Decision Procedures

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

Summer 2013

Program Correctness

Road Map

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

The programming language pi

- 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

Program 1: LinearSearch

```
@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 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)

UNI FREIBURG

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

Basic Paths: Loops

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

Basic Paths: Loops

A basic path:

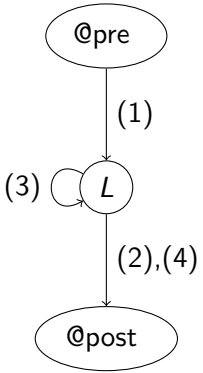
- begins at the function pre condition or a loop invariant,
- ends at an assertion, e.g., the loop invariant or the function post,
- does not contain the loop invariant inside the sequence,
- conditional branches are replaced by assume statements.

Assume statement c

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

Example: Basic Paths of LinearSearch

Visualization of basic paths of LinearSearch



Example: Basic Paths of LinearSearch

(1)

$@\text{pre } 0 \leq \ell \wedge u < |a|$
$i := \ell;$
$@L : \ \ell \leq i \wedge \forall j.\, \ell \leq j < i \rightarrow a[j] \neq e$

(2)

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

Example: Basic Paths of LinearSearch

(3)

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

(4)

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

Proving Partial Correctness

UNI FREIBURG

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 $\text{wp}(F,\ S)$

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

Proving Partial Correctness

UNI FREIBURG

Computing weakest preconditions

- $\mathrm{wp}(F,\ \texttt{assume}\ c) \Leftrightarrow c \rightarrow F$
- $\mathrm{wp}(F[v],\ v := e) \Leftrightarrow F[e]$ ("substitute v with 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

$$
\begin{array}{l}
@\ F \\
S_1; \\
\ldots \\
S_n; \\
@\ G
\end{array}
$$

is

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

Proving Partial Correctness

UNI FREIBURG

Proving partial correctness for programs with loops

- Input: Annotated program
- Produce all basic paths $P = \{p_1, \ldots, p_n\}$
- 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.

VC of 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,

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

Therefore the VC of path (1)

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

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

Program 1: VC of 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: $F \rightarrow \text{wp}(G,\ S_1; S_2; S_3)$

That is,

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

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

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

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

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

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

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

Program 1: VC of basic path (2) of LinearSearch

Therefore the VC of path (2)

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

or, equivalently,

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

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 $(T_{\mathbb{Z}} \cup T_{\mathsf{A}})$-valid.

Tool Demo: PiVC

UNI FREIBURG

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

Example 2: BinarySearch

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 $u < |a|$

Program 2: BinarySearch

```
@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);
  }
}
```

Example: Binary Search 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) {
      @pre  0 ≤ m + 1 ∧ u < |a| ∧ sorted(a, m + 1, u);
      bool tmp := BinarySearch(a, m + 1, u, e);
      @post  tmp ↔ ∃i. m + 1 ≤ i ≤ u ∧ a[i] = e; return tmp;
    } else {
      @pre  0 ≤ ℓ ∧ m − 1 < |a| ∧ sorted(a, ℓ, m − 1);
      bool tmp := BinarySearch(a, ℓ, m − 1, e);
      @post  tmp ↔ ∃i. ℓ ≤ i ≤ m − 1 ∧ a[i] = e;
      return tmp;
    }
  }
}
```

Program 3: BubbleSort

```
@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;
}
```

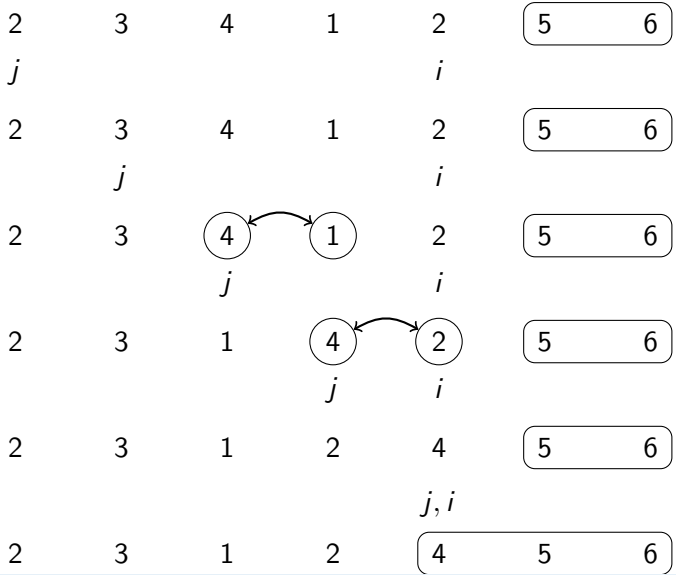
Example 3: BubbleSort

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



BubbleSort with runtime assertions

```
@pre ⊤
@post ⊤
int[] BubbleSort(int[] a_0) {
  int[] a := a_0;
  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;
}
```

BubbleSort with loop invariants

$$
\begin{array}{l}
\texttt{@pre } \top \\
\texttt{@post sorted}(rv, 0, |rv| - 1) \\
\texttt{int[] BubbleSort(int[] } a_0\texttt{)} \ \{ \\
\quad \texttt{int[] } a := a_0; \\
\quad \texttt{for} \\
\qquad @L_1 : \begin{bmatrix} -1 \leq i < |a| \\ \wedge \text{partitioned}(a, 0, i, i+1, |a| - 1) \\ \wedge \text{sorted}(a, i, |a| - 1) \end{bmatrix} \\
\qquad (\texttt{int } i := |a| - 1;\ i > 0;\ i := i - 1)\ \{
\end{array}
$$

$$
\begin{array}{l}
\quad \texttt{for} \\
\qquad @L_2 : \begin{bmatrix} 1 \leq i < |a| \wedge 0 \leq j \leq i \\ \wedge \text{partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge \text{partitioned}(a, 0, j-1, j, j) \\ \wedge \text{sorted}(a, i, |a|-1) \end{bmatrix} \\
\qquad (\texttt{int}\ j := 0;\ j < i;\ j := j+1)\ \{ \\
\qquad \texttt{if}\ (a[j] > a[j+1])\ \{ \\
\qquad\quad \texttt{int}\ t := a[j]; \\
\qquad\quad a[j] := a[j+1]; \\
\qquad\quad a[j+1] := t; \\
\qquad \} \\
\quad \} \\
\quad \texttt{return}\ a; \\
\}
\end{array}
$$

Partition

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

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

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

Basic Paths of BubbleSort

(1)

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

(2)

$@L_1 : \; -1 \leq i < |a| \wedge \text{partitioned}(a, 0, i, i+1, |a|-1)$
$\quad \wedge \text{sorted}(a, i, |a|-1)$
`assume` $i > 0;$
$j := 0;$
$@L_2 : \begin{bmatrix} 1 \leq i < |a| \wedge 0 \leq j \leq i \wedge \text{partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge \text{partitioned}(a, 0, j-1, j, j) \wedge \text{sorted}(a, i, |a|-1) \end{bmatrix}$

(3)

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

(4)

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

$\texttt{assume } j < i;$

$\texttt{assume } a[j] \leq a[j+1];$

$j := j + 1;$

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

(5)

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

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

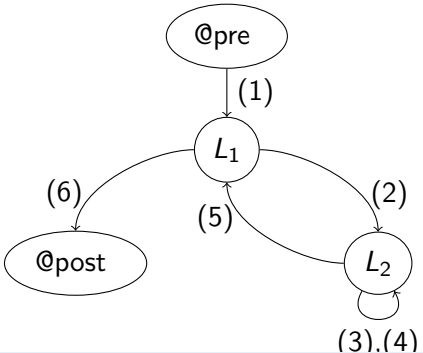
$i := i - 1;$

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

(6)

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

Visualization of basic paths of BubbleSort



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)

Total Correctness

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 termintation. Therefore, we need the notion of well-founded relations and ranking functions.

Well-founded relation

UNI FREIBURG

Definition

For a set S, a binary relation $\prec$ is a well-founded relation iff there is no infinite sequence $s_1, s_2, s_3 \ldots$ of elements of S such that $s_1 \succ s_2 \succ s_3 \succ \cdots$, 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 $\mathbb{Q}$.
$1 > \frac{1}{2} > \frac{1}{3} > \frac{1}{4} > \cdots$

Proving function termination

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

  Such a function δ is called a ranking function.

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

Proving function termination: Example

UNI FREIBURG

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) ↦ (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);
  }
}
```

Proving function termination: Example

To prove function termination:

- Show δ : (x, y) maps into $\mathbb{N}^2$, i.e.,
  $x \geq 0$ and $y \geq 0$ are invariants
- Show δ : (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` $y = 0$;
$\#(x - 1, 1)$

Proving function termination: Example

(2)

$@\text{pre}\ x \geq 0 \wedge y \geq 0$
$\#(x, y)$
$\texttt{assume}\ x \neq 0;$
$\texttt{assume}\ y \neq 0;$
$\#(x, y - 1)$

(3)

$@\text{pre}\ x \geq 0 \wedge y \geq 0$
$\#(x, y)$
$\texttt{assume}\ x \neq 0;$
$\texttt{assume}\ y \neq 0;$
$\texttt{assume}\ v_1 \geq 0;$
$z := v_1;$
$\#(x - 1, z)$

Proving function termination: Verification Condition

Showing decrease of ranking function

Basic path with ranking function:

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

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

Proving function termination: Verification Condition

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:
$x \geq 0 \wedge y \geq 0 \rightarrow$
$\mathrm{wp}((x - 1, z) <_2 (x_0, y_0)$
, `assume` $x \neq 0$; `assume` $y \neq 0$; `assume` $v_1 \geq 0$; $z := v_1$)

Proving function termination: Verification Condition

Computing the weakest precondition

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

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

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

We finally obtain the verification condition

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

Proving function termination: Verification Condition

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

Proving function termination: Example

BubbleSort — program with 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 that

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

The relevant basic paths

(1)

$@L_1 : \ i + 1 \geq 0$
$\#L_1 : \ (i + 1, i + 1)$
`assume` $i > 0$;
$j := 0$;
$\#L_2 : \ (i + 1, i - j)$

(2),(3)

$@L_2 : \ i + 1 \geq 0 \wedge i - j \geq 0$
$\#L_2 : \ (i + 1, i - j)$
`assume` $j < i$;
$\ldots$
$j := j + 1$;
$\#L_2 : \ (i + 1, i - j)$

(4)

$@L_2 : \ i + 1 \geq 0 \wedge i - j \geq 0$
$\#L_2 : \ (i + 1, i - j)$
`assume` $j \geq i$;
$i := i - 1$;
$\#L_1 : \ (i + 1, i + 1)$

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.

Summary

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 + termination $\Rightarrow$ @post
  - Notion of weakest preconditions and verification conditions
- Total correctness
  - Additionally guarantees function termination
  - Notion of well-founded relations and ranking functions