Reasoning about Programs I

Mădălina Erașcu

West University of Timișoara and Institute e-Austria Timișoara
bvd. V. Parvan 4, Timișsoara, Romania

madalina.erascu@e-uvt.ro

Based on: (1) CSE 507: Computer-Aided Reasoning for Software by Emina Torlak https://courses.cs.washington.edu/courses/cse507/14au/index.html, (2) Aaron R. Bradley, Zohar Manna: The calculus of computation - decision procedures with applications to verification. Springer 2007, pp. I-XV, 1-366

Outline

Outline

Preliminary Concepts

Aim: Applying FOL to a real problem: *specifying and proving properties of programs.*

There are three foundational methods that underly all verification and program analysis techniques:

1. specification is the precise statement of properties that a program should exhibit
   - The language of FOL is precise.
   - Task: develop a scheme for embedding FOL statements into program text as program annotations.
   - We are interested on 2 types of properties of programs:
2. inductive assertion method is based on mathematical induction:
   - To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution, assume as the inductive hypothesis that F currently holds (at some point during the execution), and prove as the inductive step that F holds after one more step of the program.
   - Challenge: discover additional annotations to make the induction go through.
3. ranking function method for proving total correctness properties:
   - Prove partial correctness using the inductive assertion method.
   - Prove that loops and recursion calls terminate/halt by using ranking function method, i.e. for each loop and recursive call in the program come up with a ranking function that maps the program variables to a well-founded domain, then one proves that whenever program control moves from one ranking function to the next, the value decreases according to a well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts and its output satisfies some relation with its input.

Preliminary Concepts

Aim: Applying FOL to a real problem: *specifying and proving properties of programs.*

There are three foundational methods that underly all verification and program analysis techniques:

1. specification is the precise statement of properties that a program should exhibit
   - The language of FOL is precise.
   - Task: develop a scheme for embedding FOL statements into program text as program annotations.
   - We are interested on 2 types of properties of programs:
     - Partial correctness properties (safety properties) assert that certain states (e.g. error states) cannot occur during the execution of a program.
       Formulation: *If a program halts, then its output satisfies some relation with its input.*
     - Total correctness properties (liveness properties) assert that certain states are eventually reached during program execution.
2. inductive assertion method is based on mathematical induction:
   - To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution; assume as the inductive hypothesis that F currently holds (at some point during the execution); and prove as the inductive step that F holds after one more step of the program.
   - Challenge: discover additional annotations to make the induction go through.
3. ranking function method for proving total correctness properties:
   - Prove partial correctness using the inductive assertion method
   - Prove that loops and recursion calls terminate/halt by using ranking function method i.e. for each loop and recursive call in the program come up with a ranking function that maps the program variables to a well-founded domain; then one proves that whenever program control moves from one ranking function to the next, the value decreases according to a well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts and its output satisfies some relation with its input.

Preliminary Concepts

Aim: Applying FOL to a real problem: *specifying and proving properties of programs.*

There are three foundational methods that underly all verification and program analysis techniques:

1. specification is the precise statement of properties that a program should exhibit
   - The language of FOL is precise.
   - Task: develop a scheme for embedding FOL statements into program text as program annotations.
   - We are interested on 2 types of properties of programs:
     - Partial correctness properties (safety properties) assert that certain states (e.g error states) cannot occur during the execution of a program.
       Formulation: *If a program halts, then its output satisfies some relation with its input.*
     - Total correctness properties (liveness properties) assert that certain states are eventually reached during program execution.
2. inductive assertion method is based on mathematical induction:
   - To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution; assume as the inductive hypothesis that F currently holds (at some point during the execution); and prove as the inductive step that F holds after one more step of the program.
   - Challenge: discover additional annotations to make the induction go through.
3. ranking function method for proving total correctness properties:
   - Prove partial correctness using the inductive assertion method
   - Prove that loops and recursion calls terminate/halt by using ranking function method i.e. for each loop and recursive call in the program come up with a ranking function that maps the program variables to a well-founded domain; then one proves that whenever program control moves from one ranking function to the next, the value decreases according to a well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts and its output satisfies some relation with its input.

Preliminary Concepts

Aim: Applying FOL to a real problem: *specifying and proving properties of programs.*

There are three foundational methods that underly all verification and program analysis techniques:

1. specification is the precise statement of properties that a program should exhibit
   - The language of FOL is precise.
   - Task: develop a scheme for embedding FOL statements into program text as program annotations.
   - We are interested on 2 types of properties of programs:
     - Partial correctness properties (safety properties) assert that certain states (e.g error states) cannot occur during the execution of a program.
       Formulation: *If a program halts, then its output satisfies some relation with its input.*
     - Total correctness properties (liveness properties) assert that certain states are eventually reached during program execution.
2. inductive assertion method is based on mathematical induction:
   - To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution; assume as the inductive hypothesis that F currently holds (at some point during the execution); and prove as the inductive step that F holds after one more step of the program.
   - Challenge: discover additional annotations to make the induction go through.
3. ranking function method for proving total correctness properties:
   - Prove partial correctness using the inductive assertion method
   - Prove that loops and recursion calls terminate/halt by using ranking function method i.e. for each loop and recursive call in the program come up with a ranking function that maps the program variables to a well-founded domain; then one proves that whenever program control moves from one ranking function to the next, the value decreases according to a well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts and its output satisfies some relation with its input.

Preliminary Concepts

Aim: Applying FOL to a real problem: *specifying and proving properties of programs.*

There are three foundational methods that underly all verification and program analysis techniques:

1. specification is the precise statement of properties that a program should exhibit
   - The language of FOL is precise.
   - Task: develop a scheme for embedding FOL statements into program text as program annotations.
   - We are interested on 2 types of properties of programs:
     - Partial correctness properties (safety properties) assert that certain states (e.g error states) cannot occur during the execution of a program.
       Formulation: *If a program halts, then its output satisfies some relation with its input.*
     - Total correctness properties (liveness properties) assert that certain states are eventually reached during program execution.
2. inductive assertion method is based on mathematical induction:
   - To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution; assume as the inductive hypothesis that F currently holds (at some point during the execution); and prove as the inductive step that F holds after one more step of the program.
   - Challenge: discover additional annotations to make the induction go through.
3. ranking function method for proving total correctness properties:
   - Prove partial correctness using the inductive assertion method
   - Prove that loops and recursion calls terminate/halt by using ranking function method i.e. for each loop and recursive call in the program come up with a ranking function that maps the program variables to a well-founded domain; then one proves that whenever program control moves from one ranking function to the next, the value decreases according to a well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts and its output satisfies some relation with its input.

Preliminary Concepts

Aim: Applying FOL to a real problem: *specifying and proving properties of programs.*

There are three foundational methods that underly all verification and program analysis techniques:

1. specification is the precise statement of properties that a program should exhibit
   - The language of FOL is precise.
   - Task: develop a scheme for embedding FOL statements into program text as program annotations.
   - We are interested on 2 types of properties of programs:
     - Partial correctness properties (safety properties) assert that certain states (e.g error states) cannot occur during the execution of a program.
       Formulation: *If a program halts, then its output satisfies some relation with its input.*
     - Total correctness properties (liveness properties) assert that certain states are eventually reached during program execution.
2. inductive assertion method is based on mathematical induction:
   - To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution; assume as the inductive hypothesis that F currently holds (at some point during the execution); and prove as the inductive step that F holds after one more step of the program.
   - Challenge: discover additional annotations to make the induction go through.
3. ranking function method for proving total correctness properties:
   - Prove partial correctness using the inductive assertion method
   - Prove that loops and recursion calls terminate/halt by using ranking function method i.e. for each loop and recursive call in the program come up with a ranking function that maps the program variables to a well-founded domain; then one proves that whenever program control moves from one ranking function to the next, the value decreases according to a well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts and its output satisfies some relation with its input.

Preliminary Concepts

Aim: Applying FOL to a real problem: *specifying and proving properties of programs.*

There are three foundational methods that underly all verification and program analysis techniques:

1. specification is the precise statement of properties that a program should exhibit
   - The language of FOL is precise.
   - Task: develop a scheme for embedding FOL statements into program text as program annotations.
   - We are interested on 2 types of properties of programs:
     - Partial correctness properties (safety properties) assert that certain states (e.g error states) cannot occur during the execution of a program.
       Formulation: *If a program halts, then its output satisfies some relation with its input.*
     - Total correctness properties (liveness properties) assert that certain states are eventually reached during program execution.
2. inductive assertion method is based on mathematical induction:
   - To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution; assume as the inductive hypothesis that F currently holds (at some point during the execution); and prove as the inductive step that F holds after one more step of the program.
   - Challenge: discover additional annotations to make the induction go through.
3. ranking function method for proving total correctness properties:
   - Prove partial correctness using the inductive assertion method
   - Prove that loops and recursion calls terminate/halt by using ranking function method i.e. for each loop and recursive call in the program come up with a ranking function that maps the program variables to a well-founded domain; then one proves that whenever program control moves from one ranking function to the next, the value decreases according to a well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts and its output satisfies some relation with its input.

Preliminary Concepts

Aim: Applying FOL to a real problem: *specifying and proving properties of programs.*

There are three foundational methods that underly all verification and program analysis techniques:

1. specification is the precise statement of properties that a program should exhibit
   - The language of FOL is precise.
   - Task: develop a scheme for embedding FOL statements into program text as program annotations.
   - We are interested on 2 types of properties of programs:
     - Partial correctness properties (safety properties) assert that certain states (e.g error states) cannot occur during the execution of a program.
       Formulation: *If a program halts, then its output satisfies some relation with its input.*
     - Total correctness properties (liveness properties) assert that certain states are eventually reached during program execution.
2. inductive assertion method is based on mathematical induction:
   - To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution; assume as the inductive hypothesis that F currently holds (at some point during the execution); and prove as the inductive step that F holds after one more step of the program.
   - Challenge: discover additional annotations to make the induction go through.
3. ranking function method for proving total correctness properties:
   - Prove partial correctness using the inductive assertion method
   - Prove that loops and recursion calls terminate/halt by using ranking function method i.e. for each loop and recursive call in the program come up with a ranking function that maps the program variables to a well-founded domain; then one proves that whenever program control moves from one ranking function to the next, the value decreases according to a well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts and its output satisfies some relation with its input.

Preliminary Concepts

Aim: Applying FOL to a real problem: *specifying and proving properties of programs.*

There are three foundational methods that underly all verification and program analysis techniques:

1. specification is the precise statement of properties that a program should exhibit
   - The language of FOL is precise.
   - Task: develop a scheme for embedding FOL statements into program text as program annotations.
   - We are interested on 2 types of properties of programs:
     - Partial correctness properties (safety properties) assert that certain states (e.g error states) cannot occur during the execution of a program.
       Formulation: *If a program halts, then its output satisfies some relation with its input.*
     - Total correctness properties (liveness properties) assert that certain states are eventually reached during program execution.
2. inductive assertion method is based on mathematical induction:
   - To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution; assume as the inductive hypothesis that F currently holds (at some point during the execution); and prove as the inductive step that F holds after one more step of the program.
   - Challenge: discover additional annotations to make the induction go through.
3. ranking function method for proving total correctness properties:
   - Prove partial correctness using the inductive assertion method
   - Prove that loops and recursion calls terminate/halt by using ranking function method i.e. for each loop and recursive call in the program come up with a ranking function that maps the program variables to a well-founded domain; then one proves that whenever program control moves from one ranking function to the next, the value decreases according to a well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts and its output satisfies some relation with its input.

Preliminary Concepts

Aim: Applying FOL to a real problem: *specifying and proving properties of programs.*

There are three foundational methods that underly all verification and program analysis techniques:

1. specification is the precise statement of properties that a program should exhibit
   - The language of FOL is precise.
   - Task: develop a scheme for embedding FOL statements into program text as program annotations.
   - We are interested on 2 types of properties of programs:
     - Partial correctness properties (safety properties) assert that certain states (e.g error states) cannot occur during the execution of a program.
       Formulation: *If a program halts, then its output satisfies some relation with its input.*
     - Total correctness properties (liveness properties) assert that certain states are eventually reached during program execution.
2. inductive assertion method is based on mathematical induction:
   - To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution; assume as the inductive hypothesis that F currently holds (at some point during the execution); and prove as the inductive step that F holds after one more step of the program.
   - Challenge: discover additional annotations to make the induction go through.
3. ranking function method for proving total correctness properties:
   - Prove partial correctness using the inductive assertion method
   - Prove that loops and recursion calls terminate/halt by using ranking function method i.e. for each loop and recursive call in the program come up with a ranking function that maps the program variables to a well-founded domain; then one proves that whenever program control moves from one ranking function to the next, the value decreases according to a well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts and its output satisfies some relation with its input.

Preliminary Concepts

Aim: Applying FOL to a real problem: *specifying and proving properties of programs.*

There are three foundational methods that underly all verification and program analysis techniques:

1. specification is the precise statement of properties that a program should exhibit
   - The language of FOL is precise.
   - Task: develop a scheme for embedding FOL statements into program text as program annotations.
   - We are interested on 2 types of properties of programs:
     - Partial correctness properties (safety properties) assert that certain states (e.g error states) cannot occur during the execution of a program.
       Formulation: *If a program halts, then its output satisfies some relation with its input.*
     - Total correctness properties (liveness properties) assert that certain states are eventually reached during program execution.
2. inductive assertion method is based on mathematical induction:
   - To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution; assume as the inductive hypothesis that F currently holds (at some point during the execution); and prove as the inductive step that F holds after one more step of the program.
   - Challenge: discover additional annotations to make the induction go through.
3. ranking function method for proving total correctness properties:
   - Prove partial correctness using the inductive assertion method
   - Prove that loops and recursion calls terminate/halt by using ranking function method i.e. for each loop and recursive call in the program come up with a ranking function that maps the program variables to a well-founded domain; then one proves that whenever program control moves from one ranking function to the next, the value decreases according to a well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts and its output satisfies some relation with its input.

Preliminary Concepts

Aim: Applying FOL to a real problem: *specifying and proving properties of programs.*

There are three foundational methods that underly all verification and program analysis techniques:

1. specification is the precise statement of properties that a program should exhibit
   - The language of FOL is precise.
   - Task: develop a scheme for embedding FOL statements into program text as program annotations.
   - We are interested on 2 types of properties of programs:
     - Partial correctness properties (safety properties) assert that certain states (e.g error states) cannot occur during the execution of a program.
       Formulation: *If a program halts, then its output satisfies some relation with its input.*
     - Total correctness properties (liveness properties) assert that certain states are eventually reached during program execution.
2. inductive assertion method is based on mathematical induction:
   - To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution; assume as the inductive hypothesis that F currently holds (at some point during the execution); and prove as the inductive step that F holds after one more step of the program.
   - Challenge: discover additional annotations to make the induction go through.
3. ranking function method for proving total correctness properties:
   - Prove partial correctness using the inductive assertion method
   - Prove that loops and recursion calls terminate/halt by using ranking function method i.e. for each loop and recursive call in the program come up with a ranking function that maps the program variables to a well-founded domain; then one proves that whenever program control moves from one ranking function to the next, the value decreases according to a well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts and its output satisfies some relation with its input.

Preliminary Concepts

Aim: Applying FOL to a real problem: *specifying and proving properties of programs.*

There are three foundational methods that underly all verification and program analysis techniques:

1. specification is the precise statement of properties that a program should exhibit
   - The language of FOL is precise.
   - Task: develop a scheme for embedding FOL statements into program text as program annotations.
   - We are interested on 2 types of properties of programs:
     - Partial correctness properties (safety properties) assert that certain states (e.g error states) cannot occur during the execution of a program.
       Formulation: *If a program halts, then its output satisfies some relation with its input.*
     - Total correctness properties (liveness properties) assert that certain states are eventually reached during program execution.
2. inductive assertion method is based on mathematical induction:
   - To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution; assume as the inductive hypothesis that F currently holds (at some point during the execution); and prove as the inductive step that F holds after one more step of the program.
   - Challenge: discover additional annotations to make the induction go through.
3. ranking function method for proving total correctness properties:
   - Prove partial correctness using the inductive assertion method
   - Prove that loops and recursion calls terminate/halt by using ranking function method i.e. for each loop and recursive call in the program come up with a ranking function that maps the program variables to a well-founded domain; then one proves that whenever program control moves from one ranking function to the next, the value decreases according to a well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts and its output satisfies some relation with its input.

Preliminary Concepts

Aim: Applying FOL to a real problem: *specifying and proving properties of programs.*

There are three foundational methods that underly all verification and program analysis techniques:

1. specification is the precise statement of properties that a program should exhibit
   - The language of FOL is precise.
   - Task: develop a scheme for embedding FOL statements into program text as program annotations.
   - We are interested on 2 types of properties of programs:
     - Partial correctness properties (safety properties) assert that certain states (e.g error states) cannot occur during the execution of a program.
       Formulation: *If a program halts, then its output satisfies some relation with its input.*
     - Total correctness properties (liveness properties) assert that certain states are eventually reached during program execution.
2. inductive assertion method is based on mathematical induction:
   - To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution; assume as the inductive hypothesis that F currently holds (at some point during the execution); and prove as the inductive step that F holds after one more step of the program.
   - Challenge: discover additional annotations to make the induction go through.
3. ranking function method for proving total correctness properties:
   - Prove partial correctness using the inductive assertion method
   - Prove that loops and recursion calls terminate/halt by using ranking function method i.e. for each loop and recursive call in the program come up with a ranking function that maps the program variables to a well-founded domain; then one proves that whenever program control moves from one ranking function to the next, the value decreases according to a well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts and its output satisfies some relation with its input.

Annotations: Function Specification

An annotation is a FOL formula F whose free variables include only the program variables of the function in which the annotation occurs. An annotation F at location L asserts that F is true whenever program control reaches L.

We have: (1) (function) specification, (2) loop invariants, (3) assertions.

Function specification of a function is a pair of annotations.

The function precondition is a formula F whose free variables include only the formal parameters and it specifies what should be true upon entering the function that is under what inputs the function is expected to work.

The function postcondition is a formula G whose free variables include only the formal parameters and the special variable rv representing the return value of the function; it relates the function's output (the return value rv) to its input (the parameters).

Example (Specification)

@pre $0 \leq l \wedge u < |a|$

@post $rv \iff \underset{i}{\exists}\ l \leq i \leq u \wedge a[i] = e$

```
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;
}
```

Other spec:

- @pre $\top$
- @post $rv \iff \underset{i}{\exists}\ 0 \leq l \leq i \leq u < |a| \wedge a[i] = e$

Annotations: Function Specification

An annotation is a FOL formula F whose free variables include only the program variables of the function in which the annotation occurs. An annotation F at location L asserts that F is true whenever program control reaches L.

We have: (1) (function) specification, (2) loop invariants, (3) assertions.

Function specification of a function is a pair of annotations.

The function precondition is a formula F whose free variables include only the formal parameters and it specifies what should be true upon entering the function that is under what inputs the function is expected to work.

The function postcondition is a formula G whose free variables include only the formal parameters and the special variable rv representing the return value of the function; it relates the function's output (the return value rv) to its input (the parameters).

Example (Specification)

@pre $0 \leq l \wedge u < |a|$

@post $rv \iff \underset{i}{\exists}\ l \leq i \leq u \wedge a[i] = e$

```
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;
}
```

Other spec:

- @pre $\top$
- @post $rv \iff \underset{i}{\exists}\ 0 \leq l \leq i \leq u < |a| \wedge a[i] = e$

Annotations: Function Specification

An annotation is a FOL formula F whose free variables include only the program variables of the function in which the annotation occurs. An annotation F at location L asserts that F is true whenever program control reaches L.

We have: (1) (function) specification, (2) loop invariants, (3) assertions.

Function specification of a function is a pair of annotations.

The function precondition is a formula F whose free variables include only the formal parameters and it specifies what should be true upon entering the function that is under what inputs the function is expected to work.

The function postcondition is a formula G whose free variables include only the formal parameters and the special variable rv representing the return value of the function; it relates the function's output (the return value rv) to its input (the parameters).

Example (Specification)

@pre $0 \leq l \land u < |a|$

@post $rv \iff \underset{i}{\exists}\ l \leq i \leq u \land a[i] = e$

```
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;
}
```

Other spec:

- @pre $\top$
- @post $rv \iff \underset{i}{\exists}\ 0 \leq l \leq i \leq u < |a| \land a[i] = e$

Annotations: Function Specification

An annotation is a FOL formula F whose free variables include only the program variables of the function in which the annotation occurs. An annotation F at location L asserts that F is true whenever program control reaches L.

We have: (1) (function) specification, (2) loop invariants, (3) assertions.

Function specification of a function is a pair of annotations.

The function precondition is a formula F whose free variables include only the formal parameters and it specifies what should be true upon entering the function that is under what inputs the function is expected to work.

The function postcondition is a formula G whose free variables include only the formal parameters and the special variable rv representing the return value of the function; it relates the function's output (the return value rv) to its input (the parameters).

Example (Specification)

@pre $0 \leq l \wedge u < |a|$

@post $rv \iff \underset{i}{\exists}\ l \leq i \leq u \wedge a[i] = e$

```
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;
}
```

Other spec:

- @pre $\top$
- @post $rv \iff \underset{i}{\exists}\ 0 \leq l \leq i \leq u < |a| \wedge a[i] = e$

Annotations: Function Specification

An annotation is a FOL formula F whose free variables include only the program variables of the function in which the annotation occurs. An annotation F at location L asserts that F is true whenever program control reaches L.
We have: (1) (function) specification, (2) loop invariants, (3) assertions.
Function specification of a function is a pair of annotations.
The function precondition is a formula F whose free variables include only the formal parameters and it specifies what should be true upon entering the function that is under what inputs the function is expected to work.
The function postcondition is a formula G whose free variables include only the formal parameters and the special variable rv representing the return value of the function; it relates the function's output (the return value rv) to its input (the parameters).

Example (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;
}
```

Other spec:

- @pre $\top$
- @post $rv \iff \exists_i\ 0 \le l \le i \le u < |a| \land a[i] = e$

Annotations: Function Specification

An annotation is a FOL formula F whose free variables include only the program variables of the function in which the annotation occurs. An annotation F at location L asserts that F is true whenever program control reaches L.
We have: (1) (function) specification, (2) loop invariants, (3) assertions.
Function specification of a function is a pair of annotations.
The function precondition is a formula F whose free variables include only the formal parameters and it specifies what should be true upon entering the function that is under what inputs the function is expected to work.
The function postcondition is a formula G whose free variables include only the formal parameters and the special variable rv representing the return value of the function; it relates the function's output (the return value rv) to its input (the parameters).

Example (Specification)

`@pre` $0 \leq l \wedge u < |a|$
`@post` $rv \iff \exists_{i}\ l \leq i \leq u \wedge a[i] = e$

```
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;
}
```

Other spec:

- `@pre` $\top$
- `@post` $rv \iff \exists_{i}\ 0 \leq l \leq i \leq u < |a| \wedge a[i] = e$

Annotations: Function Specification

An annotation is a FOL formula F whose free variables include only the program variables of the function in which the annotation occurs. An annotation F at location L asserts that F is true whenever program control reaches L.

We have: (1) (function) specification, (2) loop invariants, (3) assertions.

Function specification of a function is a pair of annotations.

The function precondition is a formula F whose free variables include only the formal parameters and it specifies what should be true upon entering the function that is under what inputs the function is expected to work.

The function postcondition is a formula G whose free variables include only the formal parameters and the special variable rv representing the return value of the function; it relates the function's output (the return value rv) to its input (the parameters).

Example (Specification)

@pre $0 \leq l \wedge u < |a|$

@post $rv \iff \underset{i}{\exists}\ l \leq i \leq u \wedge a[i] = e$

```
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;
}
```

Other spec:

- @pre $\top$
- @post $rv \iff \underset{i}{\exists}\ 0 \leq l \leq i \leq u < |a| \wedge a[i] = e$

Annotations: Function Specification

An annotation is a FOL formula F whose free variables include only the program variables of the function in which the annotation occurs. An annotation F at location L asserts that F is true whenever program control reaches L.

We have: (1) (function) specification, (2) loop invariants, (3) assertions.

Function specification of a function is a pair of annotations.

The function precondition is a formula F whose free variables include only the formal parameters and it specifies what should be true upon entering the function that is under what inputs the function is expected to work.

The function postcondition is a formula G whose free variables include only the formal parameters and the special variable rv representing the return value of the function; it relates the function's output (the return value rv) to its input (the parameters).

Example (Specification)

@pre $0 \leq l \land u < |a|$

@post $rv \iff \underset{i}{\exists}\ l \leq i \leq u \land a[i] = e$

```
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;
}
```

Other spec:

- @pre $\top$
- @post $rv \iff \underset{i}{\exists}\ 0 \leq l \leq i \leq u < |a| \land a[i] = e$

Annotations: Function Specification

An annotation is a FOL formula F whose free variables include only the program variables of the function in which the annotation occurs. An annotation F at location L asserts that F is true whenever program control reaches L.
We have: (1) (function) specification, (2) loop invariants, (3) assertions.
Function specification of a function is a pair of annotations.
The function precondition is a formula F whose free variables include only the formal parameters and it specifies what should be true upon entering the function that is under what inputs the function is expected to work.
The function postcondition is a formula G whose free variables include only the formal parameters and the special variable rv representing the return value of the function; it relates the function's output (the return value rv) to its input (the parameters).

Example (Specification)

@pre $0 \leq l \wedge u < |a|$
@post $rv \iff \underset{i}{\exists}\; l \leq i \leq u \wedge a[i] = e$

```
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;
}
```

Other spec:

- @pre $\top$
- @post $rv \iff \underset{i}{\exists}\; 0 \leq l \leq i \leq u < |a| \wedge a[i] = e$

Annotations: Loop invariant

Each `for` loop and `while` loop has an attendant annotation called the loop invariant.

For example:

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

says to apply the ⟨*body*⟩ as long as ⟨*condition*⟩ holds. The assertion F must hold at the beginning of every iteration. F is evaluated before the ⟨*condition*⟩ is evaluated, so it must hold even on the final iteration when ⟨*condition*⟩ is false. Therefore, on entering the ⟨*body*⟩ of the loop, $F \wedge \langle \mathit{condition} \rangle$ must hold, and on exiting the loop, $F \wedge \neg\langle \mathit{condition} \rangle$ must hold.

A `for` loop is as follows:

```
for
        @F
        (⟨initialize⟩; ⟨condition⟩; ⟨increment⟩) {
        ⟨body⟩
}
```

and analyzing it is the same as for `while` loops since `for` loops can be easily translated into `while` loops.

Annotations: Loop invariant

Each `for` loop and `while` loop has an attendant annotation called the loop invariant. For example:

while
    @F
    ($\langle$*condition*$\rangle$) {
    $\langle$*body*$\rangle$
}

says to apply the $\langle$*body*$\rangle$ as long as $\langle$*condition*$\rangle$ holds. The assertion F must hold at the beginning of every iteration. F is evaluated before the $\langle$*condition*$\rangle$ is evaluated, so it must hold even on the final iteration when $\langle$*condition*$\rangle$ is false. Therefore, on entering the $\langle$*body*$\rangle$ of the loop, $F \wedge \langle condition \rangle$ must hold, and on exiting the loop, $F \wedge \neg \langle condition \rangle$ must hold.

A `for` loop is as follows:

for
    @F
    ($\langle$*initialize*$\rangle$; $\langle$*condition*$\rangle$; $\langle$*increment*$\rangle$) {
    $\langle$*body*$\rangle$
}

and analyzing it is the same as for `while` loops since `for` loops can be easily translated into `while` loops.

Annotations: Loop invariant

Each `for` loop and `while` loop has an attendant annotation called the loop invariant. For example:

while
  @F
  ($\langle condition \rangle$) {
  $\langle body \rangle$
}

says to apply the $\langle body \rangle$ as long as $\langle condition \rangle$ holds. The assertion F must hold at the beginning of every iteration. F is evaluated before the $\langle condition \rangle$ is evaluated, so it must hold even on the final iteration when $\langle condition \rangle$ is false. Therefore, on entering the $\langle body \rangle$ of the loop, $F \wedge \langle condition \rangle$ must hold, and on exiting the loop, $F \wedge \neg \langle condition \rangle$ must hold.

A `for` loop is as follows:

for
  @F
  ($\langle initialize \rangle$; $\langle condition \rangle$; $\langle increment \rangle$) {
  $\langle body \rangle$
}

and analyzing it is the same as for `while` loops since `for` loops can be easily translated into `while` loops.

Annotations: Loop invariant and assertions

Example (Loop invariant)

@pre $0 \leq l \wedge u < |a|$
@post $rv \iff \underset{l \leq i \leq u}{\exists_i} \; a[i] = e$

bool LinearSearch(int[] a, int l, int u, int e) {
    for
        @L : $l \leq i \wedge (\underset{l \leq j < i}{\forall_j} \; a[j] \neq e)$
        (int $i := l; i \leq u; i := i + 1$) {
        if ($a[i] = e$) return true;
    }
    return false;
}

When an annotation is not a precondition/postcondition, or loop invariant, we call it an assertion.

Assertions allow programmers to provide a formal comment.

Example

If at the statement $i := i + k$; the programmer thinks that k is positive, then the programmer can add an assertion stating that supposition, i.e. @ $k > 0$; before $i := i + k$; The assertion @ $k > 0$; will be either formally verified (at compile time) or checked with dynamic assertion tests (at runtime).

Runtime assertions are a special class of assertions which allow avoiding runtime errors such as: division by 0, modulo by 0, and dereference of `null`.

Annotations: Loop invariant and assertions

Example (Loop invariant)

@pre $0 \leq l \wedge u < |a|$
@post $rv \iff \underset{l \leq i \leq u}{\underset{i}{\exists}} \; a[i] = e$

```
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;
}
```

When an annotation is not a precondition/postcondition, or loop invariant, we call it an assertion.

Assertions allow programmers to provide a formal comment.

Example

If at the statement $i := i + k$; the programmer thinks that k is positive, then the programmer can add an assertion stating that supposition, i.e. `@` $k > 0$; before $i := i + k$; The assertion `@` $k > 0$; will be either formally verified (at compile time) or checked with dynamic assertion tests (at runtime).

Runtime assertions are a special class of assertions which allow avoiding runtime errors such as: division by 0, modulo by 0, and dereference of `null`.

Annotations: Loop invariant and assertions

Example (Loop invariant)

```
@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;
 }
```

When an annotation is not a precondition/postcondition, or loop invariant, we call it an assertion.

Assertions allow programmers to provide a formal comment.

Example

If at the statement $i := i + k$; the programmer thinks that k is positive, then the programmer can add an assertion stating that supposition, i.e. `@` $k > 0$; before $i := i + k$; The assertion `@` $k > 0$; will be either formally verified (at compile time) or checked with dynamic assertion tests (at runtime).

Runtime assertions are a special class of assertions which allow avoiding runtime errors such as: division by 0, modulo by 0, and dereference of `null`.

Annotations: Loop invariant and assertions

Example (Loop invariant)

$$@\texttt{pre}\ 0 \le l \wedge u < |a|$$
$$@\texttt{post}\ rv \iff \underset{l \le i \le u}{\exists_i}\ a[i] = e$$

bool LinearSearch(int[] a, int l, int u, int e) {
    for
        $@L:\ l \le i \wedge (\underset{l \le j < i}{\forall_j}\ a[j] \neq e)$
        (int $i := l; i \le u; i := i + 1$) {
        if ($a[i] = e$) return true;
    }
    return false;
}

When an annotation is not a precondition/postcondition, or loop invariant, we call it an assertion.

Assertions allow programmers to provide a formal comment.

Example

If at the statement $i := i + k$; the programmer thinks that k is positive, then the programmer can add an assertion stating that supposition, i.e. @ $k > 0$; before $i := i + k$; The assertion @ $k > 0$; will be either formally verified (at compile time) or checked with dynamic assertion tests (at runtime).

Runtime assertions are a special class of assertions which allow avoiding runtime errors such as: division by 0, modulo by 0, and dereference of null.

Annotations: Loop invariant and assertions

Example (Loop invariant)

```
@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;
 }
```

When an annotation is not a precondition/postcondition, or loop invariant, we call it an assertion.

Assertions allow programmers to provide a formal comment.

Example

If at the statement $i := i + k$; the programmer thinks that k is positive, then the programmer can add an assertion stating that supposition, i.e. `@` $k > 0$; before $i := i + k$; The assertion `@` $k > 0$; will be either formally verified (at compile time) or checked with dynamic assertion tests (at runtime).

Runtime assertions are a special class of assertions which allow avoiding runtime errors such as: division by 0, modulo by 0, and dereference of `null`.

Outline

Classic Verification Seminal Papers

- 1967: Assigning Meaning to Programs (Floyd) [**?**]
  - 1978 Turing Award
- 1969: An Axiomatic Basis for Computer Programming (Hoare) [**?**]
  - 1980 Turing Award
- Guarded Commands, Nondeterminacy and Formal Derivation of Programs (Dijkstra) [**?**]
  - 1972 Turing Award

Preliminaries (cont'd)

Aim: prove that the functions obey their specifications.

Definition (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).

Method for proving partial correctness: inductive assertion method.

Principle of the inductive assertion method: reducing each function and its annotations to a finite set of verification conditions (VCs), which are FOL formulae.

Relationship with program correctness: If all of a function's VCs are valid, then the function obeys its specification.

Reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths
2. each basic path generates a verification condition

Observation: Loops and recursive functions complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit.

A path is a sequence of program statements.

- Loops: loop invariants cut the paths into a finite set of basic paths
- Recursive functions: the function specification of the recursive function cuts the paths.

In this course, we will focus on the task of generating the verification conditions (VCs) given specification and loop invariants.

In practice, the generation of VCs is performed by a verifying compiler or a verification conditions generator (VCG).

Preliminaries (cont'd)

Aim: prove that the functions obey their specifications.

Definition (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).

Method for proving partial correctness: inductive assertion method.

Principle of the inductive assertion method: reducing each function and its annotations to a finite set of verification conditions (VCs), which are FOL formulae.

Relationship with program correctness: If all of a function's VCs are valid, then the function obeys its specification.

Reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths
2. each basic path generates a verification condition

Observation: Loops and recursive functions complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit.

A path is a sequence of program statements.

- Loops: loop invariants cut the paths into a finite set of basic paths
- Recursive functions: the function specification of the recursive function cuts the paths.

In this course, we will focus on the task of generating the verification conditions (VCs) given specification and loop invariants.

In practice, the generation of VCs is performed by a verifying compiler or a verification conditions generator (VCG).

Preliminaries (cont'd)

Aim: prove that the functions obey their specifications.

Definition (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).

Method for proving partial correctness: inductive assertion method.

Principle of the inductive assertion method: reducing each function and its annotations to a finite set of verification conditions (VCs), which are FOL formulae.

Relationship with program correctness: If all of a function's VCs are valid, then the function obeys its specification.

Reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths
2. each basic path generates a verification condition

Observation: Loops and recursive functions complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit.

A path is a sequence of program statements.

- Loops: loop invariants cut the paths into a finite set of basic paths
- Recursive functions: the function specification of the recursive function cuts the paths.

In this course, we will focus on the task of generating the verification conditions (VCs) given specification and loop invariants.

In practice, the generation of VCs is performed by a verifying compiler or a verification conditions generator (VCG).

Preliminaries (cont'd)

Aim: prove that the functions obey their specifications.

Definition (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).

Method for proving partial correctness: inductive assertion method.

Principle of the inductive assertion method: reducing each function and its annotations to a finite set of verification conditions (VCs), which are FOL formulae.

Relationship with program correctness: If all of a function's VCs are valid, then the function obeys its specification.

Reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths
2. each basic path generates a verification condition

Observation: Loops and recursive functions complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit.

A path is a sequence of program statements.

- Loops: loop invariants cut the paths into a finite set of basic paths
- Recursive functions: the function specification of the recursive function cuts the paths.

In this course, we will focus on the task of generating the verification conditions (VCs) given specification and loop invariants.

In practice, the generation of VCs is performed by a verifying compiler or a verification conditions generator (VCG).

Preliminaries (cont'd)

Aim: prove that the functions obey their specifications.

Definition (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).

Method for proving partial correctness: inductive assertion method.

Principle of the inductive assertion method: reducing each function and its annotations to a finite set of verification conditions (VCs), which are FOL formulae.

Relationship with program correctness: If all of a function's VCs are valid, then the function obeys its specification.

Reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths
2. each basic path generates a verification condition

Observation: Loops and recursive functions complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit.

A path is a sequence of program statements.

- Loops: loop invariants cut the paths into a finite set of basic paths
- Recursive functions: the function specification of the recursive function cuts the paths.

In this course, we will focus on the task of generating the verification conditions (VCs) given specification and loop invariants.

In practice, the generation of VCs is performed by a verifying compiler or a verification conditions generator (VCG).

Preliminaries (cont'd)

Aim: prove that the functions obey their specifications.

Definition (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).

Method for proving partial correctness: inductive assertion method.

Principle of the inductive assertion method: reducing each function and its annotations to a finite set of verification conditions (VCs), which are FOL formulae.

Relationship with program correctness: If all of a function's VCs are valid, then the function obeys its specification.

Reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths
2. each basic path generates a verification condition

Observation: Loops and recursive functions complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit.

A path is a sequence of program statements.

- Loops: loop invariants cut the paths into a finite set of basic paths
- Recursive functions: the function specification of the recursive function cuts the paths.

In this course, we will focus on the task of generating the verification conditions (VCs) given specification and loop invariants.

In practice, the generation of VCs is performed by a verifying compiler or a verification conditions generator (VCG).

Preliminaries (cont'd)

Aim: prove that the functions obey their specifications.

Definition (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).

Method for proving partial correctness: inductive assertion method.

Principle of the inductive assertion method: reducing each function and its annotations to a finite set of verification conditions (VCs), which are FOL formulae.

Relationship with program correctness: If all of a function's VCs are valid, then the function obeys its specification.

Reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths
2. each basic path generates a verification condition

Observation: Loops and recursive functions complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit.

A path is a sequence of program statements.

- Loops: loop invariants cut the paths into a finite set of basic paths
- Recursive functions: the function specification of the recursive function cuts the paths.

In this course, we will focus on the task of generating the verification conditions (VCs) given specification and loop invariants.

In practice, the generation of VCs is performed by a verifying compiler or a verification conditions generator (VCG).

Preliminaries (cont'd)

Aim: prove that the functions obey their specifications.

Definition (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).

Method for proving partial correctness: inductive assertion method.

Principle of the inductive assertion method: reducing each function and its annotations to a finite set of verification conditions (VCs), which are FOL formulae.

Relationship with program correctness: If all of a function's VCs are valid, then the function obeys its specification.

Reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths
2. each basic path generates a verification condition

Observation: Loops and recursive functions complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit.

A path is a sequence of program statements.

- Loops: loop invariants cut the paths into a finite set of basic paths
- Recursive functions: the function specification of the recursive function cuts the paths.

In this course, we will focus on the task of generating the verification conditions (VCs) given specification and loop invariants.

In practice, the generation of VCs is performed by a verifying compiler or a verification conditions generator (VCG).

Preliminaries (cont'd)

Aim: prove that the functions obey their specifications.

Definition (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).

Method for proving partial correctness: inductive assertion method.

Principle of the inductive assertion method: reducing each function and its annotations to a finite set of verification conditions (VCs), which are FOL formulae.

Relationship with program correctness: If all of a function's VCs are valid, then the function obeys its specification.

Reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths
2. each basic path generates a verification condition

Observation: Loops and recursive functions complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit.

A path is a sequence of program statements.

- Loops: loop invariants cut the paths into a finite set of basic paths
- Recursive functions: the function specification of the recursive function cuts the paths.

In this course, we will focus on the task of generating the verification conditions (VCs) given specification and loop invariants.

In practice, the generation of VCs is performed by a verifying compiler or a verification conditions generator (VCG).

Preliminaries (cont'd)

Aim: prove that the functions obey their specifications.

Definition (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).

Method for proving partial correctness: inductive assertion method.

Principle of the inductive assertion method: reducing each function and its annotations to a finite set of verification conditions (VCs), which are FOL formulae.

Relationship with program correctness: If all of a function's VCs are valid, then the function obeys its specification.

Reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths
2. each basic path generates a verification condition

Observation: Loops and recursive functions complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit.

A path is a sequence of program statements.

- Loops: loop invariants cut the paths into a finite set of basic paths
- Recursive functions: the function specification of the recursive function cuts the paths.

In this course, we will focus on the task of generating the verification conditions (VCs) given specification and loop invariants.

In practice, the generation of VCs is performed by a verifying compiler or a verification conditions generator (VCG).

Preliminaries (cont'd)

Aim: prove that the functions obey their specifications.

Definition (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).

Method for proving partial correctness: inductive assertion method.

Principle of the inductive assertion method: reducing each function and its annotations to a finite set of verification conditions (VCs), which are FOL formulae.

Relationship with program correctness: If all of a function's VCs are valid, then the function obeys its specification.

Reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths
2. each basic path generates a verification condition

Observation: Loops and recursive functions complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit.

A path is a sequence of program statements.

- Loops: loop invariants cut the paths into a finite set of basic paths
- Recursive functions: the function specification of the recursive function cuts the paths.

In this course, we will focus on the task of generating the verification conditions (VCs) given specification and loop invariants.

In practice, the generation of VCs is performed by a verifying compiler or a verification conditions generator (VCG).

Preliminaries (cont'd)

Aim: prove that the functions obey their specifications.

Definition (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).

Method for proving partial correctness: inductive assertion method.

Principle of the inductive assertion method: reducing each function and its annotations to a finite set of verification conditions (VCs), which are FOL formulae.

Relationship with program correctness: If all of a function's VCs are valid, then the function obeys its specification.

Reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths
2. each basic path generates a verification condition

Observation: Loops and recursive functions complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit.

A path is a sequence of program statements.

- Loops: loop invariants cut the paths into a finite set of basic paths
- Recursive functions: the function specification of the recursive function cuts the paths.

In this course, we will focus on the task of generating the verification conditions (VCs) given specification and loop invariants.

In practice, the generation of VCs is performed by a verifying compiler or a verification conditions generator (VCG).

Preliminaries (cont'd)

Aim: prove that the functions obey their specifications.

Definition (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).

Method for proving partial correctness: inductive assertion method.

Principle of the inductive assertion method: reducing each function and its annotations to a finite set of verification conditions (VCs), which are FOL formulae.

Relationship with program correctness: If all of a function's VCs are valid, then the function obeys its specification.

Reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths
2. each basic path generates a verification condition

Observation: Loops and recursive functions complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit.

A path is a sequence of program statements.

- Loops: loop invariants cut the paths into a finite set of basic paths
- Recursive functions: the function specification of the recursive function cuts the paths.

In this course, we will focus on the task of generating the verification conditions (VCs) given specification and loop invariants.

In practice, the generation of VCs is performed by a verifying compiler or a verification conditions generator (VCG).

Preliminaries (cont'd)

Aim: prove that the functions obey their specifications.

Definition (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).

Method for proving partial correctness: inductive assertion method.

Principle of the inductive assertion method: reducing each function and its annotations to a finite set of verification conditions (VCs), which are FOL formulae.

Relationship with program correctness: If all of a function's VCs are valid, then the function obeys its specification.

Reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths
2. each basic path generates a verification condition

Observation: Loops and recursive functions complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit.

A path is a sequence of program statements.

- Loops: loop invariants cut the paths into a finite set of basic paths
- Recursive functions: the function specification of the recursive function cuts the paths.

In this course, we will focus on the task of generating the verification conditions (VCs) given specification and loop invariants.

In practice, the generation of VCs is performed by a verifying compiler or a verification conditions generator (VCG).

Basic Paths: Loops

Definition

A basic path is a sequence of instructions that begins at the function precondition or a loop invariant and ends at a loop invariant, an assertion, or the function postcondition. A loop invariant can only occur at the beginning or the ending of a basic path, hence basic paths do not cross loops.

Example

@pre $0 \leq l \wedge u < |a|$

@post $rv \iff \underset{i}{\exists}\ l \leq i \leq u \wedge a[i] = e$

bool LinearSearch(int[] a, int l, int u, int e) {

for @$L : l \leq i \wedge \left(\underset{l \leq j < i}{\underset{j}{\forall}}\ a[j] \neq e \right)$

(int $i := l$; $i \leq u$; $i := i + 1$) {

if ($a[i] = e$) return true;

}

return false;

}

How many basic paths are there?

Basic Paths: Loops

Definition

A basic path is a sequence of instructions that begins at the function precondition or a loop invariant and ends at a loop invariant, an assertion, or the function postcondition. A loop invariant can only occur at the beginning or the ending of a basic path, hence basic paths do not cross loops.

Example

@pre $0 \leq l \wedge u < |a|$

@post $rv \iff \underset{i}{\exists}\; l \leq i \leq u \wedge a[i] = e$

bool LinearSearch(int[] a, int l, int u, int e) {

for @$L : l \leq i \wedge \left(\underset{l \leq j < i}{\underset{j}{\forall}}\; a[j] \neq e \right)$

(int $i := l$; $i \leq u$; $i := i + 1$) {

if ($a[i] = e$) return true;

}

return false;

}

How many basic paths are there?

Program States

A program state s is an assignment of values of the proper type to all variables.

The program variables include a distinguished variable pc – the program counter, holding the current location of control.

Example

For `LinearSearch` from previous slide, a possible state of the program is:

$$s : \{pc \mapsto L, a \mapsto [2, 0, 1], i \mapsto 1, j \mapsto 1, l \mapsto 0, u \mapsto 2, e \mapsto 12, rv \mapsto true\}$$

Program States

A program state s is an assignment of values of the proper type to all variables.

The program variables include a distinguished variable pc – the program counter, holding the current location of control.

Example

For `LinearSearch` from previous slide, a possible state of the program is:

$$s : \{pc \mapsto L, a \mapsto [2,0,1], i \mapsto 1, j \mapsto 1, l \mapsto 0, u \mapsto 2, e \mapsto 12, rv \mapsto true\}$$

Program States

A program state s is an assignment of values of the proper type to all variables.

The program variables include a distinguished variable pc – the program counter, holding the current location of control.

Example

For `LinearSearch` from previous slide, a possible state of the program is:

$$s : \{pc \mapsto L, a \mapsto [2,0,1], i \mapsto 1, j \mapsto 1, l \mapsto 0, u \mapsto 2, e \mapsto 12, rv \mapsto true\}$$

Verification Conditions

Aim: reduce an annotated function to a finite set of FOL formulae (verification conditions) such that their validity implies that the function's behavior agrees with its annotations.

Recall the reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths ✓
2. each basic path generates a verification condition – requires a mechanism for incorporating the effects of program statements into FOL formulae $\rightsquigarrow$ (1) weakest precondition predicate transformer, (2) strongest postcondition predicate transformer.

Definition

A predicate transformer p is a function $p : \mathit{stmts} \times \mathit{FOL} \to \mathit{FOL}$ that maps a FOL formula $F \in \mathit{FOL}$ and program statement $S \in \mathit{stmts}$ to a FOL formula.

Then the verification condition of basic path
@F
S_1;
$\vdots$
S_n;
@G
is $F \Rightarrow wp(G, S_1; \ldots; S_n)$.
The VC above is denoted by the Hoare triple $\{F\}S_1; \ldots; S_n\{G\}$.

Verification Conditions

Aim: reduce an annotated function to a finite set of FOL formulae (verification conditions) such that their validity implies that the function's behavior agrees with its annotations.

Recall the reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths ✓
2. each basic path generates a verification condition – requires a mechanism for incorporating the effects of program statements into FOL formulae ⇝ (1) weakest precondition predicate transformer, (2) strongest postcondition predicate transformer.

Definition
A predicate transformer p is a function $p : stmts \times FOL \to FOL$ that maps a FOL formula $F \in FOL$ and program statement $S \in stmts$ to a FOL formula.

Then the verification condition of basic path
@F
S_1;
⋮
S_n;
@G
is $F \Rightarrow wp(G, S_1; \ldots; S_n)$.
The VC above is denoted by the Hoare triple $\{F\}S_1; \ldots; S_n\{G\}$.

Verification Conditions

Aim: reduce an annotated function to a finite set of FOL formulae (verification conditions) such that their validity implies that the function's behavior agrees with its annotations.

Recall the reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths ✓
2. each basic path generates a verification condition – requires a mechanism for incorporating the effects of program statements into FOL formulae $\rightsquigarrow$ (1) weakest precondition predicate transformer, (2) strongest postcondition predicate transformer.

Definition
A predicate transformer p is a function $p : \mathit{stmts} \times \mathit{FOL} \rightarrow \mathit{FOL}$ that maps a FOL formula $F \in \mathit{FOL}$ and program statement $S \in \mathit{stmts}$ to a FOL formula.

Then the verification condition of basic path
@F
S_1 ;
$\vdots$
S_n ;
@G
is $F \Rightarrow wp(G, S_1; \ldots; S_n)$.
The VC above is denoted by the Hoare triple $\{F\}S_1; \ldots; S_n\{G\}$.

Verification Conditions

Aim: reduce an annotated function to a finite set of FOL formulae (verification conditions) such that their validity implies that the function's behavior agrees with its annotations.

Recall the reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths ✓
2. each basic path generates a verification condition – requires a mechanism for incorporating the effects of program statements into FOL formulae ⇝ (1) weakest precondition predicate transformer, (2) strongest postcondition predicate transformer.

Definition

A predicate transformer p is a function $p : \mathit{stmts} \times \mathit{FOL} \to \mathit{FOL}$ that maps a FOL formula $F \in \mathit{FOL}$ and program statement $S \in \mathit{stmts}$ to a FOL formula.

Then the verification condition of basic path
@F
S_1;
⋮
S_n;
@G
is $F \Rightarrow wp(G, S_1; \ldots; S_n)$.
The VC above is denoted by the Hoare triple $\{F\}S_1; \ldots; S_n\{G\}$.

Verification Conditions

Aim: reduce an annotated function to a finite set of FOL formulae (verification conditions) such that their validity implies that the function's behavior agrees with its annotations.

Recall the reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths ✓
2. each basic path generates a verification condition – requires a mechanism for incorporating the effects of program statements into FOL formulae $\rightsquigarrow$ (1) weakest precondition predicate transformer, (2) strongest postcondition predicate transformer.

Definition

A predicate transformer p is a function $p : \mathit{stmts} \times \mathit{FOL} \to \mathit{FOL}$ that maps a FOL formula $F \in \mathit{FOL}$ and program statement $S \in \mathit{stmts}$ to a FOL formula.

Then the verification condition of basic path

@F

S_1 ;

$\vdots$

S_n ;

@G

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

The VC above is denoted by the Hoare triple $\{F\}S_1; \ldots; S_n\{G\}$.

Verification Conditions

Aim: reduce an annotated function to a finite set of FOL formulae (verification conditions) such that their validity implies that the function's behavior agrees with its annotations.

Recall the reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths ✓
2. each basic path generates a verification condition – requires a mechanism for incorporating the effects of program statements into FOL formulae $\rightsquigarrow$ (1) weakest precondition predicate transformer, (2) strongest postcondition predicate transformer.

Definition

A predicate transformer p is a function $p : \mathit{stmts} \times \mathit{FOL} \to \mathit{FOL}$ that maps a FOL formula $F \in \mathit{FOL}$ and program statement $S \in \mathit{stmts}$ to a FOL formula.

Then the verification condition of basic path

`@`F

S_1;

$\vdots$

S_n;

`@`G

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

The VC above is denoted by the Hoare triple $\{F\}S_1; \ldots; S_n\{G\}$.

Verification Conditions

Aim: reduce an annotated function to a finite set of FOL formulae (verification conditions) such that their validity implies that the function's behavior agrees with its annotations.

Recall the reduction steps:

1. each function of the annotated program is broken down into a finite set of basic paths ✓
2. each basic path generates a verification condition – requires a mechanism for incorporating the effects of program statements into FOL formulae $\rightsquigarrow$ (1) weakest precondition predicate transformer, (2) strongest postcondition predicate transformer.

Definition

A predicate transformer p is a function $p : \mathit{stmts} \times \mathit{FOL} \rightarrow \mathit{FOL}$ that maps a FOL formula $F \in \mathit{FOL}$ and program statement $S \in \mathit{stmts}$ to a FOL formula.

Then the verification condition of basic path

$@F$

S_1;

$\vdots$

S_n;

$@G$

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

The VC above is denoted by the Hoare triple $\{F\}S_1; \ldots; S_n\{G\}$.

Verification Conditions (cont'd)

Example

For the basic path:
$@x \geq 0$
$x := x + 1;$
$@x \geq 1$

the VC is

$$\{x \geq 0\}\ x := x + 1\ \{x \geq 1\} \quad \rightsquigarrow \quad x \geq 0 \Rightarrow wp(x \geq 1, x := x + 1)$$

Computing wp, we have:

$$wp(x \geq 1, x := x + 1) \iff (x \geq 1)x \mapsto x + 1 \iff x + 1 \geq 1 \iff x \geq 0$$

Hence we have $x \geq 0 \Rightarrow x \geq 0 \iff \mathit{True}$

Systematic rules for wp computation are given in the following slides.

Verification Conditions (cont'd)

Example

For the basic path:
$@x \geq 0$
$x := x + 1;$
$@x \geq 1$

the VC is

$$\{x \geq 0\}\ x := x + 1\ \{x \geq 1\} \quad \rightsquigarrow \quad x \geq 0 \Rightarrow wp(x \geq 1, x := x + 1)$$

Computing *wp*, we have:

$$wp(x \geq 1, x := x + 1) \iff (x \geq 1)x \mapsto x + 1 \iff x + 1 \geq 1 \iff x \geq 0$$

Hence we have $x \geq 0 \Rightarrow x \geq 0 \iff \mathit{True}$

Systematic rules for *wp* computation are given in the following slides.

Verification Conditions (cont'd)

Example

For the basic path:

$$@L: \quad F : l \leq i \wedge \left(\underset{\substack{j \\ l \leq j < i}}{\forall} \quad a[j] \neq e \right)$$

S_1 : assume $i \leq u$;
S_2 : assume $a[i] = e$;
S_3 : $rv := true$;
@post: $G : rv \iff \underset{\substack{j \\ l \leq j \leq u}}{\exists} \quad a[j] = e$

the VC is $F \Rightarrow wp(G, S_1; S_2; S_3)$. Computing $wp(G, S_1; S_2; S_3)$, we have

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

$$\iff wp(wp(wp(rv \Leftrightarrow \underset{\substack{j \\ l \leq j \leq u}}{\exists} \quad a[j] = e, rv := true), S_2), S_1)$$

$$\iff wp(wp(true \Leftrightarrow \underset{\substack{j \\ l \leq j \leq u}}{\exists} \quad a[j] = e, S_2), S_1) \iff wp(wp(\underset{\substack{j \\ l \leq j \leq u}}{\exists} \quad a[j] = e), S_2), S_1)$$

$$\iff wp(wp(\underset{\substack{j \\ l \leq j \leq u}}{\exists} \quad a[j] = e, \texttt{assume } a[i] = e;), S_1) \iff wp(a[i] = e \Rightarrow \underset{\substack{j \\ l \leq j \leq u}}{\exists} \quad a[j] = e, S_1)$$

$$\iff wp(a[i] = e \Rightarrow \underset{\substack{j \\ l \leq j \leq u}}{\exists} \quad a[j] = e, \texttt{assume } i \leq u) \iff \left(i \leq u \Rightarrow a[i] = e \Rightarrow \underset{\substack{j \\ l \leq j \leq u}}{\exists} \quad a[j] = e \right.$$

Further we have (next slide)

Verification Conditions (cont'd)

We have to prove

$$
\begin{aligned}
& F \Rightarrow wp(G, S_1; S_2; S_3) \\
\iff & \left(l \leq i \land \left(\underset{\substack{j \\ l \leq j < i}}{\forall} a[j] \neq e \right) \right) \Rightarrow \left(i \leq u \Rightarrow a[i] = e \Rightarrow \underset{\substack{j \\ l \leq j \leq u}}{\exists} a[j] = e \right)
\end{aligned}
$$

We use the basic proof techniques for the proof. The goal holds by taking $j = i$ as witness.

How to compute systematically wp and verification conditions?

Verification Conditions (cont'd)

We have to prove

$$
\begin{aligned}
& F \Rightarrow wp(G, S_1; S_2; S_3) \\
\iff & \left(l \leq i \wedge \left(\underset{\substack{j \\ l \leq j < i}}{\forall} a[j] \neq e \right) \right) \Rightarrow \left(i \leq u \Rightarrow a[i] = e \Rightarrow \underset{\substack{j \\ l \leq j \leq u}}{\exists} a[j] = e \right)
\end{aligned}
$$

We use the basic proof techniques for the proof. The goal holds by taking $j = i$ as witness.

How to compute systematically wp and verification conditions?

Verification Conditions (cont'd)

We have to prove

$$
\begin{aligned}
& F \Rightarrow wp(G, S_1; S_2; S_3) \\
\iff & \left(l \leq i \wedge \left(\underset{\substack{j \\ l \leq j < i}}{\forall} a[j] \neq e \right) \right) \Rightarrow \left(i \leq u \Rightarrow a[i] = e \Rightarrow \underset{\substack{j \\ l \leq j \leq u}}{\exists} a[j] = e \right)
\end{aligned}
$$

We use the basic proof techniques for the proof. The goal holds by taking $j = i$ as witness.

How to compute systematically wp and verification conditions?

Specifying correctness in Hoare logic

$$\{P\}S\{Q\}$$

This is a so-called Hoare triple:

- S is a program statement (or fragment).
- P is an FOL formula called the precondition.
- Q is an FOL formula called the postcondition.

Partial correctness (safety) (Hoare triple semantics)
If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

Total correctness (liveness) $[P]S[Q]$
If S executes from a state satisfying P, then its execution terminates and the resulting state satisfies Q.

Examples of Hoare Triples

- $\{false\}S\{Q\}$: valid for all S and Q.
- $\{P\}$ *while* $(true)$ *do skip* $\{Q\}$: Valid for all P and Q.
- $\{true\}\ S\ \{Q\}$: if S terminates, then Q must hold.
- $\{P\}\ S\ \{true\}$: valid for all P and S.

Specifying correctness in Hoare logic

$$\{P\}S\{Q\}$$

This is a so-called Hoare triple:

- S is a program statement (or fragment).
- P is an FOL formula called the precondition.
- Q is an FOL formula called the postcondition.

Partial correctness (safety) (Hoare triple semantics)
If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

Total correctness (liveness) $[P]S[Q]$
If S executes from a state satisfying P, then its execution terminates and the resulting state satisfies Q.

Examples of Hoare Triples

- $\{false\}S\{Q\}$: valid for all S and Q.
- $\{P\}$ *while* $(true)$ *do skip* $\{Q\}$: Valid for all P and Q.
- $\{true\}$ S $\{Q\}$: if S terminates, then Q must hold.
- $\{P\}$ S $\{true\}$: valid for all P and S.

Specifying correctness in Hoare logic

$$\{P\}S\{Q\}$$

This is a so-called Hoare triple:

- S is a program statement (or fragment).
- P is an FOL formula called the precondition.
- Q is an FOL formula called the postcondition.

Partial correctness (safety) (Hoare triple semantics)
If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

Total correctness (liveness) $[P]S[Q]$
If S executes from a state satisfying P, then its execution terminates and the resulting state satisfies Q.

Examples of Hoare Triples

- $\{false\}S\{Q\}$: valid for all S and Q.
- $\{P\}$ *while* $(true)$ *do skip* $\{Q\}$: Valid for all P and Q.
- $\{true\}\ S\ \{Q\}$: if S terminates, then Q must hold.
- $\{P\}\ S\ \{true\}$: valid for all P and S.

Specifying correctness in Hoare logic

$$\{P\}S\{Q\}$$

This is a so-called Hoare triple:

- S is a program statement (or fragment).
- P is an FOL formula called the precondition.
- Q is an FOL formula called the postcondition.

Partial correctness (safety) (Hoare triple semantics)
If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

Total correctness (liveness) $[P]S[Q]$
If S executes from a state satisfying P, then its execution terminates and the resulting state satisfies Q.

Examples of Hoare Triples

- $\{false\}S\{Q\}$: valid for all S and Q.
- $\{P\}$ *while* $(true)$ *do skip* $\{Q\}$: Valid for all P and Q.
- $\{true\}\ S\ \{Q\}$: if S terminates, then Q must hold.
- $\{P\}\ S\ \{true\}$: valid for all P and S.

Specifying correctness in Hoare logic

$$\{P\}S\{Q\}$$

This is a so-called Hoare triple:

- S is a program statement (or fragment).
- P is an FOL formula called the precondition.
- Q is an FOL formula called the postcondition.

Partial correctness (safety) (Hoare triple semantics)
If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

Total correctness (liveness) $[P]S[Q]$
If S executes from a state satisfying P, then its execution terminates and the resulting state satisfies Q.

Examples of Hoare Triples

- $\{false\}S\{Q\}$: valid for all S and Q.
- $\{P\}$ *while* $(true)$ *do skip* $\{Q\}$: Valid for all P and Q.
- $\{true\}$ S $\{Q\}$: if S terminates, then Q must hold.
- $\{P\}$ S $\{true\}$: valid for all P and S.

Specifying correctness in Hoare logic

$$\{P\}S\{Q\}$$

This is a so-called Hoare triple:

- S is a program statement (or fragment).
- P is an FOL formula called the precondition.
- Q is an FOL formula called the postcondition.

Partial correctness (safety) (Hoare triple semantics)
If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

Total correctness (liveness) $[P]S[Q]$
If S executes from a state satisfying P, then its execution terminates and the resulting state satisfies Q.

Examples of Hoare Triples

- $\{false\}S\{Q\}$: valid for all S and Q.
- $\{P\}$ *while* $(true)$ *do skip* $\{Q\}$: Valid for all P and Q.
- $\{true\}$ S $\{Q\}$: if S terminates, then Q must hold.
- $\{P\}$ S $\{true\}$: valid for all P and S.

Specifying correctness in Hoare logic

$$\{P\}S\{Q\}$$

This is a so-called Hoare triple:

- S is a program statement (or fragment).
- P is an FOL formula called the precondition.
- Q is an FOL formula called the postcondition.

Partial correctness (safety) (Hoare triple semantics)
If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

Total correctness (liveness) $[P]S[Q]$
If S executes from a state satisfying P, then its execution terminates and the resulting state satisfies Q.

Examples of Hoare Triples

- $\{false\}S\{Q\}$: valid for all S and Q.
- $\{P\}$ *while* $(true)$ *do skip* $\{Q\}$: Valid for all P and Q.
- $\{true\}$ S $\{Q\}$: if S terminates, then Q must hold.
- $\{P\}$ S $\{true\}$: valid for all P and S.

Specifying correctness in Hoare logic

$$\{P\}S\{Q\}$$

This is a so-called Hoare triple:

- S is a program statement (or fragment).
- P is an FOL formula called the precondition.
- Q is an FOL formula called the postcondition.

Partial correctness (safety) (Hoare triple semantics)
If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

Total correctness (liveness) $[P]S[Q]$
If S executes from a state satisfying P, then its execution terminates and the resulting state satisfies Q.

Examples of Hoare Triples

- $\{false\}S\{Q\}$: valid for all S and Q.
- $\{P\}$ *while* $(true)$ *do skip* $\{Q\}$: Valid for all P and Q.
- $\{true\}\ S\ \{Q\}$: if S terminates, then Q must hold.
- $\{P\}\ S\ \{true\}$: valid for all P and S.

Specifying correctness in Hoare logic

$$\{P\}S\{Q\}$$

This is a so-called Hoare triple:

- S is a program statement (or fragment).
- P is an FOL formula called the precondition.
- Q is an FOL formula called the postcondition.

Partial correctness (safety) (Hoare triple semantics)
If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

Total correctness (liveness) $[P]S[Q]$
If S executes from a state satisfying P, then its execution terminates and the resulting state satisfies Q.

Examples of Hoare Triples

- $\{false\}S\{Q\}$: valid for all S and Q.
- $\{P\}$ *while* $(true)$ *do skip* $\{Q\}$: Valid for all P and Q.
- $\{true\}\ S\ \{Q\}$: if S terminates, then Q must hold.
- $\{P\}\ S\ \{true\}$: valid for all P and S.

Specifying correctness in Hoare logic

$$\{P\}S\{Q\}$$

This is a so-called Hoare triple:

- S is a program statement (or fragment).
- P is an FOL formula called the precondition.
- Q is an FOL formula called the postcondition.

Partial correctness (safety) (Hoare triple semantics)
If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

Total correctness (liveness) $[P]S[Q]$
If S executes from a state satisfying P, then its execution terminates and the resulting state satisfies Q.

Examples of Hoare Triples

- $\{\mathit{false}\}S\{Q\}$: valid for all S and Q.
- $\{P\}$ *while* (true) *do skip* $\{Q\}$: Valid for all P and Q.
- $\{\mathit{true}\}\ S\ \{Q\}$: if S terminates, then Q must hold.
- $\{P\}\ S\ \{\mathit{true}\}$: valid for all P and S.

Proving partial correctness in Hoare logic

We will use a simple imperative language

- Expression E:
  - $Z \mid V \mid E1 + E2 \mid E1 * E2$
- Condition C
  - $true \mid false \mid E1 = E2 \mid E1 \leq E2 \mid E1 < E2 \mid \ldots$
- Statement S
  - $skip$ (Skip)
  - $V := E$ (Assignment)
  - $S_1; S_2$ (Composition)
  - $if\ C\ then\ S_1\ else\ S_2$ (If)
  - $while\ C\ do\ S$ (While)

There is one inference rule (see next slide) for every statement in the language:

$$\frac{\vdash \{P_1\}S_1\{Q_1\} \quad \ldots \quad \vdash \{P_n\}S_n\{Q_n\}}{\vdash \{P\}S\{Q\}}$$

It reads" *If the Hoare triples* $\{P_1\}S_1\{Q_1\}\ldots\{P_n\}S_n\{Q_n\}$ *are provable, then so is* $\{P\}S\{Q\}$.

Proving partial correctness in Hoare logic

We will use a simple imperative language

- Expression E:
  - $Z \mid V \mid E1 + E2 \mid E1 * E2$
- Condition C
  - $true \mid false \mid E1 = E2 \mid E1 \leq E2 \mid E1 < E2 \mid \ldots$
- Statement S
  - $skip$ (Skip)
  - $V := E$ (Assignment)
  - $S_1; S_2$ (Composition)
  - $if\ C\ then\ S_1\ else\ S_2$ (If)
  - $while\ C\ do\ S$ (While)

There is one inference rule (see next slide) for every statement in the language:

$$\frac{\vdash \{P_1\}S_1\{Q_1\} \quad \ldots \quad \vdash \{P_n\}S_n\{Q_n\}}{\vdash \{P\}S\{Q\}}$$

It reads" *If the Hoare triples* $\{P_1\}S_1\{Q_1\} \ldots \{P_n\}S_n\{Q_n\}$ *are provable, then so is* $\{P\}S\{Q\}$.

Proving partial correctness in Hoare logic

We will use a simple imperative language

- Expression E:
  - $Z \mid V \mid E1 + E2 \mid E1 * E2$
- Condition C
  - $true \mid false \mid E1 = E2 \mid E1 \leq E2 \mid E1 < E2 \mid \ldots$
- Statement S
  - $skip$ (Skip)
  - $V := E$ (Assignment)
  - $S_1; S_2$ (Composition)
  - $if\ C\ then\ S_1\ else\ S_2$ (If)
  - $while\ C\ do\ S$ (While)

There is one inference rule (see next slide) for every statement in the language:

$$\frac{\vdash \{P_1\}S_1\{Q_1\} \quad \ldots \quad \vdash \{P_n\}S_n\{Q_n\}}{\vdash \{P\}S\{Q\}}$$

It reads" *If the Hoare triples* $\{P_1\}S_1\{Q_1\}\ldots\{P_n\}S_n\{Q_n\}$ *are provable, then so is* $\{P\}S\{Q\}$.

Proving partial correctness in Hoare logic

We will use a simple imperative language

- Expression E:
  - $Z \mid V \mid E1 + E2 \mid E1 * E2$
- Condition C
  - $true \mid false \mid E1 = E2 \mid E1 \leq E2 \mid E1 < E2 \mid \ldots$
- Statement S
  - $skip$ (Skip)
  - $V := E$ (Assignment)
  - $S_1; S_2$ (Composition)
  - $if\ \ C\ \ then\ S_1\ else\ S_2$ (If)
  - $while\ \ C\ \ do\ \ S$ (While)

There is one inference rule (see next slide) for every statement in the language:

$$\frac{\vdash \{P_1\}S_1\{Q_1\} \quad \ldots \quad \vdash \{P_n\}S_n\{Q_n\}}{\vdash \{P\}S\{Q\}}$$

It reads" *If the Hoare triples* $\{P_1\}S_1\{Q_1\}\ldots\{P_n\}S_n\{Q_n\}$ *are provable, then so is* $\{P\}S\{Q\}$.

Proving partial correctness in Hoare logic

We will use a simple imperative language

- Expression E:
  - $Z \mid V \mid E1 + E2 \mid E1 * E2$
- Condition C
  - $true \mid false \mid E1 = E2 \mid E1 \leq E2 \mid E1 < E2 \mid \ldots$
- Statement S
  - $skip$ (Skip)
  - $V := E$ (Assignment)
  - $S_1; S_2$ (Composition)
  - $if\ \ C\ \ then\ S_1\ else\ S_2$ (If)
  - $while\ \ C\ \ do\ \ S$ (While)

There is one inference rule (see next slide) for every statement in the language:

$$\frac{\vdash \{P_1\}S_1\{Q_1\} \quad \ldots \quad \vdash \{P_n\}S_n\{Q_n\}}{\vdash \{P\}S\{Q\}}$$

It reads" *If the Hoare triples* $\{P_1\}S_1\{Q_1\}...\{P_n\}S_n\{Q_n\}$ *are provable, then so is* $\{P\}S\{Q\}$.

Inference rules for Hoare logic

- $\dfrac{}{\vdash\{P\}skip\{P\}}$
- $\dfrac{}{\vdash\{Q\{x\to E\}\}x{:=}E\{Q\}}$
- $\dfrac{\vdash\{P\}S_1\{R\} \quad \{R\}S_2\{Q\}}{\vdash\{P\}S_1;S_2\{Q\}}$
- $\dfrac{\vdash\{P\wedge C\}S_1\{Q\} \quad \{P\wedge\neg C\}S_2\{Q\}}{\vdash\{P\}\ if\ \ C\ \ then\ S_1\ \ else\ \ S_2\ \ \{Q\}}$
- $\dfrac{\vdash\{I\wedge C\}S\{I\}}{\vdash\{I\}\ while\ \ C\ \ do\ \ S\ \ \{I\wedge\neg C\}}$ I is so-called invariant

Coming up with a suitable invariant is a challenging task (invariant synthesis).

Examples:
Assignment:

$$\{?\}\ x := x{+}3\ \{x < 5\} \ \rightsquigarrow\ \{x{+}3 < 5\}\ x := x{+}3\ \{x < 5\} \ \rightsquigarrow\ \{x < 2\}\ x := x{+}3\ \{x < 5\}$$

Sequence of commands:

$$\frac{\{x+y-1>0\}\ y := y-1\ \{x+y>0\} \qquad \{x+y>0\}\ x := x+y\ \{x>0\}}{\{x+y-1>0\}\ y := y-1; x := x+y\ \{x>0\}}$$

We will learn later how to derive systematically the condition in red with predicate transformers.
Conditional:

$$\frac{\{x \neq 0 \wedge x = 0\}\ y := x\ \{y > 0\} \qquad \{x \neq 0 \wedge \neg(x = 0)\}\ y := -x\ \{y > 0\}}{\{x \neq 0\}\ \texttt{if}\ \ x = 0\ \ \texttt{then}\ \ y := x\ \ \texttt{else}\ \ y := -x\ \{y > 0\}}$$

Inference rules for Hoare logic

- $\dfrac{}{\vdash \{P\} skip \{P\}}$
- $\dfrac{}{\vdash \{Q\{x \to E\}\} x{:=}E \{Q\}}$
- $\dfrac{\vdash \{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\vdash \{P\} S_1 ; S_2 \{Q\}}$
- $\dfrac{\vdash \{P \wedge C\} S_1 \{Q\} \quad \{P \wedge \neg C\} S_2 \{Q\}}{\vdash \{P\}\ if\ \ C\ \ then\ S_1\ \ else\ \ S_2\ \{Q\}}$
- $\dfrac{\vdash \{I \wedge C\} S \{I\}}{\vdash \{I\}\ while\ \ C\ \ do\ \ S\ \{I \wedge \neg C\}}$ $\quad$ I is so-called invariant

Coming up with a suitable invariant is a challenging task (invariant synthesis).

Examples:
Assignment:

$$\{?\}\ x := x{+}3\ \{x < 5\} \ \rightsquigarrow\ \{x{+}3 < 5\}\ x := x{+}3\ \{x < 5\} \ \rightsquigarrow\ \{x < 2\}\ x := x{+}3\ \{x < 5\}$$

Sequence of commands:

$$\frac{\{x + y - 1 > 0\}\ y := y - 1\ \{x + y > 0\} \qquad \{x + y > 0\}\ x := x + y\ \{x > 0\}}{\{x + y - 1 > 0\}\ y := y - 1; x := x + y\ \{x > 0\}}$$

We will learn later how to derive systematically the condition in red with predicate transformers.
Conditional:

$$\frac{\{x \neq 0 \wedge x = 0\}\ y := x\ \{y > 0\} \qquad \{x \neq 0 \wedge \neg(x = 0)\}\ y := -x\ \{y > 0\}}{\{x \neq 0\}\ \mathtt{if}\ \ x = 0\ \ \mathtt{then}\ \ y := x\ \ \mathtt{else}\ \ y := -x\ \{y > 0\}}$$

Inference rules for Hoare logic

- $\dfrac{}{\vdash \{P\} skip \{P\}}$
- $\dfrac{}{\vdash \{Q\{x \rightarrow E\}\} x{:=}E \{Q\}}$
- $\dfrac{\vdash \{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\vdash \{P\} S_1;S_2 \{Q\}}$
- $\dfrac{\vdash \{P \wedge C\} S_1 \{Q\} \quad \{P \wedge \neg C\} S_2 \{Q\}}{\vdash \{P\} \ if \ C \ then \ S_1 \ else \ S_2 \ \{Q\}}$
- $\dfrac{\vdash \{I \wedge C\} S \{I\}}{\vdash \{I\} \ while \ C \ do \ S \ \{I \wedge \neg C\}}$ $\quad$ I is so-called invariant

Coming up with a suitable invariant is a challenging task (invariant synthesis).

Examples:
Assignment:

$$\{?\}\ x := x{+}3\ \{x < 5\} \ \rightsquigarrow\ \{x{+}3 < 5\}\ x := x{+}3\ \{x < 5\} \ \rightsquigarrow\ \{x < 2\}\ x := x{+}3\ \{x < 5\}$$

Sequence of commands:

$$\dfrac{\{x + y - 1 > 0\}\ y := y - 1\ \{x + y > 0\} \qquad \{x + y > 0\}\ x := x + y\ \{x > 0\}}{\{x + y - 1 > 0\}\ y := y - 1; x := x + y\ \{x > 0\}}$$

We will learn later how to derive systematically the condition in red with predicate transformers.
Conditional:

$$\dfrac{\{x \neq 0 \wedge x = 0\}\ y := x\ \{y > 0\} \qquad \{x \neq 0 \wedge \neg(x = 0)\}\ y := -x\ \{y > 0\}}{\{x \neq 0\}\ \texttt{if}\ x = 0\ \texttt{then}\ y := x\ \texttt{else}\ y := -x\ \{y > 0\}}$$

Inference rules for Hoare logic

- $\dfrac{}{\vdash \{P\} skip \{P\}}$
- $\dfrac{}{\vdash \{Q\{x \to E\}\} x{:=}E \{Q\}}$
- $\dfrac{\vdash \{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\vdash \{P\} S_1; S_2 \{Q\}}$
- $\dfrac{\vdash \{P \wedge C\} S_1 \{Q\} \quad \{P \wedge \neg C\} S_2 \{Q\}}{\vdash \{P\} \ if \ C \ then \ S_1 \ else \ S_2 \ \{Q\}}$
- $\dfrac{\vdash \{I \wedge C\} S \{I\}}{\vdash \{I\} \ while \ C \ do \ S \ \{I \wedge \neg C\}}$ $\quad$ I is so-called invariant

Coming up with a suitable invariant is a challenging task (invariant synthesis).

Examples:
Assignment:

$$\{?\}\ x := x{+}3\ \{x < 5\} \ \rightsquigarrow \ \{x{+}3 < 5\}\ x := x{+}3\ \{x < 5\} \ \rightsquigarrow \ \{x < 2\}\ x := x{+}3\ \{x < 5\}$$

Sequence of commands:

$$\frac{\{x + y - 1 > 0\}\ y := y - 1\ \{x + y > 0\} \qquad \{x + y > 0\}\ x := x + y\ \{x > 0\}}{\{x + y - 1 > 0\}\ y := y - 1; x := x + y\ \{x > 0\}}$$

We will learn later how to derive systematically the condition in red with predicate transformers.
Conditional:

$$\frac{\{x \neq 0 \wedge x = 0\}\ y := x\ \{y > 0\} \qquad \{x \neq 0 \wedge \neg(x = 0)\}\ y := -x\ \{y > 0\}}{\{x \neq 0\}\ \texttt{if}\ x = 0\ \texttt{then}\ y := x\ \texttt{else}\ y := -x\ \{y > 0\}}$$

Inference rules for Hoare logic

- $\dfrac{}{\vdash \{P\} skip \{P\}}$
- $\dfrac{}{\vdash \{Q\{x \to E\}\} x{:=}E \{Q\}}$
- $\dfrac{\vdash \{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\vdash \{P\} S_1 ; S_2 \{Q\}}$
- $\dfrac{\vdash \{P \wedge C\} S_1 \{Q\} \quad \{P \wedge \neg C\} S_2 \{Q\}}{\vdash \{P\} \ \mathit{if} \ C \ \mathit{then} \ S_1 \ \mathit{else} \ S_2 \ \{Q\}}$
- $\dfrac{\vdash \{I \wedge C\} S \{I\}}{\vdash \{I\} \ \mathit{while} \ C \ \mathit{do} \ S \ \{I \wedge \neg C\}}$ I is so-called invariant

Coming up with a suitable invariant is a challenging task (invariant synthesis).

Examples:

Assignment:

$$\{?\}\ x := x{+}3\ \{x < 5\} \ \rightsquigarrow\ \{x{+}3 < 5\}\ x := x{+}3\ \{x < 5\} \ \rightsquigarrow\ \{x < 2\}\ x := x{+}3\ \{x < 5\}$$

Sequence of commands:

$$\frac{\{x+y-1>0\}\ y := y-1\ \{x+y>0\} \qquad \{x+y>0\}\ x := x+y\ \{x>0\}}{\{x+y-1>0\}\ y := y-1; x := x+y\ \{x>0\}}$$

We will learn later how to derive systematically the condition in red with predicate transformers.

Conditional:

$$\frac{\{x \neq 0 \wedge x = 0\}\ y := x\ \{y > 0\} \qquad \{x \neq 0 \wedge \neg(x = 0)\}\ y := -x\ \{y > 0\}}{\{x \neq 0\}\ \texttt{if}\ x = 0\ \texttt{then}\ y := x\ \texttt{else}\ y := -x\ \{y > 0\}}$$

Inference rules for Hoare logic

- $\dfrac{}{\vdash \{P\} skip \{P\}}$
- $\dfrac{}{\vdash \{Q\{x \to E\}\} x{:=}E \{Q\}}$
- $\dfrac{\vdash \{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\vdash \{P\} S_1; S_2 \{Q\}}$
- $\dfrac{\vdash \{P \wedge C\} S_1 \{Q\} \quad \{P \wedge \neg C\} S_2 \{Q\}}{\vdash \{P\}\ if\ \ C\ \ then\ \ S_1\ \ else\ \ S_2\ \ \{Q\}}$
- $\dfrac{\vdash \{I \wedge C\} S \{I\}}{\vdash \{I\}\ while\ \ C\ \ do\ \ S\ \ \{I \wedge \neg C\}}$ $\quad$ I is so-called invariant

Coming up with a suitable invariant is a challenging task (invariant synthesis).

Examples:

Assignment:

$$\{?\}\ x := x{+}3\ \{x < 5\} \ \rightsquigarrow\ \{x{+}3 < 5\}\ x := x{+}3\ \{x < 5\} \ \rightsquigarrow\ \{x < 2\}\ x := x{+}3\ \{x < 5\}$$

Sequence of commands:

$$\dfrac{\{x + y - 1 > 0\}\ y := y - 1\ \{x + y > 0\} \qquad \{x + y > 0\}\ x := x + y\ \{x > 0\}}{\{x + y - 1 > 0\}\ y := y - 1; x := x + y\ \{x > 0\}}$$

We will learn later how to derive systematically the condition in red with predicate transformers.

Conditional:

$$\dfrac{\{x \neq 0 \wedge x = 0\}\ y := x\ \{y > 0\} \qquad \{x \neq 0 \wedge \neg(x = 0)\}\ y := -x\ \{y > 0\}}{\{x \neq 0\}\ \texttt{if}\ \ x = 0\ \ \texttt{then}\ \ y := x\ \ \texttt{else}\ \ y := -x\ \{y > 0\}}$$

Inference rules for Hoare logic

- $\dfrac{}{\vdash \{P\} skip \{P\}}$
- $\dfrac{}{\vdash \{Q\{x \rightarrow E\}\} x{:=}E \{Q\}}$
- $\dfrac{\vdash \{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\vdash \{P\} S_1 ; S_2 \{Q\}}$
- $\dfrac{\vdash \{P \wedge C\} S_1 \{Q\} \quad \{P \wedge \neg C\} S_2 \{Q\}}{\vdash \{P\} \; if \; C \; then \; S_1 \; else \; S_2 \; \{Q\}}$
- $\dfrac{\vdash \{I \wedge C\} S \{I\}}{\vdash \{I\} \; while \; C \; do \; S \; \{I \wedge \neg C\}}$ I is so-called invariant

Coming up with a suitable invariant is a challenging task (invariant synthesis).

Examples:
Assignment:

$$\{?\}\ x := x{+}3\ \{x < 5\} \;\rightsquigarrow\; \{x{+}3 < 5\}\ x := x{+}3\ \{x < 5\} \;\rightsquigarrow\; \{x < 2\}\ x := x{+}3\ \{x < 5\}$$

Sequence of commands:

$$\frac{\{x + y - 1 > 0\}\ y := y - 1\ \{x + y > 0\} \qquad \{x + y > 0\}\ x := x + y\ \{x > 0\}}{\{x + y - 1 > 0\}\ y := y - 1; x := x + y\ \{x > 0\}}$$

We will learn later how to derive systematically the condition in red with predicate transformers.
Conditional:

$$\frac{\{x \neq 0 \wedge x = 0\}\ y := x\ \{y > 0\} \qquad \{x \neq 0 \wedge \neg(x = 0)\}\ y := -x\ \{y > 0\}}{\{x \neq 0\}\ \texttt{if}\ x = 0\ \texttt{then}\ y := x\ \texttt{else}\ y := -x\ \{y > 0\}}$$

Inference rules for Hoare logic

- $\dfrac{}{\vdash \{P\} skip \{P\}}$
- $\dfrac{}{\vdash \{Q\{x \to E\}\} x{:=}E \{Q\}}$
- $\dfrac{\vdash \{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\vdash \{P\} S_1 ; S_2 \{Q\}}$
- $\dfrac{\vdash \{P \wedge C\} S_1 \{Q\} \quad \{P \wedge \neg C\} S_2 \{Q\}}{\vdash \{P\} \textit{ if } C \textit{ then } S_1 \textit{ else } S_2 \ \{Q\}}$
- $\dfrac{\vdash \{I \wedge C\} S \{I\}}{\vdash \{I\} \textit{ while } C \textit{ do } S \ \{I \wedge \neg C\}}$ $\quad I$ is so-called invariant

Coming up with a suitable invariant is a challenging task (invariant synthesis).

Examples:

Assignment:

$$\{?\}\, x := x{+}3 \,\{x < 5\} \;\rightsquigarrow\; \{x{+}3 < 5\}\, x := x{+}3 \,\{x < 5\} \;\rightsquigarrow\; \{x < 2\}\, x := x{+}3 \,\{x < 5\}$$

Sequence of commands:

$$\frac{\{x + y - 1 > 0\}\ y := y - 1\ \{x + y > 0\} \qquad \{x + y > 0\}\ x := x + y\ \{x > 0\}}{\{x + y - 1 > 0\}\ y := y - 1; x := x + y\ \{x > 0\}}$$

We will learn later how to derive systematically the condition in red with predicate transformers.

Conditional:

$$\frac{\{x \neq 0 \wedge x = 0\}\ y := x\ \{y > 0\} \qquad \{x \neq 0 \wedge \neg(x = 0)\}\ y := -x\ \{y > 0\}}{\{x \neq 0\}\ \texttt{if}\ \ x = 0\ \ \texttt{then}\ \ y := x\ \ \texttt{else}\ \ y := -x\ \{y > 0\}}$$

Inference rules for Hoare logic (cont'd)

- Loops: $\frac{\vdash\{I \wedge C\} S \{P\}}{\vdash\{I\} \text{ } while \text{ } C \text{ } do \text{ } S \text{ } \{I \wedge \neg C\}}$
- Loops generalized:

$$\frac{P \Rightarrow I \qquad \{I \wedge C\}\ S\ \{I\} \qquad I \wedge \neg C \Rightarrow Q}{\{P\}\ while\ C\ do\ S\ \{Q\}}$$

Interpretation: To show that, if before the execution of a while-loop the property P holds, after its termination the property Q holds, it suffices to show for some property I (the loop invariant) that I holds before the loop is executed (i.e. that $P \Rightarrow I$), if I holds when the loop body is entered (i.e. if also C holds), that after the execution of the loop body I still holds, when the loop terminates (i.e. if C does not hold), $I \Rightarrow Q$. The challenge is to find appropriate loop invariant I (strongest relationship between all variables modified in loop body).

Example:

$$\frac{(n{=}0 \wedge \ldots) \Rightarrow I \qquad \{I \wedge i \leq n\} s := s+i; i := i+1 \{I\} \qquad (I \wedge \neg(i \leq n)) \Rightarrow s = \sum_{j=1}^{n} j}{\{n{=}0 \wedge i{=}1 \wedge s{=}0\}\texttt{while } i \leq n \texttt{ do } s := s+i; i := i+1 \{s = \sum_{j=1}^{n} j\}}$$

where $I :\Longleftrightarrow\ s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n+1$

Inference rules for Hoare logic (cont'd)

- **Loops**: $\frac{\vdash \{I \wedge C\} S \{P\}}{\vdash \{I\}\ \textit{while}\ C\ \textit{do}\ S\ \{I \wedge \neg C\}}$
- **Loops generalized**:

$$\frac{P \Rightarrow I \qquad \{I \wedge C\}\ S\ \{I\} \qquad I \wedge \neg C \Rightarrow Q}{\{P\}\ \textit{while}\ C\ \textit{do}\ S\ \{Q\}}$$

Interpretation: To show that, if before the execution of a while-loop the property P holds, after its termination the property Q holds, it suffices to show for some property I (the loop invariant) that I holds before the loop is executed (i.e. that $P \Rightarrow I$), if I holds when the loop body is entered (i.e. if also C holds), that after the execution of the loop body I still holds, when the loop terminates (i.e. if C does not hold), $I \Rightarrow Q$. The challenge is to find appropriate loop invariant I (strongest relationship between all variables modified in loop body).

Example:

$$\frac{(n{=}0 \wedge \ldots) \Rightarrow I \qquad \{I \wedge i \leq n\} s := s+i; i := i+1 \{I\} \qquad (I \wedge \neg(i \leq n)) \Rightarrow s = \sum_{j=1}^{n} j}{\{n{=}0 \wedge i{=}1 \wedge s{=}0\} \texttt{while}\ i \leq n\ \texttt{do}\ s := s+i; i := i+1 \{s = \sum_{j=1}^{n} j\}}$$

where $I :\Longleftrightarrow s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n+1$

Inference rules for Hoare logic (cont'd)

- Loops: $\frac{\vdash \{I \wedge C\} S \{P\}}{\vdash \{I\} \ \textit{while} \ C \ \textit{do} \ S \ \{I \wedge \neg C\}}$
- Loops generalized:

$$\frac{P \Rightarrow I \qquad \{I \wedge C\} \ S \ \{I\} \qquad I \wedge \neg C \Rightarrow Q}{\{P\} \ \textit{while} \ C \ \textit{do} \ S \ \{Q\}}$$

Interpretation: To show that, if before the execution of a while-loop the property P holds, after its termination the property Q holds, it suffices to show for some property I (the loop invariant) that I holds before the loop is executed (i.e. that $P \Rightarrow I$), if I holds when the loop body is entered (i.e. if also C holds), that after the execution of the loop body I still holds, when the loop terminates (i.e. if C does not hold), $I \Rightarrow Q$. The challenge is to find appropriate loop invariant I (strongest relationship between all variables modified in loop body).

Example:

$$\frac{(n=0 \wedge \ldots) \Rightarrow I \qquad \{I \wedge i \leq n\} s := s+i; i := i+1 \{I\} \qquad (I \wedge \neg(i \leq n)) \Rightarrow s = \sum_{j=1}^{n} j}{\{n=0 \wedge i=1 \wedge s=0\} \texttt{while } i \leq n \texttt{ do } s := s+i; i := i+1 \{s = \sum_{j=1}^{n} j\}}$$

where $I :\Longleftrightarrow \ s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n+1$

Inference rules for Hoare logic (cont'd)

- Loops: $\frac{\vdash \{I \wedge C\} S \{P\}}{\vdash \{I\} \ \mathit{while}\ C\ \mathit{do}\ S\ \{I \wedge \neg C\}}$
- Loops generalized:

$$\frac{P \Rightarrow I \qquad \{I \wedge C\}\ S\ \{I\} \qquad I \wedge \neg C \Rightarrow Q}{\{P\}\ \mathit{while}\ C\ \mathit{do}\ S\ \{Q\}}$$

Interpretation: To show that, if before the execution of a while-loop the property P holds, after its termination the property Q holds, it suffices to show for some property I (the loop invariant) that I holds before the loop is executed (i.e. that $P \Rightarrow I$), if I holds when the loop body is entered (i.e. if also C holds), that after the execution of the loop body I still holds, when the loop terminates (i.e. if C does not hold), $I \Rightarrow Q$. The challenge is to find appropriate loop invariant I (strongest relationship between all variables modified in loop body).

Example:

$$\frac{(n=0 \wedge \ldots) \Rightarrow I \qquad \{I \wedge i \le n\} s := s+i; i := i+1 \{I\} \qquad (I \wedge \neg(i \le n)) \Rightarrow s = \sum_{j=1}^{n} j}{\{n=0 \wedge i=1 \wedge s=0\} \texttt{while}\ i \le n\ \texttt{do}\ s := s+i; i := i+1 \{s = \sum_{j=1}^{n} j\}}$$

where $I :\Longleftrightarrow\ s = \sum_{j=1}^{i-1} j \wedge 1 \le i \le n+1$

Soundness and relative completeness

- Proof rules for Hoare logic are sound

$$\text{If } \vdash \{P\}S\{Q\} \texttt{ then } \models \{P\}S\{Q\}$$

- Proof rules for Hoare logic are relatively complete

If $\models \{P\}S\{Q\}$ `then` $\vdash \{P\}S\{Q\}$, assuming an oracle for deciding implications

Soundness and relative completeness

- Proof rules for Hoare logic are sound

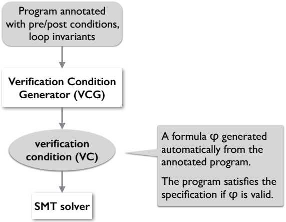
$$\text{If } \vdash \{P\}S\{Q\} \texttt{ then } \models \{P\}S\{Q\}$$

- Proof rules for Hoare logic are relatively complete

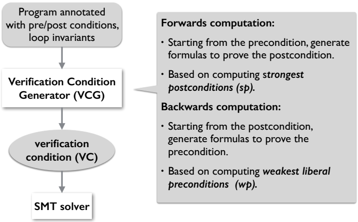
$$\text{If } \models \{P\}S\{Q\} \texttt{ then } \vdash \{P\}S\{Q\}, \text{assuming an oracle for deciding implications}$$

Verification Conditions Generation

Automating Hoare logic with VC generation

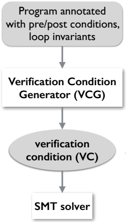


Automating Hoare logic with VC generation



Verification Conditions Generation

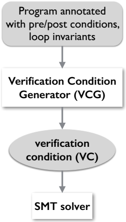
Automating Hoare logic with VC generation



A formula φ generated automatically from the annotated program.

The program satisfies the specification if φ is valid.

Automating Hoare logic with VC generation



Forwards computation:

- Starting from the precondition, generate formulas to prove the postcondition.
- Based on computing ***strongest postconditions (sp).***

Backwards computation:

- Starting from the postcondition, generate formulas to prove the precondition.
- Based on computing ***weakest liberal preconditions (wp).***

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \to E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ S_1\ \ else\ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ S, Q) = X$
  - A fixpoint cannot be expressed as a syntactic construction in terms of the post-condition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ C\ do\ \{I\}\ S, Q) = I$

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \to E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ \ S_1\ \ else\ \ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) = X$
  - A fixpoint cannot be expressed as a syntactic construction in terms of the postcondition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ \ C\ \ do\ \ \{I\}\ S, Q) = I$

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \to E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ \ S_1\ \ else\ \ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \land (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) = X$
  - A fixpoint cannot be expressed as a syntactic construct in the assertion language
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ \ C\ \ do\ \ \{I\}\ S, Q) = I$

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \to E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ \ S_1\ \ else\ \ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) = X$
  - A fixpoint cannot be expressed as a syntactic construction in terms of the postcondition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ \ C\ \ do\ \ \{I\}\ S, Q) = I$

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \rightarrow E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ \ S_1\ \ else\ \ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) = X$
  - A fixpoint cannot be expressed as a syntactic construction in terms of the postcondition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ C\ do\ \{I\}\ S, Q) = I$

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \to E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ S_1\ \ else\ \ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) = ✗$
  - A fixpoint cannot be expressed as a syntactic construction in terms of the postcondition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ \ C\ \ do\ \{I\}\ S, Q) = I$

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \to E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ \ S_1\ \ else\ \ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) = ✗$
  - A fixpoint cannot be expressed as a syntactic construction in terms of the postcondition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ \ C\ \ do\ \{I\}\ S, Q) = I$

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \to E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ S_1\ \ else\ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) = ✗$
  - A fixpoint cannot be expressed as a syntactic construction in terms of the postcondition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ \ C\ \ do\ \{I\}\ S, Q) = I$
    - I is a loop invariant provided by an oracle (e.g., programmer)
    - For each statement S, also define $VC(S, Q)$ that encodes additional conditions that must be checked

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \to E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ S_1\ \ else\ \ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) = ✗$
  - A fixpoint cannot be expressed as a syntactic construction in terms of the postcondition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ \ C\ \ do\ \ \{I\}\ \ S, Q) = I$
    - I is a loop invariant provided by an oracle (e.g., programmer)
    - For each statement S, also define $VC(S, Q)$ that encodes additional conditions that must be checked

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \to E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ S_1\ \ else\ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) = ✗$
  - A fixpoint cannot be expressed as a syntactic construction in terms of the postcondition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ \ C\ \ do\ \{I\}\ S, Q) = I$
    - I is a loop invariant provided by an oracle (i.e., programmer)
    - For each statement S, also define $VC(S, Q)$ that encodes additional conditions that must be checked

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \rightarrow E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ \ S_1\ \ else\ \ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) = ✗$
  - A fixpoint cannot be expressed as a syntactic construction in terms of the postcondition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ \ C\ \ do\ \ \{I\}\ \ S, Q) = I$
    - I is a loop invariant provided by an oracle (e.g. programmer)
    - For each statement S, also define $VC(S, Q)$ that encodes additional conditions that must be checked.

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \rightarrow E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ S_1\ \ else\ \ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) = ✗$
  - A fixpoint cannot be expressed as a syntactic construction in terms of the postcondition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ \ C\ \ do\ \{I\}\ S, Q) = I$
    - I is a loop invariant provided by an oracle (e.g. programmer)
    - For each statement S, also define $VC(S, Q)$ that encodes additional conditions that must be checked.

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \to E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ \ S_1\ \ else\ \ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) = ✗$
  - A fixpoint cannot be expressed as a syntactic construction in terms of the postcondition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ \ C\ \ do\ \ \{I\}\ \ S, Q) = I$
    - I is a loop invariant provided by an oracle (e.g. programmer)
    - For each statement S, also define $VC(S, Q)$ that encodes additional conditions that must be checked.

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \to E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ S_1\ \ else\ \ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) =$ ✗
  - A fixpoint cannot be expressed as a syntactic construction in terms of the postcondition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ \ C\ \ do\ \{I\}\ S, Q) = I$
    - I is a loop invariant provided by an oracle (e.g. programmer)
    - For each statement S, also define $VC(S, Q)$ that encodes additional conditions that must be checked.

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \rightarrow E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ \ S_1\ \ else\ \ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) = ✗$
  - A fixpoint cannot be expressed as a syntactic construction in terms of the postcondition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ \ C\ \ do\ \{I\}\ S, Q) = I$
    - I is a loop invariant provided by an oracle (e.g. programmer)
    - For each statement S, also define $VC(S, Q)$ that encodes additional conditions that must be checked.

VC generation with *wp* and *sp*

- $wp(S, Q)$: The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.
- $sp(S, P)$: The strongest predicate that holds after S is executed from a state satisfying P.
- $\{P\}\ S\ \{Q\}$ is valid iff
  - $P \Rightarrow wp(S, Q)$
  - $sp(S, P) \Rightarrow Q$

Computing $wp(S, Q)$

$wp(S, Q)$:

- $wp(skip, Q) = Q$
- $wp(x := E, Q) = Q[x \rightarrow E]$
- $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
- $wp(if\ \ C\ \ then\ \ S_1\ \ else\ \ S_2, Q) = (C \Rightarrow wp(S_1, Q)) \wedge (\neg C \Rightarrow wp(S_2, Q))$
- $wp(while\ \ C\ \ do\ \ S, Q) =$ ✗
  - A fixpoint cannot be expressed as a syntactic construction in terms of the postcondition
  - Approximate $wp(S, Q)$ with $awp(S, Q)$
  - How $awp(S, Q)$ will look like?
  - $awp(while\ \ C\ \ do\ \{I\}\ S, Q) = I$
    - I is a loop invariant provided by an oracle (e.g. programmer)
    - For each statement S, also define $VC(S, Q)$ that encodes additional conditions that must be checked.

Computing $VC(S, Q)$. Verifying a Hoare triple

Computing $VC(S, Q)$

$VC(S, Q)$:

- $VC(skip, Q) = true$
- $VC(x := E, Q) = true$
  Note that E may contain division so the condition of non-zero denominator must be imposed!
- $VC(S1; S2, Q) = VC(S_2, Q) \wedge VC(S_1, awp(S_2, Q))$
- $VC(if\ \ C\ \ then\ S_1\ else\ S_2, Q) = VC(S_1, Q) \wedge VC(S_2, Q)$
- $VC(while\ \ C\ \ do\ \{I\}\ S, Q) = \underbrace{(I \wedge C \Rightarrow awp(S, I) \wedge VC(S, I))}_{I \text{ is an invariant}} \wedge \underbrace{(I \wedge \neg C \Rightarrow Q)}_{I \text{ is strong enough}}$

Theorem

$\{P\}S\{Q\}$ *is valid if* $VC(S, Q) \wedge P \Rightarrow awp(S, Q)$

Direction $\Leftarrow$ doesn't hold because loop invariants may not be strong enough or they may be incorrect. Might get false alarms.

Computing $VC(S, Q)$. Verifying a Hoare triple

Computing $VC(S, Q)$

$VC(S, Q)$:

- $VC(skip, Q) = true$
- $VC(x := E, Q) = true$
  Note that E may contain division so the condition of non-zero denominator must be imposed!
- $VC(S1; S2, Q) = VC(S_2, Q) \wedge VC(S_1, awp(S_2, Q))$
- $VC(if\ \ C\ \ then\ S_1\ else\ S_2, Q) = VC(S_1, Q) \wedge VC(S_2, Q)$
- $VC(while\ \ C\ \ do\ \{I\}\ S, Q) = \underbrace{(I \wedge C \Rightarrow awp(S, I) \wedge VC(S, I))}_{I \text{ is an invariant}} \wedge \underbrace{(I \wedge \neg C \Rightarrow Q)}_{I \text{ is strong enough}}$

Theorem

$\{P\}S\{Q\}$ *is valid if* $VC(S, Q) \wedge P \Rightarrow awp(S, Q)$

Direction $\Leftarrow$ doesn't hold because loop invariants may not be strong enough or they may be incorrect. Might get false alarms.

Computing $VC(S, Q)$. Verifying a Hoare triple

Computing $VC(S, Q)$

$VC(S, Q)$:

- $VC(skip, Q) = true$
- $VC(x := E, Q) = true$
  Note that E may contain division so the condition of non-zero denominator must be imposed!
- $VC(S1; S2, Q) = VC(S_2, Q) \wedge VC(S_1, awp(S_2, Q))$
- $VC(if\ \ C\ \ then\ S_1\ else\ S_2, Q) = VC(S_1, Q) \wedge VC(S_2, Q)$
- $VC(while\ \ C\ \ do\ \{I\}\ S, Q) = \underbrace{(I \wedge C \Rightarrow awp(S, I) \wedge VC(S, I))}_{I \text{ is an invariant}} \wedge \underbrace{(I \wedge \neg C \Rightarrow Q)}_{I \text{ is strong enough}}$

Theorem

$\{P\}S\{Q\}$ is valid if $VC(S, Q) \wedge P \Rightarrow awp(S, Q)$

Direction $\Leftarrow$ doesn't hold because loop invariants may not be strong enough or they may be incorrect. Might get false alarms.

Computing $VC(S, Q)$. Verifying a Hoare triple

Computing $VC(S, Q)$

$VC(S, Q)$:

- $VC(skip, Q) = true$
- $VC(x := E, Q) = true$
  Note that E may contain division so the condition of non-zero denominator must be imposed!
- $VC(S1; S2, Q) = VC(S_2, Q) \wedge VC(S_1, awp(S_2, Q))$
- $VC(if\ \ C\ \ then\ S_1\ else\ S_2, Q) = VC(S_1, Q) \wedge VC(S_2, Q)$
- $VC(while\ \ C\ \ do\ \{I\}\ S, Q) = \underbrace{(I \wedge C \Rightarrow awp(S, I) \wedge VC(S, I))}_{I \text{ is an invariant}} \wedge \underbrace{(I \wedge \neg C \Rightarrow Q)}_{I \text{ is strong enough}}$

Theorem

$\{P\}S\{Q\}$ is valid if $VC(S, Q) \wedge P \Rightarrow awp(S, Q)$

Direction $\Leftarrow$ doesn't hold because loop invariants may not be strong enough or they may be incorrect. Might get false alarms.

Computing $VC(S,Q)$. Verifying a Hoare triple

Computing $VC(S,Q)$

$VC(S,Q)$:

- $VC(skip, Q) = true$
- $VC(x := E, Q) = true$
  Note that E may contain division so the condition of non-zero denominator must be imposed!
- $VC(S1; S2, Q) = VC(S_2, Q) \wedge VC(S_1, awp(S_2, Q))$
- $VC(if\ \ C\ \ then\ S_1\ else\ S_2, Q) = VC(S_1, Q) \wedge VC(S_2, Q)$
- $VC(while\ \ C\ \ do\ \{I\}\ S, Q) = \underbrace{(I \wedge C \Rightarrow awp(S, I) \wedge VC(S, I))}_{I \text{ is an invariant}} \wedge \underbrace{(I \wedge \neg C \Rightarrow Q)}_{I \text{ is strong enough}}$

Theorem

$\{P\}S\{Q\}$ *is valid if* $VC(S,Q) \wedge P \Rightarrow awp(S,Q)$

Direction $\Leftarrow$ doesn't hold because loop invariants may not be strong enough or they may be incorrect. Might get false alarms.

Computing $VC(S, Q)$. Verifying a Hoare triple

Computing $VC(S, Q)$

$VC(S, Q)$:

- $VC(skip, Q) = true$
- $VC(x := E, Q) = true$
  Note that E may contain division so the condition of non-zero denominator must be imposed!
- $VC(S1; S2, Q) = VC(S_2, Q) \wedge VC(S_1, awp(S_2, Q))$
- $VC(if\ \ C\ \ then\ \ S_1\ \ else\ \ S_2, Q) = VC(S_1, Q) \wedge VC(S_2, Q)$
- $VC(while\ \ C\ \ do\ \{I\}\ S, Q) = \underbrace{(I \wedge C \Rightarrow awp(S, I) \wedge VC(S, I))}_{I \text{ is an invariant}} \wedge \underbrace{(I \wedge \neg C \Rightarrow Q)}_{I \text{ is strong enough}}$

Theorem

$\{P\}S\{Q\}$ *is valid if* $VC(S, Q) \wedge P \Rightarrow awp(S, Q)$

Direction $\Leftarrow$ doesn't hold because loop invariants may not be strong enough or they may be incorrect. Might get false alarms.

Computing $VC(S, Q)$. Verifying a Hoare triple

Computing $VC(S, Q)$

$VC(S, Q)$:

- $VC(skip, Q) = true$
- $VC(x := E, Q) = true$
  Note that E may contain division so the condition of non-zero denominator must be imposed!
- $VC(S1; S2, Q) = VC(S_2, Q) \wedge VC(S_1, awp(S_2, Q))$
- $VC(if\ \ C\ \ then\ \ S_1\ \ else\ \ S_2, Q) = VC(S_1, Q) \wedge VC(S_2, Q)$
- $VC(while\ \ C\ \ do\ \{I\}\ S, Q) = \underbrace{(I \wedge C \Rightarrow awp(S, I) \wedge VC(S, I))}_{I \text{ is an invariant}} \wedge \underbrace{(I \wedge \neg C \Rightarrow Q)}_{I \text{ is strong enough}}$

Theorem

$\{P\}S\{Q\}$ *is valid if* $VC(S, Q) \wedge P \Rightarrow awp(S, Q)$

Direction $\Leftarrow$ doesn't hold because loop invariants may not be strong enough or they may be incorrect. Might get false alarms.

Computing $VC(S, Q)$. Verifying a Hoare triple

Computing $VC(S, Q)$

$VC(S, Q)$:

- $VC(skip, Q) = true$
- $VC(x := E, Q) = true$
  Note that E may contain division so the condition of non-zero denominator must be imposed!
- $VC(S1; S2, Q) = VC(S_2, Q) \wedge VC(S_1, awp(S_2, Q))$
- $VC(if\ \ C\ \ then\ \ S_1\ \ else\ \ S_2, Q) = VC(S_1, Q) \wedge VC(S_2, Q)$
- $VC(while\ \ C\ \ do\ \{I\}\ S, Q) = \underbrace{(I \wedge C \Rightarrow awp(S, I) \wedge VC(S, I))}_{I \text{ is an invariant}} \wedge \underbrace{(I \wedge \neg C \Rightarrow Q)}_{I \text{ is strong enough}}$

Theorem

$\{P\}S\{Q\}$ *is valid if* $VC(S, Q) \wedge P \Rightarrow awp(S, Q)$

Direction $\Leftarrow$ doesn't hold because loop invariants may not be strong enough or they may be incorrect. Might get false alarms.

Termination

Loops and recursive calls may lead to non-termination.

- Loops:

$$\frac{I \Rightarrow t \geq 0 \qquad \{I \wedge C \wedge t = N\}\ S\ \{I \wedge t < N\}}{\{I\}\ \mathit{while}\ C\ \mathit{do}\ S\ \{I \wedge \neg C\}}$$

- Loops generalized:

$$\frac{P \Rightarrow I \qquad I \Rightarrow t \geq 0 \qquad \{I \wedge C \wedge t = N\}\ S\ \{I \wedge t < N\} \qquad I \wedge \neg C \Rightarrow Q}{\{P\}\ \mathit{while}\ C\ \mathit{do}\ S\ \{Q\}}$$

Interpretation: If execution of S starts in a state where P holds, then execution terminates in a state where Q holds, unless it aborts. t is called termination term (must denote a natural number). It becomes smaller by every iteration of the loop, but it does not become negative. Consequently, the loop must eventually terminate. The initial value of t limits the number of loop iterations. Example: Let $t = n - i + 1$. We have:

$$\frac{(n{=}0 \wedge \ldots) \Rightarrow I \qquad I \Rightarrow t > 0 \qquad \{I \wedge i \leq n \wedge t < N\}\, s{:=}\ldots;\, i{:=}\ldots \{I \wedge t{=}N\} \qquad (I \wedge \neg(i \leq n)) \Rightarrow s{=}\sum_{j=1}^{n} j}{\{n{=}0 \wedge i{=}1 \wedge s{=}0\}\texttt{while}\ i \leq n\ \texttt{do}\ s{:=}s{+}i;\, i{:=}i{+}1\{s{=}\sum_{j=1}^{n} j\}}$$

where $I :\Longleftrightarrow\ s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n+1$.

Coming up with a termination term is a challenging task and not always possible (see halting problem). That is, we can not decide termination for all possible program-input pairs. In practice, the aim is to find the answer "program does terminate" (or "program does not terminate") whenever this is possible.

Termination

Loops and recursive calls may lead to non-termination.

- Loops:

$$\frac{I \Rightarrow t \geq 0 \qquad \{I \wedge C \wedge t = N\}\ S\ \{I \wedge t < N\}}{\{I\}\ \mathit{while}\ C\ \mathit{do}\ S\ \{I \wedge \neg C\}}$$

- Loops generalized:

$$\frac{P \Rightarrow I \qquad I \Rightarrow t \geq 0 \qquad \{I \wedge C \wedge t = N\}\ S\ \{I \wedge t < N\} \qquad I \wedge \neg C \Rightarrow Q}{\{P\}\ \mathit{while}\ C\ \mathit{do}\ S\ \{Q\}}$$

Interpretation: If execution of S starts in a state where P holds, then execution terminates in a state where Q holds, unless it aborts. t is called termination term (must denote a natural number). It becomes smaller by every iteration of the loop, but it does not become negative. Consequently, the loop must eventually terminate. The initial value of t limits the number of loop iterations. Example: Let $t = n - i + 1$. We have:

$$\frac{(n{=}0\wedge\ldots)\Rightarrow I \qquad I\Rightarrow t>0 \qquad \{I\wedge i\leq n\wedge t<N\}s{:=}\ldots;\ i{:=}\ldots\{I\wedge t{=}N\} \qquad (I\wedge\neg(i\leq n))\Rightarrow s{=}\sum_{j=1}^{n} j}{\{n=0\wedge i=1\wedge s=0\}\texttt{while}\ i\leq n\ \texttt{do}\ s:=s+i; i:=i+1\{s=\sum_{j=1}^{n} j\}}$$

where $I :\Longleftrightarrow\ s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n+1$.

Coming up with a termination term is a challenging task and not always possible (see halting problem). That is, we can not decide termination for all possible program-input pairs. In practice, the aim is to find the answer "program does terminate" (or "program does not terminate") whenever this is possible.

Termination

Loops and recursive calls may lead to non-termination.

- Loops:

$$\frac{I \Rightarrow t \geq 0 \qquad \{I \wedge C \wedge t = N\}\ S\ \{I \wedge t < N\}}{\{I\}\ \textit{while}\ C\ \textit{do}\ S\ \{I \wedge \neg C\}}$$

- Loops generalized:

$$\frac{P \Rightarrow I \qquad I \Rightarrow t \geq 0 \qquad \{I \wedge C \wedge t = N\}\ S\ \{I \wedge t < N\} \qquad I \wedge \neg C \Rightarrow Q}{\{P\}\ \textit{while}\ C\ \textit{do}\ S\ \{Q\}}$$

Interpretation: If execution of S starts in a state where P holds, then execution terminates in a state where Q holds, unless it aborts. t is called termination term (must denote a natural number). It becomes smaller by every iteration of the loop, but it does not become negative. Consequently, the loop must eventually terminate. The initial value of t limits the number of loop iterations. Example: Let $t = n - i + 1$. We have:

$$\frac{(n{=}0 \wedge \ldots) \Rightarrow I \qquad I \Rightarrow t > 0 \qquad \{I \wedge i \leq n \wedge t < N\} s{:=}\ldots; i{:=}\ldots \{I \wedge t{=}N\} \qquad (I \wedge \neg(i \leq n)) \Rightarrow s{=}\sum_{j=1}^{n} j}{\{n{=}0 \wedge i{=}1 \wedge s{=}0\}\texttt{while}\ i \leq n\ \texttt{do}\ s{:=}s{+}i; i{:=}i{+}1\{s{=}\sum_{j=1}^{n} j\}}$$

where $I :\iff s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n+1$.

Coming up with a termination term is a challenging task and not always possible (see halting problem). That is, we can not decide termination for all possible program-input pairs. In practice, the aim is to find the answer "program does terminate" (or "program does not terminate") whenever this is possible.

Termination

Loops and recursive calls may lead to non-termination.

- Loops:

$$\frac{I \Rightarrow t \geq 0 \qquad \{I \wedge C \wedge t = N\}\ S\ \{I \wedge t < N\}}{\{I\}\ \mathit{while}\ C\ \mathit{do}\ S\ \{I \wedge \neg C\}}$$

- Loops generalized:

$$\frac{P \Rightarrow I \qquad I \Rightarrow t \geq 0 \qquad \{I \wedge C \wedge t = N\}\ S\ \{I \wedge t < N\} \qquad I \wedge \neg C \Rightarrow Q}{\{P\}\ \mathit{while}\ C\ \mathit{do}\ S\ \{Q\}}$$

Interpretation: If execution of S starts in a state where P holds, then execution terminates in a state where Q holds, unless it aborts. t is called termination term (must denote a natural number). It becomes smaller by every iteration of the loop, but it does not become negative. Consequently, the loop must eventually terminate. The initial value of t limits the number of loop iterations. Example: Let $t = n - i + 1$. We have:

$$\frac{(n{=}0 \wedge \ldots) \Rightarrow I \qquad I \Rightarrow t > 0 \qquad \{I \wedge i \leq n \wedge t < N\} s := \ldots; i := \ldots \{I \wedge t = N\} \qquad (I \wedge \neg(i \leq n)) \Rightarrow s = \sum_{j=1}^{n} j}{\{n{=}0 \wedge i{=}1 \wedge s{=}0\}\texttt{while}\ i \leq n\ \texttt{do}\ s := s+i; i := i+1 \{s = \sum_{j=1}^{n} j\}}$$

where $I :\iff s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n+1$.

Coming up with a termination term is a challenging task and not always possible (see halting problem). That is, we can not decide termination for all possible program-input pairs. In practice, the aim is to find the answer “program does terminate” (or “program does not terminate”) whenever this is possible.

Termination

Loops and recursive calls may lead to non-termination.

- Loops:

$$\frac{I \Rightarrow t \geq 0 \qquad \{I \wedge C \wedge t = N\}\ S\ \{I \wedge t < N\}}{\{I\}\ \mathit{while}\ C\ \mathit{do}\ S\ \{I \wedge \neg C\}}$$

- Loops generalized:

$$\frac{P \Rightarrow I \qquad I \Rightarrow t \geq 0 \qquad \{I \wedge C \wedge t = N\}\ S\ \{I \wedge t < N\} \qquad I \wedge \neg C \Rightarrow Q}{\{P\}\ \mathit{while}\ C\ \mathit{do}\ S\ \{Q\}}$$

Interpretation: If execution of S starts in a state where P holds, then execution terminates in a state where Q holds, unless it aborts. t is called termination term (must denote a natural number). It becomes smaller by every iteration of the loop, but it does not become negative. Consequently, the loop must eventually terminate. The initial value of t limits the number of loop iterations. Example: Let $t = n - i + 1$. We have:

$$\frac{(n{=}0\wedge\ldots)\Rightarrow I \qquad I\Rightarrow t>0 \qquad \{I\wedge i\leq n\wedge t<N\}s{:=}\ldots;\, i{:=}\ldots\{I\wedge t{=}N\} \qquad (I\wedge\neg(i\leq n))\Rightarrow s{=}\sum_{j=1}^{n} j}{\{n{=}0\wedge i{=}1\wedge s{=}0\}\texttt{while}\ i\leq n\ \texttt{do}\ s{:=}s{+}i;\, i{:=}i{+}1\{s{=}\sum_{j=1}^{n} j\}}$$

where $I :\iff\ s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n+1$.

Coming up with a termination term is a challenging task and not always possible (see halting problem). That is, we can not decide termination for all possible program-input pairs. In practice, the aim is to find the answer “program does terminate” (or “program does not terminate”) whenever this is possible.

Outline

Examples

see `ResoningAboutPrograms1-Examples.pdf`

References