Program Correctness: Mechanics

CS410  
Fall 2017
New Reference

The Calculus of Computation
Decision Procedures with Applications to Verification
Overview

Goal: specifying and proving properties of programs.

Model: Control Flow Graph, or the program itself.

Specifications: First Order Logic (FOL) formulas.

Proof Methods: Inductive Assertion Method, and Ranking Functions.
A Simple Language

5.1 pi: A Simple Imperative Language

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;
}
A Simple Language

5.1 \pi: A Simple Imperative Language

Fig. 5.1. LinearSearch

```c
@pre T
@post T
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;
}
```

The language \pi is superficially a C-like language with restrictions, we present the essential features of \pi through examples.

**Example 5.1.** Figure 5.1 lists the function LinearSearch, which searches the range \([ℓ, u]\) of a range \([ℓ, u]\) array \(a\) of integers for a value \(e\). It returns true if the given array contains the value between the lower bound \(ℓ\) and upper bound \(u\). It behaves correctly only if \(0 ≤ ℓ\) and \(u < |a|\); otherwise, the array \(a\) is accessed outside of its domain \([0, |a| − 1]\). \(|a|\) denotes the length of array \(a\).

Notice the lines beginning with @. They are program annotations, which we discuss in detail in the next section.

**Example 5.2.** Figure 5.2 lists the recursive function BinarySearch, which searches a range \([ℓ, u]\) of a sorted (weakly increasing: \(a[i] ≤ a[j]\) if \(i ≤ j\)) array \(a\) of integers for a value \(e\). Like LinearSearch, it returns true if the...
A Simple Language

@pre T
@post T
bool LinearSearch(int[] a, int ℓ, int u, int e) {
    for @ T
        (int i := ℓ; i ≤ u; i := i + 1) {
            if (a[i] = e) return true;
        }
    return false;
}
A Simple Language

5.1. The Language

Because \( \pi \) is superficially a C-like language with restrictions, we present the essential features of \( \pi \) through examples.

Example 5.1. Figure 5.1 lists the function \( \text{LinearSearch} \), which searches the range \([\ell, u]\) of a array \( a \) of integers for a value \( e \). It returns true if the given array contains the value between the lower bound \( \ell \) and upper bound \( u \). It behaves correctly only if \( 0 \leq \ell \) and \( u < |a| \); otherwise, the array \( a \) is accessed outside of its domain \([0, |a| - 1]\). \( |a| \) denotes the length of array \( a \).

Observe that most of the syntax is similar to C. For example, the for loop sets \( i \) to be \( \ell \) initially and then executes the body of the loop and increments \( i \) by 1 as long as \( i \leq u \). Also, an integer array has type \( \text{int}[] \), which is constructed from base type \( \text{int} \). One syntactic difference occurs in assignment, which is written := to distinguish it from the equality predicate \( = \). We use \( = \) as the equality predicate, rather than \( == \), to correspond to the standard equality predicate of FOL. Finally, unlike C, \( \pi \) has type \( \text{bool} \) and constants \( \text{true} \) and \( \text{false} \).

Notice the lines beginning with @. They are program annotations, which we discuss in detail in the next section.

Example 5.2. Figure 5.2 lists the recursive function \( \text{BinarySearch} \), which searches a range \([\ell, u]\) of a sorted (weakly increasing: \( a[i] \leq a[j] \) if \( i \leq j \)) array \( a \) of integers for a value \( e \). Like \( \text{LinearSearch} \), it returns true if the given array contains the value between the lower bound \( \ell \) and upper bound \( u \).

Annotations
An annotation is a First Order Logic formula \( F \) whose free variables only include program variables.

An annotation \( F \) at program location \( L \) means that \( F \) holds whenever program control reaches \( L \).

@pre \( \top \)
@post \( \top \)

bool LinearSearch(int[] a, int \( \ell \), int \( u \), int \( e \)) {
    for @ \( \top \)
        (int \( i \) := \( \ell \); \( i \) \( \leq \) \( u \); \( i \) := \( i \) + 1) {
            if (a[\( i \)] = \( e \)) return true;
        }
    @ \( i \) = \( u \)
    return false;
}

Example 5.1.
Figure 5.1 lists the function LinearSearch, which searches the range \([\ell, u]\) of a numeric array \( a \) of integers for a value \( e \). It returns \( true \) if the given array contains the value between the lower bound \( \ell \) and upper bound \( u \). It behaves correctly only if \( 0 \leq \ell \) and \( u < |a| \); otherwise, the array \( a \) is accessed outside of its domain \([0, |a| - 1]\). |\( a | denotes the length of array \( a \).

Observe that most of the syntax is similar to C. For example, the for loop sets \( i \) to be \( \ell \) initially and then executes the body of the loop and increments \( i \) by 1 as long as \( i \leq u \). Also, an integer array has type \( \text{int}[] \), which is constructed from base type \( \text{int} \). One syntactic difference occurs in assignment, which is written := to distinguish it from the equality predicate =. We use = as the equality predicate, rather than ==, to correspond to the standard equality predicate of FOL. Finally, unlike C, \( \pi \) has type \( \text{bool} \) and constants \( true \) and \( false \). Notice the lines beginning with @. They are program annotations, which we discuss in detail in the next section.

Example 5.2.
Figure 5.2 lists the recursive function BinarySearch, which searches a range \([\ell, u]\) of a sorted (weakly increasing: \( a[\( i \)] \leq a[\( j \)] \) if \( i \leq j \)) array \( a \) of integers for a value \( e \). Like LinearSearch, it returns \( true \) if the
• **Precondition** indicates what is true upon entering the function. Free variables only include function parameters.

• **Postcondition:** indicates what is true upon exiting the function. Free variables only include function parameters and a special variable, \( rv \), that refers to the return value.

```cpp
@pre
@post
bool LinearSearch(int[] a, int \ell, int u, int e) {
    for (int i := \ell; i \leq u; i := i + 1) {
        if (a[i] = e) return true;
    }
    return false;
}
```
• **Precondition** indicates what is true upon **entering** the function. Free variables only include **function parameters**.

• **Postcondition**: indicates what is true upon **exiting** the function. Free variables only include **function parameters** and a special variable, `rv`, that refers to the return value.

```c
@pre 0 \leq \ell \land u < |a|
@post
bool LinearSearch(int[] a, int \ell, int u, int e) {
    for (int i := \ell; i \leq u; i := i + 1) {
        if (a[i] = e) return true;
    }
    return false;
}
```
• **Precondition** indicates what is true upon entering the function. Free variables only include **function parameters**.

• **Postcondition**: indicates what is true upon exiting the function. Free variables only include **function parameters** and a special variable, \( rv \), that refers to the **return value**.

```
@pre 0 \leq \ell \land u < |a|
@post rv \leftrightarrow \exists i. \ell \leq i \leq u \land a[i] = e

bool LinearSearch(int[] a, int \ell, int u, int e) {
    for (int i := \ell; i \leq u; i := i + 1) {
        if (a[i] = e) return true;
    }
    return false;
}
```
Another Example

```c
@pre
@post
bool LinearSearch(int[] a, int ℓ, int u, int e) {
    if (ℓ < 0 ∨ u ≥ |a|) return false;
    for @ ⊤
        (int i := ℓ; i ≤ u; i := i + 1) {
            if (a[i] = e) return true;
        }
    return false;
}
```

Example 5.5. In Example 5.1, we informally specified the behavior of `LinearSearch` as follows: `LinearSearch` returns `true` if the array `a` contains the value `e` in the range `[ℓ, u]`. It behaves correctly only when `ℓ ≥ 0` and `u < |a|`.

Function specifications formalize such statements. Figure 5.6 presents `LinearSearch` with its specification. The precondition asserts that the lower bound `ℓ` should be at least `0` and that the upper bound `u` should be less than the length `|a|` of the array `a`. The postcondition asserts that the return value `rv` is `true` if `a[i] = e` for some index `i` ∈ [ℓ, u] of `a`.

Example 5.6. A nontrivial precondition (a formula other than ⊤) is not always acceptable, especially if a function is public to a module. Figure 5.7 lists a more robust version of linear search. The formula 0 ≤ ℓ ≤ i ≤ u < |a| abbreviates 0 ≤ ℓ ∧ ℓ ≤ i ∧ i ≤ u ∧ u < |a|.

A nontrivial precondition is sometimes acceptable for a function that is private to a module. The verification method of this chapter checks that every instance of a call to such a function obeys the precondition.
Another Example

```java
@pre T
@post
bool LinearSearch(int[] a, int ℓ, int u, int e) {
    if (ℓ < 0 ∨ u ≥ |a|) return false;
    for @ T
        (int i := ℓ; i ≤ u; i := i + 1) {
            if (a[i] = e) return true;
        }
    return false;
}
```

Another Example

@pre T
@post rv ↔ ∃i. 0 ≤ ℓ ≤ i ≤ u < |a| ∧ a[i] = e
bool LinearSearch(int[] a, int ℓ, int u, int e) {
    if (ℓ < 0 ∨ u ≥ |a|) return false;
    for @ T
        (int i := ℓ; i ≤ u; i := i + 1) {
            if (a[i] = e) return true;
        }
    return false;
}
Loop Invariant

- **Loop Invariant** holds at the beginning of each iteration.

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

For example, the returned array `rv` should be a permutation of the original array `a` (see Exercise 6.5).
• Loop Invariant holds at the beginning of each iteration.

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

\[ F \land ⟨condition⟩ \]
• **Loop Invariant** holds at the beginning of each iteration.

\[
\begin{align*}
F & \land \langle \text{condition} \rangle \\
F & \land \neg \langle \text{condition} \rangle
\end{align*}
\]

```
while @ F 
(\langle \text{condition} \rangle) \{
\langle \text{body} \rangle
\}
```
**Loop Invariant**

- **Loop Invariant** holds at the beginning of each iteration.

\[
\begin{align*}
F & \land \langle \text{condition} \rangle \\
F & \land \neg \langle \text{condition} \rangle
\end{align*}
\]

for
\[
\begin{align*}
@ F \\
\langle \text{initialize} \rangle; \langle \text{condition} \rangle; \langle \text{increment} \rangle & \{ \\
\langle \text{body} \rangle & \}
\end{align*}
\]

\[
\begin{align*}
\text{while} & \\
@ F \\
\langle \text{condition} \rangle & \{ \\
\langle \text{body} \rangle & \}
\end{align*}
\]
Loop Invariant

- **Loop Invariant** holds at the beginning of each iteration.

\[
\begin{align*}
F \land \langle \text{condition} \rangle \\
F \land \neg\langle \text{condition} \rangle
\end{align*}
\]

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

```plaintext
@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:
            (int i := ℓ; i ≤ u; i := i + 1) {
                if (a[i] = e) return true;
            }
    return false;
}
```
Example

@pre 0 ≤ ℓ ∧ u < |a|
@post rv ↔ ∃i. ℓ ≤ i ≤ u ∧ a[i] = e

bool LinearSearch(int[] a, int ℓ, int u, int e) {
    for
        @L: ℓ ≤ i ∧ (∀j. ℓ ≤ j < i → a[j] ≠ e)
        (int i := ℓ; i ≤ u; i := i + 1) {
            if (a[i] = e) return true;
        }
    return false;
}
Assertions

We can add annotations **anywhere** in the program.

- **Assertions**: when they are not **preconditions**, postconditions, or loop invariants, they are simply called assertions.
Assertions

We can add annotations anywhere in the program.

- **Assertions**: when they are not preconditions, postconditions, or loop invariants, they are simply called assertions.

```plaintext
@ k > 0;
i := i + k;
```
Partial Correctness
Overview

A function is **partially correct** if when the function’s precondition is satisfied on entry, its postcondition is satisfied when it returns (if it ever does).
Some Definitions

- **Program States**: an assignment of values (of the proper type) to program variables.

```plaintext
@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;
}
```
Some Definitions

- **Program States**: an assignment of values (of the proper type) to program variables.

\[
s : \{\text{pc} \leftarrow L_1, l \leftarrow 1, u \leftarrow 3, i \leftarrow 3, a \leftarrow [4; 7; 1], rv \leftarrow []\}
\]

```plaintext
@pre 0 \leq l \land u < |a| 
@post rv \leftarrow \exists i. l \leq i \leq u \land a[i] = e 
bool LinearSearch(int[] a, int l, int u, int e) {
  for
    @L: l \leq i \land (\forall j. l \leq j < i \rightarrow a[j] \neq e)
    (int i := l; i \leq u; i := i + 1) {
      if (a[i] = e) return true;
    }
  return false;
}
```
Some Definitions

- **Program States:** an assignment of values (of the proper type) to program variables.

\[ s : \{ pc \leftarrow L_1, l \leftarrow 1, u \leftarrow 3, i \leftarrow 3, a \leftarrow [4; 7; 1], rv \leftarrow [] \} \]

The state can also be represented by any logical formula in any theory.
Partial Correctness

- Given pre/post conditions

\[ F_{pre}, F_{post} \]

\[ s_0[p_{c}] = L_0 \]

\[ s_0 \models F_{pre} \]
Partial Correctness

- Given pre/post conditions

  \[ F_{pre}, F_{post} \]

  \[ s_0[pc] = L_0 \]

  \[ s_0 \models F_{pre} \]

- The function may have both finite and infinite paths:

  \[ s_0 s_1 s_2 \ldots s_n \]

  \[ s_0 s_1 s_2 \ldots s_n \ldots \]
Partial Correctness

- Given pre/post conditions $F_{\text{pre}}, F_{\text{post}}$

$$s_0[pc] = L_0$$

$$s_0 \models F_{\text{pre}}$$

- The function may have both finite and infinite paths:

$$s_0 s_1 s_2 \ldots s_n$$

$$s_0 s_1 s_2 \ldots s_n \ldots$$

- The function is partially correct if for every finite path:

$$s_0 \models F_{\text{pre}} \implies s_n \models F_{\text{post}}$$
Overview

A function is **partially correct** if when the function's precondition is satisfied on entry, its postcondition is satisfied when it returns (if it ever does).
A function is **partially correct** if when the function’s precondition is satisfied on entry, its postcondition is satisfied when it returns (if it ever does).

Can we do this by only using the precondition and postcondition?

in most cases: NO!
A function is **partially correct** if when the function’s precondition is satisfied on entry, its postcondition is satisfied when it returns (if it ever does).

Can we do this by only using the precondition and postcondition?

**in most cases:** NO!

We **prove** partial correctness of programs by the **Inductive Assertion Method**.

For each function, we generate a **finite set of Verification Conditions (VC)**; if all VCs are correct, then the program is **partially correct**.
Some Definitions

- **Path**: sequence of program statements.

- **Basic Path**: a Path that starts at a precondition or a loop invariant, and ends at a loop invariant, an assertion, or a post condition.

```plaintext
@pre \(0 \leq \ell \land u < |a|\)
@post \(rv \iff \exists i. \ell \leq i \leq u \land a[i] = e\)
bool LinearSearch(int[] a, int \ell, int u, int e) {
    for
        @L : \(\ell \leq i \land (\forall j. \ell \leq j < i \rightarrow a[j] \neq e)\)
        (int i := \ell; i \leq u; i := i + 1) {
            if (a[i] = e) return true;
        }
    return false;
}
```
Some Definitions

- **Path**: sequence of program statements.

- **Basic Path**: a Path that starts at a precondition or a loop invariant, and ends at a loop invariant, an assertion, or a post condition.

```plaintext
@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;
}
```

Figure 5.15 lists an annotated version of Basic Paths of LinearSearch. The outer loop takes the form of a while loop, and the inner loop is a for loop. The final basic path has the same prefix as @L (of each element in the range [0, u - ℓ + 1], the loop index is at least ℓ, and the other has the statement L); otherwise, the programmer can add an assume statement representing the return value.
Some Definitions

- **Path**: sequence of program statements.

- **Basic Path**: a Path that starts at a precondition or a loop invariant, and ends at a loop invariant, an assertion, or a post condition.

```plaintext
@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;
}
```
Some Definitions

- **Path**: sequence of program statements.

- **Basic Path**: a Path that starts at a precondition or a loop invariant, and ends at a loop invariant, an assertion, or a post condition.

```java
@pre 0 \leq \ell \land u < |a|
@post \text{rv} \leftrightarrow \exists i. \ell \leq i \leq u \land a[i] = e

bool LinearSearch(int[] a, int \ell, int u, int e) {
    for
        @L: \ell \leq i \land (\forall j. \ell \leq j < i \rightarrow a[j] \neq e)
        (int i := \ell; i \leq u; i := i + 1) {
            if (a[i] = e) return true;
        }
    return false;
}

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

- **Path**: sequence of program statements.

- **Basic Path**: a Path that starts at a precondition or a loop invariant, and ends at a loop invariant, an assertion, or a post condition.

```plaintext
@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;
}
```
Some Definitions

- **Path:** sequence of program statements.

- **Basic Path:** a Path that starts at a precondition or a loop invariant, and ends at a loop invariant, an assertion, or a post condition.

```plaintext
@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;
}
```
Some Definitions

- **Path**: sequence of program statements.

- **Basic Path**: a Path that starts at a precondition or a loop invariant, and ends at a loop invariant, an assertion, or a post condition.

```plaintext
@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;
}
```

Figure 5.10 lists some definitions.
Inductive Assertion Method

- We reduce the reasoning about the function to reasoning about a finite set of basic paths.
Inductive Assertion Method

• We reduce the reasoning about the function to reasoning about a finite set of basic paths.

• We reason about the basic paths, by reducing the reasoning to a Verification Condition (VC).
Inductive Assertion Method

- We reduce the reasoning about the function to reasoning about a finite set of basic paths.

- We reason about the basic paths, by reducing the reasoning to a Verification Condition (VC).

The VC is

\[ x := x + 1; \]
Inductive Assertion Method

• We reduce the reasoning about the function to reasoning about a finite set of basic paths.

• We reason about the basic paths, by reducing the reasoning to a Verification Condition (VC).

\[
x := x + 1; \\
@ \ x \geq 1
\]
Inductive Assertion Method

- We **reduce** the reasoning about the function to reasoning about a finite set of basic paths.

- We reason about the basic paths, by reducing the reasoning to a Verification Condition (VC).

\[
\begin{align*}
\@ & x \geq 0 \\
x & := x + 1; \\
\@ & x \geq 1
\end{align*}
\]
P-Invariant vs. P-Inductive
P-Invariant vs. P-Inductive

- **P-Invariant**: an annotation \( F \) at location \( L \) of program \( P \) is P-invariant iff whenever program reaches location \( L \) during any computation with program state \( s \), then \( s \models F \).
P-Invariant vs. P-Inductive

• P-Invariant: an annotation $F$ at location $L$ of program $P$ is P-invariant iff whenever program reaches location $L$ during any computation with program state $s$, then $s \models F$.

$$s[pc] = L \implies s \models F$$
**P-Invariant vs. P-Inductive**

- **P-Invariant**: an annotation $F$ at location $L$ of program $P$ is P-invariant iff whenever program reaches location $L$ during any computation with program state $s$, then $s \models F$.

$$s[pc] = L \implies s \models F$$

- **P-Inductive**: if all verification conditions generated by the program are valid, then all program annotations are P-inductive.
P-Invariant vs. P-Inductive

- **P-Invariant**: an annotation $F$ at location $L$ of program $P$ is P-invariant iff whenever program reaches location $L$ during any computation with program state $s$, then $s \models F$.

$$s[pc] = L \implies s \models F$$

- **P-Inductive**: if all verification conditions generated by the program are valid, then all program annotations are P-inductive.

**Theorem**: p-inductive implies p-invariant.
**P-Invariant vs. P-Inductive**

- **P-Invariant**: an annotation $F$ at location $L$ of program $P$ is P-invariant iff whenever program reaches location $L$ during any computation with program state $s$, then $s \models F$.

$$s[pc] = L \implies s \models F$$

- **P-Inductive**: if all verification conditions generated by the program are valid, then all program annotations are P-inductive.

**Theorem**: p-inductive implies p-invariant.

For iterative programs, finding an inductive annotation mostly amounts to discovery of an appropriate loop invariant.
Examples
Involved Example

Example 5.7. Figure 5.8 lists BinarySearch with its specification. As expected, its postcondition is identical to the postcondition of LinearSearch. However, its precondition also states that the array \( a \) is sorted.

The sorted predicate is defined in the combined theory of integers and arrays, \( T_Z \cup T_A \):

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

Example 5.8. Figure 5.9 lists BubbleSort with its specification. Given any array, the returned array is sorted. Of course, other properties are desirable.