COMP 382: Reasoning about algorithms
Fall 2014

Swarat Chaudhuri & John Greiner

Unit 3: Proving algorithms correct
Prove that an algorithm written in an imperative language is correct

A.k.a., induction strikes again
Induction for functional programs:

- The program computes the right outputs on the empty list.
- Assuming the program computes the right outputs on a list \( L \), it computes the right outputs on \( m :: L \).
Induction for functional programs:
- The program computes the right outputs on the empty list
- Assuming the program computes the right outputs on a list $L$, it computes the right outputs on $m :: L$

Induction for imperative programs:
- All program executions of length 1 are correct
- If all executions of length $k$ lead to correct outputs, then so do all executions of length $(k + 1)$
How do we know this program is correct?

```java
method Find(a: array<int>, x: int) returns (j : int)
requires a != null;
{
    var m := 0;
    var n := a.Length;
    while (m < n)
    {
        j := (m + n) / 2;
        if (a[j] < x) {
            m := j + 1;
        } else if (x < a[j]) {
            n := j;
        } else {
            return;
        }
    }
    j := -1;
}
```

Swarat Chaudhuri & John Greiner
COMP 382: Reasoning about algorithms
Operational semantics defines how a program *actually* behaves.

**Specifications** state how a program *should* behave.

**Goal:** Guarantee that the program follows the specification.
We will use an automated *proof assistant* to do proofs of programs
- You write the proof, the assistant checks it for you
- The ultimate TA; doesn’t allow you to cheat

**Dafny**

http://rise4fun.com/dafny
A **(state) predicate** is a boolean function on the program state.

- $x = 8$
- $x < y$
- $m \leq n \rightarrow (\forall j : 0 \leq j < a.length \cdot a[j] \neq NaN)$
- true
- false

Intuitively, a **property** that holds at a point in a program execution.
For predicates $P$ and $Q$ and program $S$, 

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

says that if $S$ is started at (a state satisfying) $P$, then it terminates at $Q$
For predicates $P$ and $Q$ and program $S$,

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

says that if $S$ is started at (a state satisfying) $P$, then it terminates at $Q$

- $P$ is a *precondition*
- $Q$ is a *postcondition*
Examples

\[ \{ \text{true} \} x := 12 \{ x = 12 \} \]

\[ \{ x < 40 \} x := 12 \{ 10 \leq x \} \]

\[ \{ x < 40 \} x := x + 1 \{ x \leq 40 \} \]

\[ \{ 0 \leq m < n \leq a\.length \land a[m] = x \} r := \text{Find}(a, m, n, x) \{ m \leq r \} \]
If $\{P\}S\{Q\}$ and $\{P\}S\{R\}$, then $\{P\}S\{Q \land R\}$ holds.

- If $\{x = 2\}x := x + 1\{x > 0 \land x < 100\}$ and $\{x = 2\}x := x + 1\{x > 2\}$
- ...then $\{x = 2\}x := x + 1\{x > 0 \land x < 100\}$

The most precise $Q$ such that $\{P\}S\{Q\}$ is called the strongest postcondition of $S$ with respect to $P$.

- $\{x = 2\}x := x + 1\{x = 3\}$
Weakest preconditions

If \( \{P\}S\{R\} \) and \( \{Q\}S\{R\} \), then \( \{P \lor Q\}S\{R\} \) holds.

- If \( \{x > 1\}x := x^2\{x > 1\} \) and \( \{x < -1\}x := x^2\{x > 1\} \)
- ... then \( \{x > 1 \lor x < -1\}x := x^2\{x > 1\} \)

The most general \( P \) such that \( \{P\}S\{R\} \) is called the weakest precondition of \( S \) with respect to \( R \), written \( wp(S, R) \).
Here’s the crucial relationship between Hoare triples and weakest preconditions:

{P}S{Q} if and only if $P \rightarrow wp(S, Q)$
Consider programs of different shapes; for each shape, give method for systematic correctness proof

\[ e ::= f(e_1, e_2) \mid x \mid n \quad \text{where } x \text{ is a variable,} \]
\[ \quad n \text{ is a constant, } f \text{ is a function} \]
\[ b ::= R(e_1, e_2) \quad \text{where } R \text{ is a predicate} \]
\[ S ::= x := e \mid S_1; S_2 \mid \text{skip} \mid \]
\[ \quad \text{if } b \ S_1 \text{ else } S_2 \mid \text{while } B \ S_1 \]
Proving programs correct: \texttt{skip}

\[
wp(\texttt{skip}, R) \equiv R
\]

Example:

\[
wp(\texttt{skip}, x^n + y^n = z^n) \equiv x^n + y^n = z^n
\]

To prove \{P\} \texttt{skip}\{Q\}, show that $P \rightarrow Q$


$\text{wp}(w := E, R) \equiv R[w \mapsto E]$

where $R[w \mapsto E]$ is obtained by starting with $R$ and replacing $w$ by $E$

$\text{wp}(x := x + 1, x \leq 10) \equiv x + 1 \leq 10 \equiv x < 10$

$\text{wp}(x := 15, x \leq 10) \equiv 15 \leq 10 \equiv \text{false}$

$\text{wp}(y := x + 3 \ast y, x \leq 10) \equiv x \leq 10$

To prove $\{P\}w := E\{Q\}$, show that $P \rightarrow Q[w \mapsto E]$
method foo(x: int) returns (y: int)
requires x > 0;
ensures y > 1;
{
    y := x + 1;
}
Sequential composition

\[ wp(S; T, R) \equiv wp(S, wp(T, R)) \]

**Example:** Compute the value of

\[ wp(y := y + 1; x := x + 3 \times y, y \leq 10 \land 3 \leq x) \]
Branching

\[
wp(\text{if } B \text{ then } S \text{ else } T, R) \\
\equiv (B \land wp(S, R)) \lor (\neg B \land wp(T, R))
\]
Branching

\[ wp(\text{if } B \text{ then } S \text{ else } T, R) \equiv (B \land wp(S, R)) \lor (\neg B \land wp(T, R)) \]

- \((B \land wp(S, R))\) is the set of states that pass the test “if B” and lead to R when “pushed” through S.
- \((\neg B \land wp(S, R))\) is the set of states that fail the test “if B” and lead to R when “pushed” through T.

Example: Compute the value of

- \(wp(\text{if } x < y \text{ then } z := y \text{ else } z := x, 0 \leq z)\)
- \(wp(\text{if } x \neq 10 \text{ then } x := x + 1 \text{ else } x := x + 2, x \leq 10)\)
What’s the condition under which this program will use pointers safely?

```java
if (x != null) {
    n := x.f;
} else {
    n := z-1;
    z := z + 1;
}
a = new char[n];
```
To prove

\{P\} while B \ S\{Q\}

find invariant $J$ and a ranking function $rf$ such that:

- invariant holds initially: $P \rightarrow J$
- invariant is maintained: $\{J \land B\} S\{J\}$
- invariant is sufficient: $J \land \neg B \rightarrow Q$
- ranking function is bounded: $J \land B \rightarrow 0 \leq rf$
- ranking function decreases: $\{J \land B \land rf = RF\} S\{rf < RF\}$
Example: Array sum

1 \{N \geq 0\};
2 k := 0;
3 s := 0;
4 \textbf{while} (k \neq N) \{ 
5 \quad s := s + a[k];
6 \quad k := k + 1
7 \}
8 \{s = \sum_{0 \leq i < N} a[i]\}
Example: Array sum

1 \{N \geq 0\}
2 k := 0;
3 s := 0;
4 \{J\}
5 \textbf{while} (k \neq N) \{
6 \quad \{rf = RF\}
7 \quad s := s + a[k];
8 \quad k := k + 1
9 \quad \{J \land rf < RF\}
10 \}
11 \{J \land \neg(k \neq N)\}
12 \{s = \sum_{0 \leq i < N} a[i]\}
Example: Array sum

\[
\begin{align*}
N \geq 0 & , \\
 k := 0 ; & \\
 s := 0 ; & \\
 \{ J \} & \\
 \textbf{while} \ (k \neq N) \ { \{ rf = RF \} } & \\
 & \{ J \land rf < RF \} \\
 & \{ J \land \neg (k \neq N) \} \\
 & \{ s = \sum_{0 \leq i < N} a[i] \} \\
\end{align*}
\]

\[
J : \quad s = \sum_{0 \leq i < k} a[i] \land 0 \leq k \leq N \\
rf : \quad N - k
\]
Exercise: Computing cubes

1 method Cube(N: int) returns (c: int)
2   requires 0 <= N;
3   ensures c == N*N*N;
4 {
5     c := 0;
6     var n := 0;
7     var k := 1;
8     var m := 6;
9     while (n < N) {
10        c := c + k;
11        k := k + m;
12        m := m + 6;
13        n := n + 1;
14    }
15}
Exercise: Computing cubes

```
method Cube(N: int) returns (c: int)
  requires 0 <= N;
  ensures c == N*N*N;
{
  c := 0;
  var n := 0; var k := 1; var m := 6;
  while (n < N)
    invariant n <= N;
    invariant c == n*n*n;
    invariant k == 3*n*n + 3*n + 1;
    invariant m == 6*n + 6;
    decreases (N - n);
    {
      c := c + k;
      k := k + m;
      m := m + 6;
      n := n + 1;
    }
}
```
method Foo(x: int, y: int, z: int) {
    var x := x;
    var y := y;
    while (x > 0 && y > 0)
    {
        if (z == 1) {
            x := x - 1;
            y := y + 1;
        }
        else {
            y := y - 1;
        }
    }
}
Suppose you have a program

```java
method M( ) {
    ...
    P( );
    ...
}
```

Proof goals:

- At the point when control enters `P` from within `M`, the ranking function of `P` must have a lower value than the ranking function of `M`.
method Ackermann(m: int, n: int) returns (r: int)
decreases m, n;
requires m >= 0 && n >= 0;
ensures r > 0;
{
    if (m <= 0) {
        r := n + 1;
    }
    else if (n <= 0) {
        r := Ackermann(m - 1, 1);
    }
    else {
        var z;
        z := Ackermann(m, n - 1);
        r := Ackermann(m - 1, z);
    }
}
Dafny also permits a “functional” notation:

```dafny
function Ackermann(m: int, n: int): int
  decreases m, n;
  requires m >= 0 && n >= 0;
  ensures Ackermann(m, n) > 0;
{
  if m <= 0 then
    n + 1
  else if n <= 0 then
    Ackermann(m - 1, 1)
  else
    Ackermann(m - 1, Ackermann(m, n - 1))
}
```
Can you show that \texttt{Compute_Fib} is correct?

```plaintext
function Fib(n: nat): nat
{
  if n < 2 then n else Fib(n - 1) + Fib(n-2)
}

method Compute_Fib(n: nat) returns (x: nat)
ensures x == Fib(n);
{
  var i := 0;
  x := 0;
  var y := 1;
  while (i < n) {
    x, y := y, x + y;
    i := i + 1;
  }
}
```
method Compute_Fib(n: nat) returns (x: nat)
ensures x == Fib(n);
{
  var i := 0;
x := 0;
  var y := 1;
  while (i < n)
    invariant n >= i && i >= 0;
invariant y == Fib(i + 1);
invariant x == Fib(i);
  {
    x, y := y, x + y;
    i := i + 1;
  }
}
method Odd(n: nat) returns (b: bool)
ensures (b <==> (n % 2 == 1));
{
    if (n == 0) {
        b := false; }
    else {
        b := Even(n - 1); }
}

method Even(n: nat) returns (b: bool)
ensures (b <==> (n % 2 == 0));
{
    if (n == 0) {
        b := true; }
    else {
        b := Odd(n - 1); }
}
method BinarySearch(a: array<int>, value: int)
    returns (index: int)
{
    var low, high := 0, a.Length;
    while (low < high) {
        var mid := (low + high) / 2;
        if (a[mid] < value) {
            low := mid + 1;
        } else if (value < a[mid]) {
            high := mid;
        } else {
            return mid;
        }
    }
    return -1;
}
method BinarySearch(a: array<int>, value: int)
    returns (index: int)
requires a != null && 0 <= a.Length;
requires
    forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k];
ensures
    0 <= index ==> index < a.Length && a[index] == value;
ensures
    index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != value;
{
    var low, high := 0, a.Length;
    while (low < high) {
        ...
    }
    return -1;
1
2 var low, high := 0, a.Length;
3 while (low < high)
4 invariant 0 <= low <= high <= a.Length;
5 invariant forall i :: 0 <= i < a.Length &&
6 !(low <= i < high) ==> a[i] != value;
7 {
8 var mid := (low + high) / 2;
9 if (a[mid] < value) {
 10   low := mid + 1;
 11 } else if (value < a[mid]) {
 12   high := mid;
 13 } else {
 14   return mid; }
 15 }
 16 return -1;
Dafny and similar systems constitute the discipline of *program verification*

An ambitious approach to reliable software
  - Design the system so that security flaws are impossible!

Recent successes:
  - Verification of Windows device drivers
  - Verified C compiler back-ends
  - Verified microkernels
  - ...
**Dijkstra’s map problem**

**Input:** Two sets of points (call them Red and Blue) in $\mathbb{R}^2$ of equal cardinality

**Matching:** A set of undirected line segments such that (1) each segment connects a red point to a blue point; and (2) only one line out of each point;

**Goal:** Show that the following algorithm for computing a matching terminates:

```plaintext
1. choose any one-one mapping;
2. while (exists crossing)
   3. uncross a pair of crossing lines
```
Puzzle: McCarthy’s function

Does this method terminate?

```java
method McCarthy(n0: int) returns (c : int) {
    var n := n0;
    c := 1;
    while (c > 0) {
        if (n > 100) {
            n := n - 10;
            c := c - 1;
        } else {
            n := n + 11;
            c := c + 1;
        }
    }
}
```
method McCarthy(n0: int) returns (c : int) {
    var n := n0;
    c := 1;
    while (c > 0)
        decreases 101 - n + 10 * c, c;
        {
            if (n > 100) {
                n := n - 10;
                c := c - 1;
            }
        }
    else {
        n := n + 11;
        c := c + 1;
    }
}
Does this method terminate?

```java
method McCarthy(n0: int) returns (c : int)
requires n0 >= 0;
{
    var n := n0;
    var z: int;
    if (n <= 100) {
        z := McCarthy (n + 11);
        c := McCarthy (z);
        return;
    }
    else {
        return (n - 10);
    }
}
```
McCarthy’s function

```plaintext
function f (i: int): int {
    if (i <= 100) then 91 else (i - 10) }

method McCarthy(n0: int) returns (c : int)
requires n0 >= 0;
ensures c == f(n0);
{
    var n := n0;
    var z: int;
    if (n <= 100) {
        z := McCarthy (n + 11);
        c := McCarthy (z);
        return;
    }
    else {
        return (n - 10);
    }
}
```
function \( f \) (i: int): int {
    if (i <= 100) then 91 else (i - 10) }

method McCarthy(n0: int) returns (c : int)
requires n0 >= 0;
decreases 101 - n0;
ensures c == f(n0);
{
    var n := n0;
    var z: int;
    if (n <= 100) {
        z := McCarthy (n + 11);
        c := McCarthy (z);
        return;
    }
    else {
        return (n - 10);
    }
}