Induction and Program Correctness

Peter J. Haas

INFO 150
Fall Semester 2019
Overview

Goal

- Apply inductive reasoning to Java programs
- In context of loops and recursion
Program Correctness

Informally: A program is correct if it performs according to its specification.

- If certain inputs are given, then certain outputs will be obtained.
- If other inputs are given, then program is not incorrect, even if it throws an exception or enters an infinite loop.

Definition: Pre-conditions and post-conditions are sets of propositions that describe inputs, outputs, object states, aspects of environment.

Definition: A program is partially correct if, when the pre-conditions hold prior to a program run and the program terminates, then the post-conditions will hold.

Note: A program that never terminates is always partially correct.

We usually make separate proofs for termination and correctness.
Program Correctness

Informally: A program is correct if it performs according to its specification.

- If certain inputs are given, then certain outputs will be obtained
- If other inputs are given, then program is not incorrect, even if it throws an exception or enters an infinite loop

Definition

Pre-conditions and post-conditions are sets of propositions that describe inputs, outputs, object states, aspects of environment.
Program Correctness

**Informally:** A program is **correct** if it performs according to its **specification**.

- If certain inputs are given, then certain outputs will be obtained
- If other inputs are given, then program is not incorrect, even if it throws an exception or enters an infinite loop

**Definition**

*Pre-conditions and post-conditions* are sets of propositions that describe inputs, outputs, object states, aspects of environment.

**Definition**

A program is **partially correct** if, when the pre-conditions hold prior to a program run and the program terminates, then the post-conditions will hold.
**Program Correctness**

**Informally:** A program is correct if it performs according to its specification.
- If certain inputs are given, then certain outputs will be obtained
- If other inputs are given, then program is not incorrect, even if it throws an exception or enters an infinite loop

**Definition**

*Pre-conditions and post-conditions* are sets of propositions that describe inputs, outputs, object states, aspects of environment.

**Definition**

A program is partially correct if, when the pre-conditions hold prior to a program run and the program terminates, then the post-conditions will hold.

**Note:** A program that never terminates is always partially correct
- We usually make separate proofs for termination and correctness
Example: Calculating Remainders

Algorithm: Compute the remainder when $n$ is divided by $b$ (i.e., $n \mod b$)

```c
int remainder (int n, int b) {
    int x = n;
    while (x >= b) x -= b;
    return x;
}
```

Pre-conditions: $n \geq 0$ and $b > 0$

Post-conditions: $0 \leq \text{output} < b$ and $\exists k : n = kb + \text{output}$

If preconditions not true, we might get an output that violates the post-conditions

- Ex: $n = -1$ and $b = 2$: returns $-1$ (should be $1$ since $n = 2 \cdot -1 + 1$)
- Ex: $n = 3$ and $b = -2$: infinite loop

Will show both termination and correctness using induction
Example: Calculating Remainders

```c
int remainder (int n, int b) {
    int x = n;
    while (x >= b) x -= b;
    return x;
}
```

**Pre-conditions:** \( n \geq 1 \) and \( b > 0 \)

**Post-conditions:** \( 0 \leq \text{output} < b \) and \( \exists k : n = kb + \text{output} \)

**Inductive proof of \( P(n) \) for fixed \( b > 0 \)**

1. \( n = 1 \):
   1.1 Case 1: if \( b = 1 \), returns 0 after going through while loop once ✓
   1.2 Case 2: if \( b > 1 \), returns 1 without going through while loop ✓

2. Assume that program is correct for \( n = 1, 2, \ldots, m - 1 \), so need to prove \( P(m) \)
   2.1 Case 1: if \( m < b \), returns \( m \) without going through while loop ✓
   2.2 Case 2: if \( m \geq b \), enters while loop and changes \( x \) to \( m - b \).
      2.2.1 Now as if we started algorithm with inputs of \( m - b \) and \( b \)
      2.2.2 By induction, returns output satisfying post-conditions for \( m - b, b \)
      2.2.3 \( 0 \leq \text{output} < b \ ✓
      2.2.4 \( \exists k : m - b = kb + \text{output} \)
      2.2.5 For this \( k \), we have \( m = (k + 1)b + \text{output} \ ✓
```

\[ m = (m - b) + b \]
Example: Calculating Remainders Recursively

Algorithm: Recursively compute the remainder when \( n \) is divided by \( b \)

```c
int remainder (int n, int b) {
    if (n < b) return n;
    return remainder(n - b, b);
}
```

**Pre-conditions:** \( n \geq 1 \) and \( b > 0 \)

**Post-conditions:** \( 0 \leq \text{output} < b \) and \( \exists k : n = kb + \text{output} \)

**Inductive proof of \( P(n) \) for fixed \( b > 0 \)**

1. \( n = 1 \):
   1.1 Case 1: if \( b = 1 \), does recursive call with 0 and \( b \), which returns 0 ✓
   1.2 Case 2: if \( b > 1 \), returns 1 without recursive call ✓

2. Assume that program is correct for \( n = 1, 2, \ldots, m - 1 \), so need to prove \( P(m) \)
   2.1 Case 1: if \( m < b \), returns \( m \) without recursive call ✓
   2.2 Case 2: if \( m \geq b \), does recursive call with \( m - b \) and \( b \).
      2.2.1 By induction, returns output that satisfying post-conditions
      2.2.2 \( 0 \leq \text{output} < b \) ✓
      2.2.3 \( \exists k : m - b = kb + \text{output} \)
      2.2.4 For this \( k \), we have \( m = (k + 1)b + \text{output} \) ✓
Example: Recursively Computing Factorials

Algorithm: Recursively compute \( n! \)

```c
int factorial (int n);
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

Easy to show inductively that algorithm terminates

1. \( P(1) \): If \( n = 1 \), algorithm terminates and returns 1 (line 2)
2. Let \( m \geq 2 \) and assume that \( P(1), P(2), \ldots, P(m-1) \) hold
3. \( P(m) \): If input = \( m \), then line 3 executes \( \text{factorial}(m-1) \) which terminates since \( P(m-1) \) is true

To prove correctness is also easy

- Define \( n! \) recursively by \( 1! = 1 \) and \( n! = n \cdot (n - 1)! \) for \( n > 1 \)
- Proof follows immediately

In general, recursive algorithms lead naturally to inductive proofs
Example: Recursively Printing Prime Factorization

Algorithm: Print a List of Prime Factors

```java
void factor (int n) {
    if (n == 1) return;
    int d = 2;
    while (n % d != 0) d++;
    System.out.println(d);
    factor (n/d);
}
```

Example of operation: call factor(60)
1. print a 2, call factor(30)
2. print a 2, call factor(15)
3. print a 3, call factor(5)
4. print a 5, and call factor(1) which terminates without doing anything.
Example: Recursively Printing Prime Factorization

Algorithm: Print a List of Prime Factors

```java
void factor (int n) {
    if (n == 1) return;
    int d = 2;
    while (n % d != 0) d++;
    System.out.println(d);
    factor (n/d);
}
```

Example of operation: call factor(60)
1. print a 2, call factor(30)
2. print a 2, call factor(15)
3. print a 3, call factor(5)
4. print a 5, and call factor(1) which terminates without doing anything.

Inductive proof of correctness
1. Define $P(n)$ as “on input $n$, factor terminates and prints a sequence of prime numbers that multiply to give $n$”
2. $P(1)$ is true because factor(1) terminates and prints nothing, empty sequence multiplies to give 1 (by definition)
3. {Complete the rest of the proof as homework}