0$

↓  $(x, y)$

assume  $x \neq 0$ ;

assume  $y = 0$ ;

↓  $(x - 1, 1)$

---

(2)

@pre  $x \geq 0 \wedge y \geq 0$

↓  $(x, y)$

assume  $x \neq 0$ ;

assume  $y \neq 0$ ;

↓  $(x, y - 1)$

---

---

(3)

@pre  $x \geq 0 \wedge y \geq 0$

↓  $(x, y)$

assume  $x \neq 0$ ;

assume  $y \neq 0$ ;

assume  $v_1 \geq 0$ ;

$z := v_1$ ;

↓  $(x - 1, z)$

---

## Showing decrease of ranking function

For basic path with ranking function

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

We must prove that

the value of  $\kappa$  after executing  $S_1; \dots ; S_n$

is less than

the value of  $\delta$  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: Ackermann function — recursive calls

Verification conditions for the three basic paths

1.  $x \geq 0 \wedge y \geq 0 \wedge x \neq 0 \wedge y = 0 \Rightarrow (x - 1, 1) <_2 (x, y)$
2.  $x \geq 0 \wedge y \geq 0 \wedge x \neq 0 \wedge y \neq 0 \Rightarrow (x, y - 1) <_2 (x, y)$
3.  $x \geq 0 \wedge y \geq 0 \wedge x \neq 0 \wedge y \neq 0 \wedge v_1 \geq 0 \Rightarrow$   
 $(x - 1, v_1) <_2 (x, y)$

Then compute

$$\begin{aligned} & \text{wp}((x - 1, z) <_2 (x_0, y_0) \\ & \quad , \text{assume } x \neq 0; \text{assume } y \neq 0; \text{assume } v_1 \geq 0; z := v_1) \\ & \Leftrightarrow \text{wp}((x - 1, v_1) <_2 (x_0, y_0) \\ & \quad , \text{assume } x \neq 0; \text{assume } y \neq 0; \text{assume } v_1 \geq 0) \\ & \Leftrightarrow x \neq 0 \wedge y \neq 0 \wedge v_1 \geq 0 \rightarrow (x - 1, v_1) <_2 (x_0, y_0) \end{aligned}$$

Renaming  $x_0$  and  $y_0$  to  $x$  and  $y$ , respectively, gives

$$x \neq 0 \wedge y \neq 0 \wedge v_1 \geq 0 \rightarrow (x - 1, v_1) <_2 (x, y) .$$

Noting that path **(3)** begins by asserting  $x \geq 0 \wedge y \geq 0$ , we finally have

$$x \geq 0 \wedge y \geq 0 \wedge x \neq 0 \wedge y \neq 0 \wedge v_1 \geq 0 \Rightarrow (x - 1, v_1) <_2 (x, y) .$$

## Example: BubbleSort — loops

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

---

```
@pre T
@post T
int[] BubbleSort(int[] a0) {
  int[] a := a0;
  for
    @L1 :  $i + 1 \geq 0$ 
    ↓  $(i + 1, i + 1)$            ... ranking function  $\delta_1$ 
    (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
- ▶ 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  $i > 0$ ;

$j := 0$ ;

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

---

(2),(3)

---

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

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

assume  $j < i$ ;

...

$j := j + 1$ ;

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

---

---

(4)

---

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

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

assume  $j \geq i$ ;

$i := i - 1$ ;

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

---

### Verification conditions

#### Path (1)

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

#### Paths (2) and (3)

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

#### Path (4)

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

which are valid. Hence, BubbleSort always halts.