5. Program Correctness: Mechanics

Program A: LinearSearch 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 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 ℓ,

execute the body and increment i by 1

as long as $i \leq n$

@ - 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 subarray of sorted array a of integers for specified value e.

sorted: weakly increasing order, i.e.

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

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

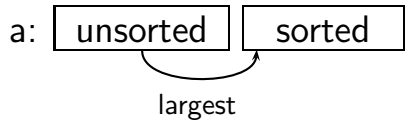
Function specifications

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

Program C: BubbleSort 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;
}
```

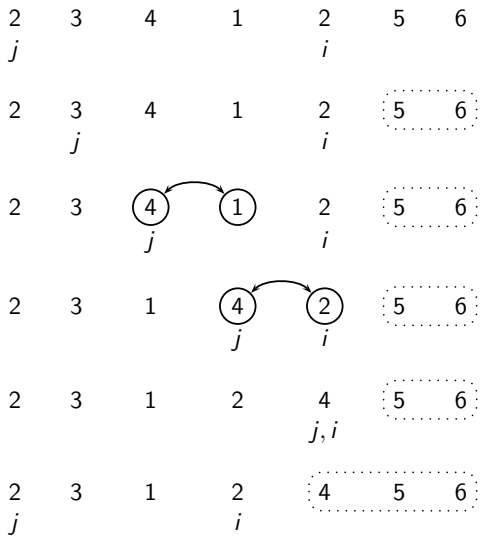

Function BubbleSort sorts integer array a



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.

Sample execution of BubbleSort



Program Annotation

- Function Specifications

  function postcondition (@*post*)

  function precondition (@*pre*)

- Runtime Assertions

  e.g., @ $0 \le j < |a| \ \wedge \ 0 \le j+1 < |a|$

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

- Loop Invariants

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

Program A: LinearSearch with runtime assertions

```
@pre ⊤
@post ⊤
bool LinearSearch(int[] a, int ℓ, int u, int e) {
  for @ ⊤
    (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 ⊤
@post ⊤
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 ⊤
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) {
      @ 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 $\langle body \rangle$ 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
  (⟨init⟩; ⟨cond⟩; ⟨incr⟩) { ⟨body⟩ }
```

$\Rightarrow$

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


Program A: LinearSearch with loop invariants

```
@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 : 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 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 $0 \le \ell \ \wedge \ u < |a|$
$i := \ell$;
@L : $\ell \le i \ \wedge \ \forall j.\, \ell \le j < i \ \rightarrow \ a[j] \neq e$

(2)

@L : $\ell \le i \ \wedge \ \forall j.\, \ell \le j < i \ \rightarrow \ a[j] \neq e$
`assume` $i \le u$;
`assume` $a[i] = e$;
$rv :=$ `true`;
@post $rv \ \leftrightarrow \ \exists j.\, \ell \le j \le u \ \wedge \ a[j] = e$

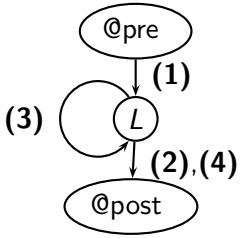
(3)

@L : $\ell \le i \ \wedge \ \forall j.\, \ell \le j < i \ \rightarrow \ a[j] \neq e$
`assume` $i \le u$;
`assume` $a[i] \neq e$;
$i := i + 1$;
@L : $\ell \le i \ \wedge \ \forall j.\, \ell \le j < i \ \rightarrow \ a[j] \neq e$

(4)

@L : $\ell \le i \ \wedge \ \forall j.\, \ell \le j < i \ \rightarrow \ a[j] \neq e$
`assume` $i > u$;
$rv :=$ `false`;
@post $rv \ \leftrightarrow \ \exists j.\, \ell \le j \le u \ \wedge \ a[j] = e$

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`

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

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

```
    for
          @L_2 : [ 1 ≤ i < |a| ∧ 0 ≤ j ≤ i
                 ∧ 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

$$\begin{aligned}&\text{partitioned}(a, \ell_1, u_1, \ell_2, u_2)\\ &\quad \Leftrightarrow\ \forall i, j.\ \ell_1 \le i \le u_1 < \ell_2 \le j \le u_2\ \rightarrow\ a[i] \le a[j]\end{aligned}$$

in $T_{\mathbb{Z}} \cup T_{\mathsf{A}}$.

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

Basic Paths of BubbleSort

(1)

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

(2)

$@L_1 :\ -1 \le i < |a| \ \wedge\ \text{partitioned}(a, 0, i, i + 1, |a| - 1)$
$\quad \wedge\ \text{sorted}(a, i, |a| - 1)$
assume $i > 0$;
$j := 0$;
$@L_2 : \left[\begin{array}{l} 1 \le i < |a| \ \wedge\ 0 \le j \le i \ \wedge\ \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge\ \text{partitioned}(a, 0, j - 1, j, j) \ \wedge\ \text{sorted}(a, i, |a| - 1) \end{array}\right]$

(3)

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

(4)

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

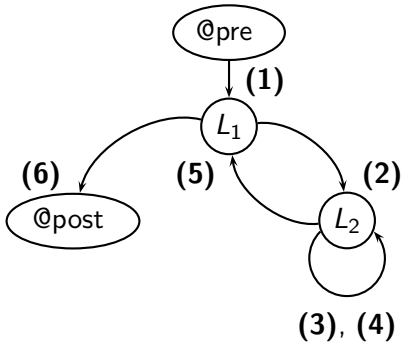
(5)

$@L_2 : \left[\begin{array}{l} 1 \le i < |a| \ \wedge\ 0 \le j \le i \ \wedge\ \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge\ \text{partitioned}(a, 0, j - 1, j, j) \ \wedge\ \text{sorted}(a, i, |a| - 1) \end{array}\right]$
assume $j \ge i$;
$i := i - 1$;
$@L_1 :\ -1 \le i < |a| \ \wedge\ \text{partitioned}(a, 0, i, i + 1, |a| - 1)$
$\quad \wedge\ \text{sorted}(a, i, |a| - 1)$

(6)

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

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

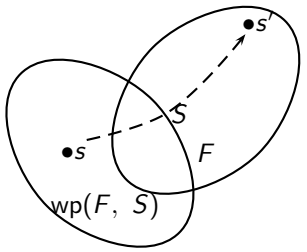
@pre $0 \le \ell\ \wedge\ u < |a|\ \wedge\ \text{sorted}(a, \ell, u)$ $\quad \ldots F[a, \ell, u, e]$
$\vdots$
$@R_1:\ 0 \le m+1\ \wedge\ u < |a|\ \wedge\ \text{sorted}(a, m+1, u)$ $\quad \ldots F[a, m+1, u, e]$
`return` $\text{BinarySearch}(a, m+1, u, e)$
$\vdots$
$@R_2:\ 0 \le \ell\ \wedge\ m-1 < |a|\ \wedge\ \text{sorted}(a, \ell, m-1)$ $\quad \ldots F[a, \ell, m-1, e]$
`return` $\text{BinarySearch}(a, \ell, m-1, e)$

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);
    }
  }
}
```


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



Weakest Precondition $\mathrm{wp}(F,\ S)$

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

Verification Condition of basic path

$@\ F$
$S_1;$
$\ldots$
$S_n;$
$@\ G$

is

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

Also denoted by

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

Example: Basic path

(1)

$@\ F:\ x \geq 0$
$S_1:\ x := x + 1;$
$@\ G:\ x \geq 1$

The VC is

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

That is,

$\mathrm{wp}(G,\ S_1)$
$\Leftrightarrow \mathrm{wp}(x \geq 1,\ x := x + 1)$
$\Leftrightarrow (x \geq 1)\{x \mapsto x + 1\}$
$\Leftrightarrow x + 1 \geq 1$
$\Leftrightarrow x \geq 0$

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)

$@L:\ F:\ \ell \leq i \wedge \forall j.\ \ell \leq j < i \rightarrow a[j] \neq e$
$S_1:\ \texttt{assume}\ i \leq u;$
$S_2:\ \texttt{assume}\ a[i] = e;$
$S_3:\ rv := \texttt{true};$
$@\mathrm{post}\ G:\ rv \leftrightarrow \exists j.\ \ell \leq j \leq u \wedge a[j] = e$

The VC is

$F \rightarrow \mathrm{wp}(G,\ S_1; S_2; S_3)$

That is,

$\mathrm{wp}(G,\ S_1; S_2; S_3)$
$\Leftrightarrow \mathrm{wp}(\mathrm{wp}(rv \leftrightarrow \exists j.\ \ell \leq j \leq u \wedge a[j] = e,\ rv := \texttt{true}),\ S_1; S_2)$
$\Leftrightarrow \mathrm{wp}(\texttt{true} \leftrightarrow \exists j.\ \ell \leq j \leq u \wedge a[j] = e,\ S_1; S_2)$
$\Leftrightarrow \mathrm{wp}(\exists j.\ \ell \leq j \leq u \wedge a[j] = e,\ S_1; S_2)$
$\Leftrightarrow \mathrm{wp}(\mathrm{wp}(\exists j.\ \ell \leq j \leq u \wedge a[j] = e,\ \texttt{assume}\ a[i] = e),\ S_1)$
$\Leftrightarrow \mathrm{wp}(a[i] = e \rightarrow \exists j.\ \ell \leq j \leq u \wedge a[j] = e,\ S_1)$
$\Leftrightarrow \mathrm{wp}(a[i] = e \rightarrow \exists j.\ \ell \leq j \leq u \wedge a[j] = e,\ \texttt{assume}\ i \leq u)$
$\Leftrightarrow i \leq u \rightarrow (a[i] = e \rightarrow \exists j.\ \ell \leq j \leq u \wedge a[j] = e)$

Therefore the VC of path (2)

$$\ell \leq i \wedge (\forall j.\ \ell \leq j < i \rightarrow a[j] \neq e) \rightarrow (i \leq u \rightarrow (a[i] = e \rightarrow \exists j.\ \ell \leq j \leq u \wedge a[j] = e)) \quad (1)$$

or, equivalently,

$$\ell \leq i \wedge (\forall j.\ \ell \leq j < i \rightarrow a[j] \neq e) \wedge i \leq u \wedge a[i] = e \rightarrow \exists j.\ \ell \leq j \leq u \wedge a[j] = e \quad (2)$$

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 $(T_{\mathbb{Z}} \cup T_{\mathrm{A}})$-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

$s_0, s_1, s_2, \ldots$

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 $s_0, s_1, s_2, \ldots$ 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

Total Correctness = Partial Correctness + 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-tupules of natural numbers with the lexicographic extension $<_n$
- Find function δ (ranking function)
  mapping
  program states $\rightarrow$ S
  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.

Choosing well-founded relation and ranking function

Example: Ackermann function — recursive calls

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

```
@pre x ≥ 0 ∧ y ≥ 0
@post rv ≥ 0
↓ (x, y)                    ...ranking function δ : (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 $x \geq 0 \ \wedge \ y \geq 0$
$\downarrow (x, y)$
assume $x \neq 0$;
assume $y = 0$;
$\downarrow (x - 1, 1)$

(2)

@pre $x \geq 0 \ \wedge \ y \geq 0$
$\downarrow (x, y)$
assume $x \neq 0$;
assume $y \neq 0$;
$\downarrow (x, y - 1)$

(3)

@pre $x \geq 0 \ \wedge \ y \geq 0$
$\downarrow (x, y)$
assume $x \neq 0$;
assume $y \neq 0$;
assume $v_1 \geq 0$;
$z := v_1$;
$\downarrow (x - 1, z)$

Showing decrease of ranking function

For basic path with ranking function

@ F
$\downarrow \delta[\overline{x}]$
S_1;
$\vdots$
S_k;
$\downarrow \kappa[\overline{x}]$

We must prove that
the value of κ after executing $S_1; \cdots ; S_n$
is less than
the value of δ before executing the statements
Thus, we show the verification condition

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

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

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

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 ⊤
@post ⊤
int[] BubbleSort(int[] a_0) {
  int[] a := a_0;
  for
    @L_1 : i + 1 ≥ 0
    ↓ (i + 1, i + 1)            ... ranking function δ_1
    (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
- 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.