Verification

Lecture 19

Bernd Finkbeiner
Peter Faymonville
Michael Gerke

UNIVERSITÄT
DES
SAARLANDES

Deductive verification

Verification of infinite-state programs
based on automatic decision procedures

Textbooks:

Bradley/Manna, *The Calculus of Computation*

- sequential programs
- first-order logic
- inductive assertions, ranking functions
- decision procedures

Manna/Pnueli: *Temporal Verification of Reactive Systems*

- reactive programs
- temporal logic
- verification rules and diagrams

Deductive Verification: Basic mechanics

Example: Linear search

```
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 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;
}
```

- Precondition: Function guaranteed to behave correctly if $0 \leq l$ and $u < |a|$
- Postcondition: Function returns true iff a contains value e in the range $[l, u]$.

Example: Binary search

```
@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);
  }
}
```

sorted: weakly increasing order, i.e.,

$$sorted(a, l, u) \Leftrightarrow \forall i, j.\ l \leq i \leq j \leq u \rightarrow a[i] \leq a[j]$$

Example: Bubble sort

```
@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;
```

Loop invariants

```
while
@ F
(⟨condition⟩) {
⟨body⟩
}
```

```
for
@ F
(⟨init⟩; ⟨cond⟩; ⟨incr⟩) {
⟨body⟩
}
```

- ⟨body⟩ is applied as long as ⟨condition⟩ holds
- Assertion *F* must hold in every iteration
- *F* is evaluated before ⟨condition⟩
  $\Rightarrow$ must hold even on final iteration
  when ⟨condition⟩ is false

Example: linear search

```
@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;
}
```

Assertions

`@` *F*

- Assertions can be added anywhere
- Assertions are “formal comments”
- Special class: runtime assertions
  refer to runtime errors such as
  - division by 0
  - array access out of bounds

Example: Binary search

```
@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 {
    @ 2 ≠ 0
    int m := (l + u) div 2;
    @ 0 ≤ m < |a|
    if (a[m] = e) return true;
    @ 0 ≤ m < |a|
    else if (a[m] < e) return BinarySearch(a, m + 1, u, e);
    else return BinarySearch(a, l, m − 1, e);
  }
}
```

Partial Correctness

A function is partially correct if

- when the function's precondition is satisfied on entry,
- its postcondition is satisfied when the function returns (if it ever does).

Inductive assertion method

- Each function and its annotion are reduced to a finite set of verification conditions (VCs)
- VCs are formulas of first-order logic
- If all VCs are valid, then the function is partially correct.

Basic paths

We break the program into basic paths:

@ precondition or loop invariant sequence of instructions (no loop invariants) @ loop invariant, assertion, or postcondition

- loops and recursive calls produce an unbounded number of paths
- loop invariants "cut loops" into a finite set of basic paths
- function specifications "cut function calls" into a finite set of basic paths

Basic paths: Linear search

```
@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;
}
```

(1)	@pre $0 \le l \wedge u < \lvert a \rvert$ $i := l$ @ $L : l \le i \wedge (\forall j.\ l \le j < i \rightarrow a[j] \ne e)$	**(3)**	@ $L : l \le i \wedge (\forall j.\ l \le j < i \rightarrow a[j] \ne e)$ assume $i \le u$; assume $a[i] \ne e$; $i := i + 1$ @ $L : l \le i \wedge (\forall j.\ l \le j < i \rightarrow a[j] \ne e)$
(2)	@ $L : l \le i \wedge (\forall j.\ l \le j < i \rightarrow a[j] \ne e)$ assume $i \le u$; assume $a[i] = e$; $rv :=$ true @post $rv \leftrightarrow \exists i.\ l \le i \le u \wedge a[i] = e$	**(4)**	@ $L : l \le i \wedge (\forall j.\ l \le j < i \rightarrow a[j] \ne e)$ assume $i > u$; $rv :=$ false @post $rv \leftrightarrow \exists i.\ l \le i \le u \wedge a[i] = e$

Function call assertions: Binary search

```
@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);
  }
}
```

Function call assertion R_1 results from precondition

$F[a, l, u, e] : 0 \leq l \land u < |a| \land sorted(a, l, u)$ as $R_1 = F[a, m+1, u, e]$

Function summary: Binary search

(3)	`@pre` $0 \leq l \land u < \lvert a \rvert \land sorted(a, l, u)$ `assume` $l \leq u$; $m := (l + u)$ `div` 2; `assume` $a[m] \neq e$; `assume` $a[m] < e$; $@R_1 : 0 \leq m + 1 \land u < \lvert a \rvert \land sorted(a, m + 1, u)$
(4)	`@pre` $0 \leq l \land u < \lvert a \rvert \land sorted(a, l, u)$ `assume` $l \leq u$; $m := (l + u)$ `div` 2; `assume` $a[m] \neq e$; `assume` $a[m] < e$; `assume` $v_1 \leftrightarrow \exists i.\ m + 1 \leq i \leq u \land a[i] = e$; $rv := v_1$; `@post` $rv \leftrightarrow \exists i.\ l \leq i \leq u \land a[i] = e$

Function summary results from function postcondition `@post`
$G[a, l, u, e, rv] : rv \leftrightarrow \exists i.\ l \leq i \leq u \land a[i] = e$

- Introduce a fresh variable (v_1)
- Assume that the function postcondition holds
  `assume` $G[a, m + 1, u, e, v_1]$.

Program states

- Program counter *pc* holds current location of control
- State *s* is an assignment of values (of proper type) to all variables

  Example:

  $$s : \left\{ \begin{array}{l} pc \mapsto L, \\ a \mapsto [0; 1; 2], i \mapsto 3 \end{array} \right\}$$

  is a state of LinearSearch

- Reachable state *s* is a state that can be reached during some computation

  Example:

  $$s : \left\{ \begin{array}{l} pc \mapsto L, \\ a \mapsto [0; 1; 2], i \mapsto 2 \end{array} \right\}$$

  is a reachable state of LinearSearch

Weakest precondition

- A predicate transformer is a function

$$p : \text{FOL} \times \text{stmnts} \rightarrow \text{FOL}$$

  that maps a formula of first-order logic and a statement to another formula of first-order logic.
- The weakest precondition $wp(F, S)$ is a predicate transformer such that for every state s with

$$s \models wp(F, S),$$

  if statement S is executed on s to produce s', then

$$s' \models F.$$

Verification condition

- $wp(F, \texttt{assume}\ c) \Leftrightarrow c \rightarrow F$
- $wp(F[v], v := e) \Leftrightarrow F[e]$
- $wp(F, S_1; S_2; \ldots; S_{n-1}; S_n) \Leftrightarrow wp(wp(F, S_n), S_1; S_2; \ldots S_{n-1})$

The verification condition of basic path

```
@ F
S_1;
...
S_n;
@ G
```

is

$$F \rightarrow wp(G, S_1; \ldots; S_n).$$

Traditionally, this verification condition is denoted by the
Hoare triple $\{F\}\ S_1; \ldots; S_n\ \{G\}$.

Example: Linear search

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

The VC of basic path **(2)** is

$$F \rightarrow wp(G, S_1; S_2; S_3).$$

We compute

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

Example: Linear search

$$
\begin{array}{ll}
\textbf{(2)} & @L : F : l \leq i \land (\forall j.\ \ l \leq j < i \rightarrow a[j] \neq e) \\
 & S_1 : \texttt{assume}\ i \leq u \\
 & S_2 : \texttt{assume}\ a[i] = e \\
 & S_3 : rv := \texttt{true} \\
 & @\texttt{post}\ \ G : rv \leftrightarrow \exists i.\ \ l \leq i \leq u \land a[i] = e
\end{array}
$$

The VC of basic path **(2)** is

$$F \rightarrow wp(G, S_1; S_2; S_3).$$

We compute

$$wp(G, S_1; S_2; S_3) \Leftrightarrow i \leq u \rightarrow (a[i] = e \rightarrow \exists i.\, l \leq i \leq u \land a[i] = e)$$

Hence, the VC is

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

which is valid.

Computations

- Consider program P with function f, function precondition F_{pre} 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 $s_0 \models F_{pre}$, and
  - for each i, s_{i+1} is the result of executing the instruction at $s_i[pc]$ on state s_i.

Notation: $s_i[pc]$ = value of pc given by state s_i.

P-invariant and *P*-inductive

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,

$$s_i[pc] = L \Rightarrow s_i \models F$$

Annotations of P are *P*-invariant iff each annotation of P is P-invariant at its location.

Note: this definition is not implementable. Checking if F is P-invariant requires an infinite number of P-computations in general.

P-inductive

Instead, we check if the annotations are P-inductive.

Annotations of P are P-inductive iff all VCs generated from the basic paths of program P are valid.

$$P\text{-inductive} \Rightarrow P\text{-invariant}$$

Theorem (Verification Conditions)

If for every basic path

@ $L_1 : F$ S_1 $\vdots$ S_n @ $L_j : G$

of program P, the verification condition

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

is valid, then the annotatons are P-inductive, and therefore P-invariant.

If there is a P-invariant annotation, then P is partially correct.

Example: Bubble sort

```
@pre ⊤
@post sorted(rv, 0, |rv| − 1)
int[] BubbleSort(int[] a₀) {
int[] a := a₀;
for @ L₁
  (int i := |a| − 1; i > 0; i := i − 1) {
  for @ L₂
    (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;
```