

# Decision Procedures

Jochen Hoenicke



Software Engineering  
Albert-Ludwigs-University Freiburg

Winter Term 2016/17

# Program Correctness

- So far: decision procedures to decide validity in theories
- In the next lectures: the “practical” part
- Application of decision procedures to program verification

- pi is an imperative programming language.
- built-in program annotations in first order logic
- annotation  $F$  at location  $L$  asserts that  $F$  is true whenever program control reaches  $L$

---

```
@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;
}
```

---

A function  $f$  is **partially correct** if  
when  $f$ 's precondition is satisfied on entry and  $f$  terminates,  
then  $f$ 's postcondition is satisfied.

- 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)

## Loop invariants

- Each loop needs an annotation  $@L$  called **loop invariant**
- while loop:  $L$  must hold
  - at the beginning of each iteration before the loop condition is evaluated
- for loop:  $L$  must hold
  - after the loop initialization, and
  - before the loop condition is evaluated

To handle loops, we break the function into **basic paths**.

@ ← precondition or loop invariant

finite sequence of instructions  
(with no loop invariants)

@ ← loop invariant, assertion, or postcondition



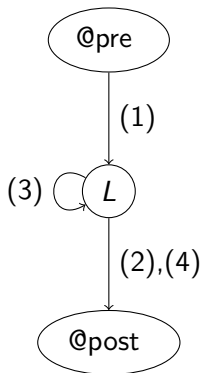
A basic path:

- begins at the function pre condition or a loop invariant,
- ends at an assertion, e.g., the loop invariant or the function post,
- does not contain the loop invariant inside the sequence,
- conditional branches are replaced by **assume statements**.

Assume statement  $c$

- Remainder of basic path is executed only if  $c$  holds
- Guards with condition  $c$  split the path ( $\text{assume}(c)$  and  $\text{assume}(\neg c)$ )

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

---

---

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

---

## Goal

- Prove that annotated function  $f$  agrees with annotations
- Therefore: Reduce  $f$  to finite set of **verification conditions** VC
- Validity of VC implies that function behaviour agrees with annotations

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

- Informally: What must hold before executing statement  $S$  to ensure that formula  $F$  holds afterwards?
- $wp(F, S)$  = weakest formula such that executing  $S$  results in formula that satisfies  $F$
- For all states  $s$  such that  $s \models wp(F, S)$ : successor state  $s' \models F$ .

## Computing weakest preconditions

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

## Verification Condition of basic path

@  $F$   
 $S_1$ ;  
...  
 $S_n$ ;  
@  $G$

is

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

## Proving partial correctness for programs with loops

- Input: Annotated program
- Produce all basic paths  $P = \{p_1, \dots, p_n\}$
- For all  $p \in P$ : generate verification condition  $VC(p)$
- Check validity of  $\bigwedge_{p \in P} VC(p)$

### Theorem

If  $\bigwedge_{p \in P} VC(p)$  is valid, then each function agrees with its annotation.

(1)

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

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

$$\textcircled{0} G : x \geq 1$$

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.



---

(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.

- Verifies pi programs
- Available at <http://cs.stanford.edu/people/jasonaue/pivc/>

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,  $T_{\mathbb{Z} \cup A}$

### 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  $u < |a|$

---

```
@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 := ( $l + u$ ) 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);
  }
}
```

---

---

```
@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 ( $a[m] < e$ ) {
      @pre  $0 \leq m + 1 \wedge u < |a| \wedge \text{sorted}(a, m + 1, u)$ ;
      bool  $tmp := \text{BinarySearch}(a, m + 1, u, e)$ ;
      @post  $tmp \leftrightarrow \exists i. m + 1 \leq i \leq u \wedge a[i] = e$ ; return  $tmp$ ;
    } else {
      @pre  $0 \leq \ell \wedge m - 1 < |a| \wedge \text{sorted}(a, \ell, m - 1)$ ;
      bool  $tmp := \text{BinarySearch}(a, \ell, m - 1, e)$ ;
      @post  $tmp \leftrightarrow \exists i. \ell \leq i \leq m - 1 \wedge a[i] = e$ ;
      return  $tmp$ ;
    }
  }
}
```

---

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

a: unsorted sorted

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.





```
@pre T
@post T
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) {
          @ 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;
}
```

---

## 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  $T_{\mathbb{Z}} \cup T_{\mathbb{A}}$ .

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> :  $\left[ 1 \leq i < |a| \wedge 0 \leq j \leq i \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \right]$   
 $\left[ \wedge \text{partitioned}(a, 0, j - 1, j, j) \wedge \text{sorted}(a, i, |a| - 1) \right]$

---

(3)

---

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

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> :  $\left[ 1 \leq i < |a| \wedge 0 \leq j \leq i \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \right]$   
 $\left[ \wedge \text{partitioned}(a, 0, j - 1, j, j) \wedge \text{sorted}(a, i, |a| - 1) \right]$

---

---

**(4)**

---

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

assume  $j < i$ ;

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

$j := j + 1$ ;

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

---

**(5)**

---

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

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)**

---

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

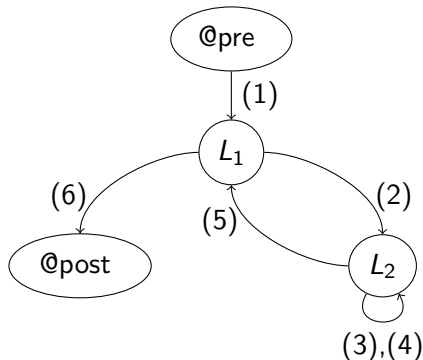
assume  $i \leq 0$ ;

$rv := a$ ;

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

---

Visualization of basic paths of BubbleSort





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)

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

**Total Correctness** = **Partial Correctness** + **Termination**

In the following, we focus on proving function termination. Therefore, we need the notion of **well-founded relations** and **ranking functions**.

## Definition

For a set  $S$ , a binary relation  $\prec$  is a **well-founded relation** iff there is no infinite sequence  $s_1, s_2, s_3 \dots$  of elements of  $S$  such that  $s_1 \succ s_2 \succ s_3 \succ \dots$ , where  $s \prec t$  iff  $t \succ s$ .

## Example

$<$  is well-founded over  $\mathbb{N}$ . Decreasing sequences w.r.t.  $<$  are always finite.

$123 > 98 > 42 > 11 > 7 > 2 > 0$

$<$  is not well-founded over  $\mathbb{Q}$ .

$1 > \frac{1}{2} > \frac{1}{3} > \frac{1}{4} > \dots$

- Choose set  $S$  with well-founded relation  $\prec$   
Usually set of  $n$ -tuples of natural numbers with the lexicographic ordering.
- Find function  $\delta$  such that
  - $\delta$  maps program states to  $S$ , and
  - $\delta$  decreases according to  $\prec$  along every basic path.Such a function  $\delta$  is called a **ranking function**.

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

**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  $\delta : (x, y) \mapsto (x, y)$

```
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);
  }
}
```

---

To prove function termination:

- 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.

The relevant 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 := v1;  
#(x - 1, z)
```

---

## Showing decrease of ranking function

Basic path with ranking function:

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

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

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



**Example:** Ackermann function — verification condition for basic path **(3)**

**(3)**

---

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

#  $(x, y)$

assume  $x \neq 0$ ;

assume  $y \neq 0$ ;

assume  $v_1 \geq 0$ ;

$z := v_1$ ;

#  $(x - 1, z)$

---

Verification condition:

$x \geq 0 \wedge y \geq 0 \rightarrow$

$wp((x - 1, z) <_2 (x_0, y_0))$

, assume  $x \neq 0$ ; assume  $y \neq 0$ ; assume  $v_1 \geq 0$ ;  $z := v_1$ )

## Computing the weakest precondition

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

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

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

We finally obtain the verification condition

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

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

BubbleSort — program with loops

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

---

```
@pre  $\top$ 
@post  $\top$ 
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 that

- program is partially correct
- function decreases along each basic path.

The relevant basic paths

---

(1)

@L<sub>1</sub> :  $i + 1 \geq 0$

#L<sub>1</sub> :  $(i + 1, i + 1)$

assume  $i > 0$ ;

$j := 0$ ;

#L<sub>2</sub> :  $(i + 1, i - j)$

---

(2),(3)

@L<sub>2</sub> :  $i + 1 \geq 0 \wedge i - j \geq 0$

#L<sub>2</sub> :  $(i + 1, i - j)$

assume  $j < i$ ;

...

$j := j + 1$ ;

#L<sub>2</sub> :  $(i + 1, i - j)$

---

---

(4)

---

@L<sub>2</sub> :  $i + 1 \geq 0 \wedge i - j \geq 0$

#L<sub>2</sub> :  $(i + 1, i - j)$

assume  $j \geq i$ ;

$i := i - 1$ ;

#L<sub>1</sub> :  $(i + 1, i + 1)$

---

## 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.

## Specification and verification of sequential programs

- Programming language pi and the PiVC verifier
- Program specification
  - Program annotations as assertions
  - Including function preconditions, postconditions, loop invariants, ...
- Partial correctness
  - $@pre + \text{termination} \Rightarrow @post$
  - Notion of weakest preconditions and verification conditions
- Total correctness
  - Additionally guarantees function termination
  - Notion of well-founded relations and ranking functions