Softwaretechnik

Program verification

Albert-Ludwigs-Universität Freiburg

June 20, 2013
Road Map

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

Automatic Verification

- Function + annotation is transformed to finite set of FOL formulae, the **verification conditions** (VCs)
- If all VCs are valid, then the function obeys its specification (partially correct)
Programs with Loops

Loop invariants

- Each loop must be annotated with a loop invariant, $\mathcal{L}$
- **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

@ ← precondition or loop invariant

finite sequence of instructions
(no loop invariants)

@ ← loop invariant, assertion, or postcondition
Basic Paths: Conditionals

Basic paths split at conditionals

Replace each path $BP[\text{if } B \text{ then } S_1 \text{ else } S_2]$ by two paths
- $BP[\text{assume } B; S_1]$
- $BP[\text{assume } \neg B; S_2]$

Semantics of “assume $B$”

Execution ends unless $B$ holds
Example: LinearSearch

@pre 0 ≤ ℓ ∧ u < a.length
@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

\( \text{(1)} \)
\[
\begin{align*}
\text{@pre } & 0 \leq \ell \land u < a.length \\
& i := \ell; \\
\text{@L} : & \ell \leq i \land \forall j. \ell \leq j < i \rightarrow a[j] \neq e
\end{align*}
\]

\( \text{(2)} \)
\[
\begin{align*}
\text{@L} : & \ell \leq i \land \forall j. \ell \leq j < i \rightarrow a[j] \neq e \\
& \text{assume } i \leq u; \\
& \text{assume } a[i] = e; \\
& rv := true; \\
\text{@post } & rv \iff \exists j. \ell \leq j \leq u \land a[j] = e
\end{align*}
\]
Example: Basic Paths of LinearSearch

(3) \[ \forall L : \ell \leq i \land \forall j. \ell \leq j < i \rightarrow a[j] \neq e \]
assume \( i \leq u \);
assume \( a[i] \neq e \);
\( i := i + 1 ; \)
\[ \forall L : \ell \leq i \land \forall j. \ell \leq j < i \rightarrow a[j] \neq e \]

(4) \[ \forall L : \ell \leq i \land \forall j. \ell \leq j < i \rightarrow a[j] \neq e \]
assume \( i > u \);
\( rv := false \);
\[ \forall post \ rv \leftrightarrow \exists j. \ell \leq j \leq u \land a[j] = e \]
Example: Basic Paths of LinearSearch

Visualization of basic paths of LinearSearch

@pre

(1)

(3) L

(2),(4)

@post
Proving Partial Correctness

Goal

- Prove that annotated function \( f \) agrees with annotations
- Transform \( f \) 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 \( \sigma \) such that \( \sigma \in \text{wp}(F, S) \): successor state \( S[S]\sigma \in F \).
Weakest preconditions for each statement

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

  \[ \text{wp}(F, \text{assume } B) \iff B \rightarrow F \]

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

  \[ \text{wp}(F[x], x := e) \iff F[e] \]

  (“substitute `x` with `e`”)

- **Sequence of statements** `S_1; \ldots; S_n` (`n > 1`),

  \[ \text{wp}(F, S_1; \ldots; S_n) \iff \text{wp}(\text{wp}(F, S_n), S_1; \ldots; S_{n-1}) \]
Verification condition of basic path

@ F
S_1;
...
S_n;
@ G

is defined as

F \rightarrow \text{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

Approach

- Input: Annotated program
- Compute the set $P$ of all basic paths (finite)
- 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

\[ (1) \]
\[
\begin{align*}
\@ F : & \quad x \geq 0 \\
S_1 : & \quad x := x + 1; \\
\@ G : & \quad x \geq 1
\end{align*}
\]

The VC is
\[ F \rightarrow \text{wp}(G, S_1) \]

That is,
\[ \text{wp}(G, S_1) \]
\[ \iff \text{wp}(x \geq 1, x := x + 1) \]
\[ \iff (x \geq 1)\{x \Rightarrow x + 1\} \]
\[ \iff x + 1 \geq 1 \]
\[ \iff x \geq 0 \]

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

[@L : ]  \( F : \ell \leq i \land \forall j. \ell \leq j < i \rightarrow a[j] \neq e \)

\( S_1 \) : assume \( i \leq u \);

\( S_2 \) : assume \( a[i] = e \);

\( S_3 \) : \( rv := true \);

@post \( G : \) \( rv \leftrightarrow \exists j. \ell \leq j \leq u \land a[j] = e \)

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

\( wp(G, S_1; S_2; S_3) \)

\( \leftrightarrow wp(wp(rv \leftrightarrow \exists j. \ell \leq j \leq u \land a[j] = e, rv := true), S_1; S_2) \)

\( \leftrightarrow wp(true \leftrightarrow \exists j. \ell \leq j \leq u \land a[j] = e, S_1; S_2) \)

\( \leftrightarrow wp(\exists j. \ell \leq j \leq u \land a[j] = e, S_1; S_2) \)

\( \leftrightarrow wp(wp(\exists j. \ell \leq j \leq u \land a[j] = e, assume a[i] = e), S_1) \)

\( \leftrightarrow wp(a[i] = e \rightarrow \exists j. \ell \leq j \leq u \land a[j] = e, S_1) \)

\( \leftrightarrow wp(a[i] = e \rightarrow \exists j. \ell \leq j \leq u \land a[j] = e, assume i \leq u) \)

\( \leftrightarrow i \leq u \rightarrow (a[i] = e \rightarrow \exists j. \ell \leq j \leq u \land a[j] = e) \)
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
- Instantiate 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.

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

**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 $u < a.length$
Example: BinarySearch

@pre $0 \leq \ell \land u < a.length \land \text{sorted}(a, \ell, u)$
@post $\text{rv} \leftrightarrow \exists i. \ell \leq i \leq u \land a[i] = e$

```cpp
bool BinarySearch(int[] a, int \ell, int u, int e) {
    if ($\ell > u$) return false;
    else {
        int m := (\ell + 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, \ell, m - 1, e);
    }
}
```
Example: Binary Search with Function Call Assertions

@pre $0 \leq \ell \land u < a.length \land \text{sorted}(a, \ell, u)$
@post $rv \leftrightarrow \exists i. \ell \leq i \leq u \land a[i] = e$

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

Softwaretechnik  June 20, 2013  23 / 24
## Summary

### Automatic verification of sequential programs

- **Goal:** Proof of partial correctness
- **Program specification**
  - Pre- and postconditions
  - Loop invariants
- **Tools**
  - Basic paths
  - Weakest precondition
  - Verification conditions
  - Function summaries