Softwaretechnik

Program verification

Software Engineering
Albert-Ludwigs-University Freiburg

June 30, 2011

Road Map

- Program verification
- Automatic program verification
  - Programs with loops
  - Programs with recursive function calls

Road Map

- Automatic program verification
    - Programs with loops
    - Programs with recursive function calls

Partial Correctness vs. Total Correctness

Two forms of properties.

Partial Correctness

- For a given program p: if p terminates for given input I, then p's output satisfies some relation with I.

Total Correctness

- Partial correctness of p + termination

We focus on proving partial correctness.

Proving Program Correctness: General Approach

Program annotation

- Annotation @F at program location L asserts that formula F is true whenever program control reaches L
- Special annotation: function specification
  - Precondition = specifies what should be true upon entering
  - Postcondition = specifies what must hold after executing

Proving Program Correctness

- Input: Program with annotations
- Translate input to first order formula f
- Validity of f implies program correctness

Outline

- Proving partial correctness
  - Programs with loops
  - Programs with recursive function calls

Outline

- Proving partial correctness
  - Programs with loops

Proving Partial Correctness

Recall

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)
- Remark: Checking validity of formula requires special algorithms ($\rightsquigarrow$ lecture on Decision Procedures)

Programs with Loops

Loop invariants

- Each loop has attendant 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.

Basic Path

@ $\leftarrow$ precondition or loop invariant

finite sequence of instructions
(with no loop invariants)

@ $\leftarrow$ loop invariant, assertion, or postcondition

Basic Paths: Loops

A basic path:

- begins at the function pre condition or a loop invariant,
- ends at 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: 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;
}
```

Example: Basic Paths of LinearSearch

(1)

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

(2)

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

Example: Basic Paths of LinearSearch

(3)

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

`assume` $i \le u$;

`assume` $a[i] \ne e$;

$i := i + 1$;

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

(4)

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

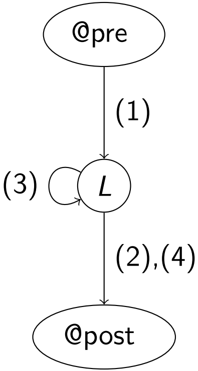
`assume` $i > u$;

$rv :=$ `false`;

$@\mathrm{post}\ rv\ \leftrightarrow\ \exists j.\ \ell \le j \le u\ \wedge\ a[j] = e$

Example: Basic Paths of LinearSearch

Visualization of basic paths of LinearSearch



Proving Partial Correctness

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

Computing weakest preconditions

- Assumption: What must hold before statement `assume` c is executed to ensure that F holds afterward?

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

- Assignment: What must hold before statement $v := e$ is executed to ensure that $F[v]$ holds afterward?

$$\text{wp}(F[v],\ v := e)\ \Leftrightarrow\ F[e]$$

("substitute v with e")

- For sequence of statements $S_1; \ldots; S_n$,
$\text{wp}(F,\ S_1; \ldots; S_n)\ \Leftrightarrow\ \text{wp}(\text{wp}(F,\ S_n),\ S_1; \ldots; S_{n-1})$

Verification Condition

Verification Condition of basic path

```
@ F
S1;
...
Sn;
@ G
```

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

is defined as

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

This verification condition is often denoted by the Hoare triple

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

Proving Partial Correctness

Summary

- 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.

Example 1: 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 valid.

Example 2: VC of basic path (2) of LinearSearch

(2)

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

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

That is,

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

Outline

- Proving partial correctness
  - Programs with loops
  - Programs with recursive function calls

Outline

- Proving partial correctness
  - Programs with recursive function calls

Basic Paths: Recursive Function Calls

- Loops produce unbounded number of paths
  loop invariants cut loops to produce
  finite number of basic paths
- Recursive calls produce unbounded number of paths
  function specifications cut function calls

Function specification

- Add function summary for each function call
- Replace pre- and postcondition with parameters of recursive call

Example: BinarySearch

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

sorted: weakly increasing order, i.e.

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

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 $u < |a|$

Example: 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;
    }
```

Summary

Specification and verification of sequential programs

- Program specification
  - Assertions
  - Including function preconditions, postconditions, loop invariants, ...
- Partial correctness
  - `@pre` + termination ⇒ `@post`
  - Notion of weakest preconditions and verification conditions

Not discussed (so far): Total correctness

- Additionally guarantees function termination