Deductive Verification

Mooly Sagiv

Original Slides: Rustan Leino (MSR)
Swart Chaudhuri (Rice U)

Formal Semantics of Programming Languages: Glynn Winskel Chapter 6
Goal: Correctness Proof

- Prove that an algorithm written in an imperative language is correct
Proving Correctness

• 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
  • The technical part of the proof is automated
  • The user is responsible for the loop invariants
Functions on Lists

```
let rec length l =
  match l with
  | [] -> 0
  | hd :: tl -> 1 + length tl

length [1; 2; 3] + length ["red"; "yellow"; "green"]
```
Induction for algorithmic correctness

• 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?

```csharp
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;
}
```
Correctness of programs

• *Operational semantics* defines how a program actually behaves
• *Speciations* state how a program should behave
• **Goal:** Guarantee that the program follows the
Loop invariants

Reachable states

Error states

Inv

Counterexample to induction
Simple Example: loop Invariants

1: x := 1;
2: y := 2;
while * do {
  3: assert odd[x];
  4: x := x + y;
  5: y := y + 2
}
6:
Simple Example: loop Invariants

1: x := 1;
2: y := 2;
while * do {
  3: assert odd[x];
  4: x:= x + y;
  5: y := y + 2
}
6:

Inv = odd[x] \land \neg odd[y]
Deductive verification by reductions to SAT

- Program Tr(X, X')
- Candidate Invariant Inv
  \( \text{Inv} \Rightarrow \varphi \)
- Desired Property \( \varphi \)

Front-End

\[ \text{Inv}(X) \land [\text{Tr}] (X', X) \land \neg \text{Inv} (X') \]

SAT(DPLL)

Y

N

Counterexample to Induction (CTI)

Proof

\( \neg \text{Inv}(X') \)
Deductive Verification

Is there a behavior of P that violates the inductiveness of I?

1: x := 1;
2: y := 2;
while * do {
  3: assert odd[x];
  4: x := x + y;
  5: y := y + 2
}
6:

Inv = at(3) ⇒ odd[x]

Solver

at(3) ⇒ odd[x]

odd[x]' = (odd[x]∧¬odd[y])∨(¬odd[x]∧odd[y])
odd[y]' = odd[y]

3: odd[x]=1, odd[y]=1

¬Inv

odd[x]' = 0, odd[y]' = 1
Deductive Verification

Is there a behavior of $P$ that violates the inductiveness of $I$?

Proof

N
Inductive (loop) Invariants Mists and Reality

• Mist
  • An *inductive* invariant is a property that holds in every iteration

• Reality
  • An *inductive* invariant is a property that
    • Holds at every iteration
    • Suffices to prove that the invariant is preserved by any execution
Inductive Invariants Formally

• Inputs:
  • A set of states $\Sigma$
  • A set of initial states $\Sigma_0 \subseteq \Sigma$
  • A transition relation $\tau \subseteq \Sigma \times \Sigma$

• The set of reachable states is defined by:
  • $\text{reach}[0] = \Sigma_0$
  • $\text{reach}[i+1] = \{ \sigma' \in \Sigma \mid \sigma \in \text{reach}[i] \land <\sigma, \sigma'> \in \tau \}$
  • $\text{reach}[\infty] = \text{reach}[0] \cup \text{reach}[1] \cup \ldots$

• A set $\text{Inv} \subseteq \Sigma$ is inductive if
  • $\Sigma_0 \subseteq \text{Inv}$ (Initiation)
  • For all $\sigma \in \text{Inv}$ and $<\sigma, \sigma'> \in \tau$, $\sigma' \in \text{Inv}$ (Consecution)

• A set $\text{Inv} \subseteq \Sigma$ is inductive for $P \subseteq \Sigma$ if it is inductive and $\text{Inv} \subseteq P$
  • “The reason why $P$ holds”
Simple Example

```
require x >= 1
int it = x;
int res = 1;
int z = 0
while it != 1 do
    res := res * it
    it := it - 1
    z := z + 1
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Inductive</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>yes</td>
</tr>
<tr>
<td>res ≥0</td>
<td>no</td>
</tr>
<tr>
<td>res ≥0 ∧ it ≥ 1</td>
<td>yes</td>
</tr>
<tr>
<td>res ≥0 ∧ it ≥ 1 ∧ x ≥ 1</td>
<td>yes</td>
</tr>
<tr>
<td>res ≥0 ∧ it ≥ 1 ∧ res = x!/it!</td>
<td>yes</td>
</tr>
<tr>
<td>res ≥0 ∧ it ≥ 1 ∧ res = x!/it! ∧ z + it = x</td>
<td>yes</td>
</tr>
</tbody>
</table>
The **While** Programming Language

- Abstract syntax
  \[ S ::= x := a \mid \text{skip} \mid S_1 ; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S \]

- Use parentheses for precedence

- Informal Semantics
  - **skip** behaves like no-operation
  - Import meaning of arithmetic and Boolean operations
Example While Program

\[
y := 1; \\
\text{while } \neg(x=1) \text{ do (} \\
\quad y := y * x; \\
\quad x := x - 1; \\
\text{)}
\]
General Notations

• Syntactic categories
  – Var the set of program variables
  – Aexp the set of arithmetic expressions
  – Bexp the set of Boolean expressions
  – Stm set of program statements

• Semantic categories
  – Natural values \( Z = \{..., -2, -1, 0, 1, 2, ...\} \)
  – Truth values \( T = \{ff, tt\} \)
  – \( \Sigma = \text{Var} \rightarrow \text{Z} \)
  – Lookup in a state \( \sigma: \sigma \ x \)
  – Update of a state \( \sigma: \sigma \ [x \mapsto 5] \)
Example State Manipulations

• \([x\leftarrow 1, y\leftarrow 7, z\leftarrow 16]\) \(y = \)
• \([x\leftarrow 1, y\leftarrow 7, z\leftarrow 16]\) \(t = \)
• \([x\leftarrow 1, y\leftarrow 7, z\leftarrow 16][x\leftarrow 5] = \)
• \([x\leftarrow 1, y\leftarrow 7, z\leftarrow 16][x\leftarrow 5] x = \)
• \([x\leftarrow 1, y\leftarrow 7, z\leftarrow 16][x\leftarrow 5] y = \)
Semantics of arithmetic expressions

• Assume that arithmetic expressions are side-effect free
• \( A[\text{ Aexp }] : \Sigma \rightarrow \mathbb{Z} \)
• Defined by **structural** induction on the syntax tree
  
  – \( A[\ n \ ] \sigma = n \)
  
  – \( A[\ x \ ] \sigma = \sigma x \)
  
  – \( A[\ e_1 + e_2 \ ] \sigma = A[\ e_1 \ ] \sigma + A[\ e_2 \ ] \sigma \)
  
  – \( A[\ e_1 * e_2 \ ] \sigma = A[\ e_1 \ ] \sigma * A[\ e_2 \ ] \sigma \)
  
  – \( A[\ - e_1 \ ] \sigma = -A[\ e_1 \ ] \sigma \)
Semantics of Boolean expressions

- Assume that Boolean expressions are side-effect free
- \( B[B\text{exp}] : \Sigma \rightarrow \mathbb{T} \)
- Defined by induction on the syntax tree
  - \( B[\text{true}] \sigma = \text{tt} \)
  - \( B[\text{false}] \sigma = \text{ff} \)
  - \( B[e_1 = e_2] \sigma = \begin{cases} \text{tt} & \text{if } A[e_1] \sigma = \text{tt} \text{ and } B[e_2] \sigma = \text{tt} \\ \text{ff} & \text{if } B[e_1] \sigma = \text{ff} \text{ or } B[e_2] \sigma = \text{ff} \end{cases} \)
  - \( B[e_1 \land e_2] \sigma = \begin{cases} \text{tt} & \text{if } A[e_1] \sigma = A[e_2] \sigma \\ \text{ff} & \text{if } A[e_1] \sigma \neq A[e_2] \sigma \end{cases} \)
  - \( B[e_1 \geq e_2] s = \begin{cases} \text{tt} & \text{if } A[e_1] \sigma \geq A[e_2] \sigma \\ \text{ff} & \text{if } A[e_1] \sigma < A[e_2] \sigma \end{cases} \)
Natural Semantics for While

Axioms

[ass_{ns}] \langle x := a, \sigma \rangle \rightarrow \sigma[x \mapsto A[a]]\sigma

[skip_{ns}] \langle skip, \sigma \rangle \rightarrow \sigma

[comp_{ns}] \langle S_1, \sigma \rangle \rightarrow \sigma', \langle S_2, \sigma' \rangle \rightarrow \sigma''

\langle S_1; S_2, \sigma \rangle \rightarrow \sigma''

Rules

[if^{tt}_{ns}] \langle S_1, \sigma \rangle \rightarrow \sigma'

\langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \sigma'

if \text{ } B[a] \sigma = \text{tt}

[if^{ff}_{ns}] \langle S_2, \sigma \rangle \rightarrow \sigma'

\langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \sigma'

if \text{ } B[a] \sigma = \text{ff}
Natural Semantics for While (More rules)

\[
\begin{align*}
\text{[while}^{\text{ff}}_{\text{ns}}] & \quad <\text{while } b \text{ do } S, \sigma> \rightarrow \sigma \\
& \quad \text{if } B[b]\sigma=\text{ff}
\end{align*}
\]

\[
\begin{align*}
\text{[while}^{\text{tt}}_{\text{ns}}] & \quad <S, \sigma> \rightarrow \sigma', <\text{while } b \text{ do } S, \sigma'> \rightarrow \sigma'' \\
& \quad <\text{while } b \text{ do } S, \sigma> \rightarrow \sigma'' \\
& \quad \text{if } B[b]\sigma=\text{tt}
\end{align*}
\]
Simple Examples

• Let $\sigma_0$ be the state which assigns zero to all program variables

• Assignments
  
  \[\text{Assignments} \quad [\text{ass}_{ns}] \langle x := x+1, \sigma_0 \rangle \rightarrow [x \leftarrow 1]\]

• Skip statement
  
  \[\text{Skip statement} \quad [\text{skip}_{ns}] \langle \text{skip}, \sigma_0 \rangle \rightarrow \sigma_0\]

• Composition

 \[\begin{align*}
 [\text{comp}_{ns}] \langle \text{skip}, \sigma_0 \rangle & \rightarrow \sigma_0, \\
 [x := x+1, \sigma_0] & \rightarrow [x \leftarrow 1] \\
 \text{<skip; x := x +1, } & \sigma_0 \rightarrow [x \leftarrow 1] 
\end{align*}\]
A Derivation Tree

• A “proof” that \(<S, \sigma> \rightarrow \sigma'\)
• The root of tree is \(<S, \sigma> \rightarrow \sigma'\)
• Leaves are instances of axioms
• Internal nodes rules
  – Immediate children match rule premises
• Simple Example
An Example Derivation Tree

\[(x := x+1; y := x+1) ; z := y), \sigma_0 \rightarrow [x \mapsto 1][y \mapsto 2][z \mapsto 2]\]
Top Down Evaluation of Derivation Trees

• Given a program S and an input state \( \sigma \)
• Find an output state \( \sigma' \) such that
  \[ \langle S, \sigma \rangle \rightarrow \sigma' \]
• Start with the root and repeatedly apply rules until the axioms are reached
• Inspect different alternatives in order
• In While \( \sigma' \) and the derivation tree is unique
Example of Top Down Tree Construction

• Input state $s$ such that $\sigma x = 2$

• Factorial program

$$<y := 1; \text{while } \neg(x=1) \text{ do } (y := y \cdot x; x := x - 1), \sigma > \rightarrow [y \mapsto 2][x \mapsto 1]$$
Program Termination

• Given a statement S and input $\sigma$

• S terminates on $\sigma$ if there exists a state $s'$ such that
  $<S, \sigma> \rightarrow s'$
  – S loops on s if there is no state $s'$ such that
    $<S, \sigma> \rightarrow s'$

• Given a statement S
  – S always terminates if for every input state $\sigma$, S terminates on $\sigma$
  – S always loops if for every input state $\sigma$, S loops on $\sigma$
The Semantic Function

- The meaning of a statement $S$ is defined as a partial function from $\Sigma$ to $\Sigma$
- $S_{ns} : Stm \rightarrow \Sigma_\bot \rightarrow \Sigma_\bot$
- $S_{ns} \llbracket S \rrbracket \sigma = \sigma'$ if $<S, \sigma> \rightarrow \sigma'$ and otherwise
  
  $S_{ns} \llbracket S \rrbracket \sigma = \bot$
- $S_{ns} \llbracket S \rrbracket \bot = \bot$
- Examples
  - $S_{ns} \llbracket \text{skip} \rrbracket \sigma = \sigma$
  - $S_{ns} \llbracket x := 1 \rrbracket \sigma = \sigma [x \mapsto 1]$
  - $S_{ns} \llbracket \text{while true do skip} \rrbracket \sigma = \bot$
An Assertion Language

• Extend Bexp

• Allow quantifications
  – $\forall i$: ...
  – $\exists i$: ...
  - $\exists i. k = i \times l$

• Import well known mathematical concepts
  – $n! \triangleq n \times (n-1) \times \cdots \times 2 \times 1$
Assertion Language

Aexpv

\[ a := n \mid X \mid i \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1 \]

Assn

\[ A := \text{true} \mid \text{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid A_0 \land A_1 \mid A_0 \lor A_1 \mid \neg A \mid A_0 \Rightarrow A_1 \mid \forall i. A \mid \exists i. A \]
Example

while \neg(M=\text{N}) \ do \\
\text{if } M \leq N \\
\quad \text{then } \text{N} := \text{N} - \text{M} \\
\text{else } \text{M} := \text{M} - \text{N}
Example

```c
int power(int x, unsigned int y)
{
    int temp;
    if (y == 0)
        return 1;
    temp = power(x, y/2);
    if (y%2 == 0)
        return temp*temp;
    else
        return x*temp*temp;
}
```
Free and Bound Variables

• An integer variable is **bound** when it occurs in the scope of a quantifier
• Otherwise it is **free**
• Examples

\[
\exists i. \, k = i \times l \land (i + 100 \leq 77) \land \forall i. j + 1 = i + 3
\]

\[
\begin{align*}
FV(n) &= FV(X) = \emptyset \\
FV(i) &= \{i\}
\end{align*}
\]

\[
\begin{align*}
FV(a_0 + a_1) &= FV(a_0 - a_1) = FV(a_0 \times a_1) = FV(a_0) \cup FV(a_1) \\
FV(\text{true}) &= FV(\text{false}) = \emptyset \\
FV(\text{true}) &= FV(\text{false}) = \emptyset \\
FV(\text{true}) &= FV(\text{false}) = \emptyset \\
FV(A_0 \land A_1) &= FV(A_0 \lor A_1) = FV(A_0 \Rightarrow A_1) = FV(A_0) \cup FV(A_1) \\
FV(\neg A) &= FV(A) \\
FV(\forall i. \, A) &= FV(\exists i. \, A) = FV(A) \setminus \{i\}
\end{align*}
\]
Substitution Arithmetic Expression

- Visualization of arithmetic expression
  ---i---i---
- Consider a “pure” arithmetic expression
  \(A[a/i]---a---a---\)

\[n[a/i] = n\]
\[X[a/i]=X\]
\[i[a/i] = a\]
\[j[a/i] = j\]
\[(a_0 + a_1)[a/i] = a_0[a/i] + a_1/[a/i]\]
\[(a_0 - a_1)[a/i] = a_0[a/i] - a_1[a/i]\]
\[(a_0 \times a_1)[a/i] = a_0[a/i] \times a_1[a/i]\]
Substitution Assertion

• Visualization of an assertion A

• Consider a “pure” arithmetic expression

  \[ A[a/i] \]

true[a/i] = true
false[a/i] = false

\[
(a_0 = a_1)[a/i] = (a_0[/a/i] = a_1[/a/i])
(a_0 \leq a_1)[a/i] = (a_0[/a/i] \leq a_1[/a/i])
(A_0 \land A_1)[a/i] = (A_0[/a/i] \land A_1[/a/i])
(A_0 \lor A_1)[a/i] = (A_0[/a/i] \lor A_1[/a/i])
(A_0 \Rightarrow A_1)[a/i] = (A_0[/a/i] \Rightarrow A_1[/a/i])
(\neg A)[a/i] = \neg (A[a/i])
\]

\[
(\forall i. A)[a/i] = \forall i. A
(\exists i. A)[a/i] = \exists i. A
\]

\[
(\forall j. A)[a/i] = (\forall j. A[a/i])
(\exists j. A)[a/i] = (\exists j. A[a/i])
\]
Location Substitution

• Visualization of an assertion $A$
  ---$X$---$X$---

• Consider a “pure” arithmetic expression
  $A[a/X]$ ---a---a---
Example Assertions

• i is a prime number
• i is the least common multiple of j and k
Semantics of Assertions

• An interpretation $I_z : z \mapsto \mathbb{Z}$

• The meaning of Aexpv
  – $A\llbracket n \rrbracket I_z \sigma = n$
  – $A\llbracket X \rrbracket I_z \sigma = \sigma(X)$
  – $A\llbracket i \rrbracket I_z \sigma = I_z(i)$
  – $A\llbracket a_0 + a_1 \rrbracket I_z \sigma = A\llbracket a_0 \rrbracket I_z \sigma + A\llbracket a_1 \rrbracket I_z \sigma$
  – ...

• For all $a \in \text{Aexp}$ states $\sigma$ and Interpretations $I_z$
  – $A\llbracket a \rrbracket \sigma = A\llbracket a \rrbracket I_z \sigma$
Semantics of Assertions (II)

- \( \text{Iz}[n/i] \) change \( i \) in \( \text{Iz} \) to \( n \)
- For \( \text{Iz} \) and \( \sigma \in \Sigma_{\perp} \), define \( \sigma \models_{\text{Iz}} A \) by structural induction
  - \( \sigma \models_{\text{Iz}} \text{true} \)
  - \( \sigma \models_{\text{Iz}} (a_0 = a_1) \) if \( \text{Av}[a_0] \models_{\text{Iz}} \sigma = \text{Av}[a_1] \models_{\text{Iz}} \sigma \)
  - \( \sigma \models_{\text{Iz}} (A \land B) \) if \( \sigma \models_{\text{Iz}} A \) and \( \sigma \models_{\text{Iz}} B \)
  - \( \sigma \models_{\text{Iz}} \neg A \) if not \( \sigma \models_{\text{Iz}} A \)
  - \( \sigma \models_{\text{Iz}} A \Rightarrow B \) if (not \( \sigma \models_{\text{Iz}} A \)) or \( \sigma \models_{\text{Iz}} B \)
  - \( \sigma \models_{\text{Iz}} \forall i.A \) if \( \sigma \models_{\text{Iz}[n/i]} A \) for all \( n \in \mathbb{N} \)
  - \( \bot \models_{\text{Iz}} A \)
Partial Correctness Assertions

• \{P\} S \{Q\}
  - \(P, Q \in \text{Assn} \text{ and } S \in \text{Stmt}\)
• For a state \(\sigma \in \Sigma_\perp\) and interpretation \(I_z\)
  - \(\sigma \models_{I_z} \{P\} S \{Q\} \text{ if } (\sigma \models_{I_z} P \Rightarrow S_{\ns} [S] \sigma \models_{I_z} Q)\)
• Validity
  - When \(\forall \sigma \in \Sigma_\perp, \sigma \models_{I_z} \{P\} S \{Q\}\) we write
    • \(\models_{I_z} \{P\} S \{Q\}\)
  - When \(\forall \sigma \in \Sigma_\perp\) and \(I \sigma \models_{I_z} \{P\} S \{Q\}\) we write
    • \(\models \{P\} S \{Q\}\)
    • \{P\} S \{Q\}\ is valid
Example Hoare Triples

{true} X := 12 { X = 12 }

{true} X := 12 { X = 0 }

{true} X := 12 { X > 0 }

{true} X := 12 { true }

{false} X := 12 { X=0 }

{X=i} X := X+1 { X=i+1 }

{true} while true do skip { false }
The extension of an assertion

\[ A^{Iz} \doteq \{ \sigma \in \Sigma_\perp \mid \sigma \models^{Iz} A \} \]
The extension of assertions

Suppose that $\models (P \Rightarrow Q)$

Then for any interpretation $Iz$

$\forall \sigma \in \Sigma_{\bot}. \sigma \models_{Iz} P \Rightarrow \sigma \models_{Iz} Q$

$P^{Iz} \subseteq Q^{Iz}$
The extension of Hoare Triples

Suppose that $\models \{P\}S\{Q\}$

Then for any interpretation $Iz$
$\forall \sigma \in \Sigma_{\perp}. \sigma \models_{Iz} P \Rightarrow S_{ns} \llbracket S \rrbracket \sigma \models_{Iz} Q$

$S_{ns} \llbracket S \rrbracket P_{Iz} \subseteq Q_{Iz}$

$\Sigma_{\perp}$

$C \llbracket c \rrbracket$

$P_{Iz}$

$Q_{Iz}$
Hoare Proof Rules for Partial Correctness (Take 1)

\{A\} \text{skip} \{A\}
\{B[a/X]\} X:=a \{B\}

\{P\} S_0 \{C\} \{C\} S_1 \{Q\}
\{P\} S_0;S_1 \{Q\}

\{P \land b\} S_0 \{Q\} \{P \land \neg b\} S_1 \{Q\}
\{P\} \text{if } b \text{ then } S_0 \text{ else } S_1 \{Q\}

\{\text{Inv} \land b\} S \{\text{Inv}\}
\{\text{Inv}\} \text{ while } b \text{ do } S \{\text{Inv} \land \neg b\}
\models P \Rightarrow P' \{P'\} S \{Q'\} \models Q' \Rightarrow Q
\{P\} S \{Q\}
Simple Example

{X = 5}

Y := X

{Y ≥ 0}
Hoare Proof Rules for Partial Correctness

\{A\} \text{skip} \{A\}
\{B[a/X]\} X:=a \{B\}

\{P\} S_0 \{C\} \{C\} S_1 \{Q\}
\{P\} S_0;S_1 \{Q\}

\{P \land b\} S_0 \{Q\} \{P \land \neg b\} S_1 \{Q\}
\{P\} \text{if } b \text{ then } S_0 \text{ else } S_1 \{Q\}

\{\text{Inv} \land b\} S \{\text{Inv}\}
\{\text{Inv}\} \text{while } b \text{ do } S \{\text{Inv} \land \neg b\}
\models P \Rightarrow P' \{P'\} S \{Q'\} \models Q' \Rightarrow Q
\{P\} S \{Q\}
Simple Example

\{ X = 5 \}

Y := X

\{ Y \geq 0 \}
Example

\{
X = n \land n \geq 0
\}

Y := 1;
\{
X = n \land Y=1 \land n \geq 0
\}

while X > 0 do \{X \geq 0 \land n \geq 0 \land Y=n!/X!\}
\{X > 0 \land n \geq 0 \land Y=n!/X!\}

Y := X \times Y;
\{X > 0 \land n \geq 0 \land Y=n!/X!\}

X := X - 1
\{X > 0 \land n \geq 0 \land Y=n!/X!\}

\{Y = n! \}
Example Formal

1. \( \{ X = n \land n \geq 0 \} \ Y := 1 \ \{ X = n \land Y = 1 \land n \geq 0 \} \)
2. \( \{ X = n \land n \geq 0 \} \ Y := 1 \ \{ X \geq 0 \land n \geq 0 \land Y = n!/X! \} \)
3. \( \{ X > 0 \land n \geq 0 \land Y = n!/X! \} \ Y := X \times Y; \ \{ X > 0 \land n \geq 0 \land Y = n!/(X-1)! \} \)
4. \( \{ X > 0 \land n \geq 0 \land Y = n!/X! \} \ X := X - 1; \ \{ X \geq 0 \land n \geq 0 \land Y = n!/X! \} \)
5. \( \{ X > 0 \land n \geq 0 \land Y = n!/X! \} \ Y := X \times Y; \ X := X - 1 \ \{ X \geq 0 \land n \geq 0 \land Y = n!/X! \} \)
6. \( \{ X \geq 0 \land n \geq 0 \land Y = n!/X! \land X > 0 \} \ Y := X \times Y; \ X := X - 1 \ \{ X \geq 0 \land n \geq 0 \land Y = n!/X! \} \)
7. \( \{ X \geq 0 \land n \geq 0 \land Y = n!/X! \} \ \text{while} \ X > 0 \ \text{do} \ Y := X \times Y; \ X := X - 1 \ \{ X \geq 0 \land n \geq 0 \land Y = n!/X! \land \neg X > 0 \} \)
8. \( \{ X \geq 0 \land n \geq 0 \land Y = n!/X! \} \ \text{while} \ X > 0 \ \text{do} \ Y := X \times Y; \ X := X - 1 \ \{ Y = n! \} \)
9. \( \{ X = n \land n \geq 0 \} \ Y := 1; \ \text{while} \ X > 0 \ \text{do} \ Y := X \times Y; \ X := X - 1 \ \{ Y = n! \} \)
Unsound Proof Rules for Partial Correctness

\[
\begin{align*}
\{A\} \text{ skip } \{B\} \\
\{A\} X:=a \quad \{A[a/X]\} \\
\{P\} S_1 \{C\} \{C\} S_0 \{Q\} \\
\{P\} S_0 ; S_1 \{Q\} \\
\{P \land b\} S_0 \{Q\} \quad \{P \land \neg b\} S_1 \{Q\} \\
\{P\} \text{ if } b \text{ then } S_0 \text{ else } S_1 \{Q\} \\
\{\text{Inv} \land b\} S \{Q\} \\
\{\text{Inv}\} \text{ while } b \text{ do } S \{\text{Q} \land \neg b\} \\
\models P' \implies P \quad \models \{P'\} S \{Q'\} \implies Q \implies Q' \\
\{P\} S \{Q\}
\end{align*}
\]
Incomplete Proof Rules for Partial Correctness

\{A\} \text{skip} \{A\}
\{B[a/X]\} \; X:=a \; \{B\}

\{P\} \; S_0 \; \{P\} \; S_1 \; \{P\}
\{P\} \; S_0;S_1\{P\}

\{P\} \; S_0 \; \{Q\} \; \{P\} \; S_1 \; \{Q\}
\{P\} \text{ if } b \text{ then } S_0 \text{ else } S_1 \{Q\}

\{\text{Inv}\} \; S \; \{\text{Inv}\}
\{\text{Inv}\} \text{ while } b \text{ do } S\{\text{Inv} \land \neg b\}
\models P \Rightarrow P' \; \{P'\} \; S \; \{Q'\} \models Q' \Rightarrow Q
\{P\} \; S \; \{Q\}
Soundness

• Every theorem obtained by the rule system is valid
  – $\vdash \{P\} \subseteq \{Q\} \implies \vdash \{P\} \subseteq \{Q\}$

• The system can be implemented (HOL, LCF, Coq)
  – Requires user assistance

• Proof of soundness
  – Every rule preserves validity (Theorem 6.1)
Soundness of skip axiom

\[ \models \{ A \} \text{skip} \{ A \} \]
Soundness of the assignment axiom

\[ \models \{ B[a/X] \} \ X := a \ \{ B \} \]
Soundness of the sequential composition rule

• Assume that
  \[ \models \{P\} S_0 \{C\} \]
  and
  \[ \models \{C\} S_1 \{Q\} \]

• Show that
  \[ \models \{P\} S_0;S_1 \{Q\} \]
Soundness of the conditional rule

• Assume that
  \[ \models \{ P \land b \} S_0 \{ Q \} \]
  and
  \[ \models \{ P \land \neg b \} S_1 \{ Q \} \]

• Show that
  \[ \models \{ P \} \text{ if } b \text{ then } S_0 \text{ else } S_1 \{ Q \} \]
Soundness of the while rule

• Assume that
  $$\models \{\text{Inv} \land b\} S \{\text{Inv}\}$$

• Show that
  $$\models \{\text{Inv}\} \text{ while } b \text{ do } S \{\text{Inv} \land \neg b\}$$
Soundness of the consequence rule

- Assume that
  \[ \vdash \{ P' \} \models \{ Q' \} \]
  and
  \[ \vdash P \Rightarrow P' \]
  and
  \[ \vdash Q' \Rightarrow Q \]
- Show that
  \[ \vdash \{ P \} \models \{ Q \} \]
(Ideal) Completeness

• Every valid theorem can be proved by the rule system
• For every P and Q such that $\models \{P\} \S \{Q\}$ there exists a proof such $\vdash \{P\} \S \{Q\}$
• But what about Gödel’s incompleteness? $\models \{\text{true}\} \S \{\text{false}\}$
• What does $\models \{\text{true}\} \S \{\text{false}\}$ mean?
Relative Completeness (Chapter 7)

• Assume that every math theorem can be proved
  \( \models \{P\} S \{Q\} \) implies \( \vdash \{P\} S \{Q\} \)
Relative completeness of composition rule

- Prove that \( \{P\} S_0; S_1 \{Q\} \)
- Does there exist an assertion \( I \) such that
  \[ \models \{P\} S_0 \{C\} \]
  and
  \[ \models \{I\} S_1 \{Q\} \]
Precise Hoare Triples

• If \{P\} S \{Q\} and \{P\} S \{R\} then what can we tell?
• The most precise Q such that \{P\}S\{Q\} is called the strongest (liberal) postcondition of S with respect to P
• Denoted by \text{sp}(S, P)
• An imperative program is a predicate transformer
Weakest (Liberal) Precondition

• If \{P\} S \{Q\} and \{R\} S \{Q\} then what can we tell?
• The most general P such that \{P\}S\{Q\} is called the weakest (liberal) precondition of S with respect to Q
• Denoted by wp(S, Q)
• An imperative program is a predicate transformer
Hoare Triples vs. sp vs. wp

• \{P\} S \{Q\} holds when P ⇒ wp(S, Q) and sp(S, p) ⇒ Q
Weakest (Liberal) Precondition

- \( \text{wp}(S, Q) \) – the weakest condition such that every terminating computation of \( S \) results in a state satisfying \( Q \)

- \( \llbracket \text{wp}^l(S, Q) \rrbracket = \{ \sigma \in \Sigma^\downarrow \mid S[\sigma] \sigma \models^l Q \} \)

- [Can employ predicate transformer semantics to formally define the meaning (Chapter 7.5)]

- Prove that \( \{P\} S_0; S_1 \{Q\} \) by proving
  \( \models \{P\} S_0 \{I\} \)
  and
  \( \models \{I\} S_1 \{Q\} \) where \( I = \text{wp}(S_1, Q) \)

- \( \models \{P\} S \{Q\} \) iff for all \( I \models \llbracket P \rrbracket \subseteq \llbracket \text{wp}^l(S, Q) \rrbracket \)

- \( \models \{P\} S \{Q\} \) iff for \( P \Rightarrow \text{wp}(S, Q) \)
The rule of excluded miracle

• \( wp(S, \text{false}) = \)
Proving that programs correct: skip

• \( \text{wp}(\text{skip}, Q) = Q \)

• Example
  – \( \text{wp}(\text{skip}, X^n + Y^n = Z^n) = \)

To prove \( \{P\} \text{ skip } \{Q\} \), show that \( P \Rightarrow Q \)
Proving that programs correct: assignments

- \( \text{wp}(X := a, Q) = Q[a/X] \)

- Examples
  - \( \text{wp}(X := X+1, X=7) = \)
  - \( \text{wp}(X := X+1, X \leq 7) = \)
  - \( \text{wp}(X := 15, X \leq 10) = \)
  - \( \text{wp}(Y := X+3*Y, X \leq 10) = \)

To prove \{P\} X := a \{Q\}, show that \( P \Rightarrow Q[a/X] \)
Proving that programs correct: pointers

• \( wp(*X := a, Q) = \)
Dafny Example

```dax
method foo(x: int) returns (y: int)
requires x > 0;
ensures y > 1;
{
y := x + 1;
}
```

http://rise4fun.com/Dafny
Proving that programs correct: sequential composition

• $wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))$

• Examples

• $wp(y := y + 1; x := x + 3, y \leq 10 \land 3 \leq x) =$

To prove $\{P\} S_1; S_2 \{Q\}$, show that $P \Rightarrow wp(S_1; S_2, Q)$
Proving that programs correct: Conditional Statements

• $wp(\text{if } b \text{ then } S_0 \text{ else } S_1, Q) =$
  
  $b \land wp(S_0, Q) \lor \neg b \land wp(S_1, Q)$

• $b \land wp(S_0, Q)$ is the set of states which path the if condition

• $\neg b \land wp(S_1, Q)$ is the set of states which path the else condition

• $wp(\text{if } x < y \text{ then } z := y \text{ else } z := x, 0 \leq z) =$

• $wp(\text{if } x \neq 10 \text{ then } x := x + 1 \text{ else } x := x + 2, x \leq 10)$
Proving that the programs are correct: Loops

• To prove that \( \{P\} S \{Q\} \) where \( S = \text{while } C \text{ do } S' \text{ do} \):
Example: Array Sum

\{N \geq 0\}
k := 0;
s := 0;
while (k != N) {
s := s + a[k];
k := k + 1
}
\{s = \sum_{1 \leq i \leq N} a[i] \}
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) {
        c := c + k;
        k := k + m;
        m := m + 6;
        n := n + 1;
    }
}
Proving programs correct: Procedures

• Suppose you have a program

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

• Proof strategy:
  – Check that precondition of P is valid at the call
  – Assume that all the variables potentially changed by P are “havoked”
  – Assume that the postcondition of P holds after the call
  – Prove that the postcondition M holds at end of M
  – Technically implemented with wp and “projection”
Proving programs correct: Procedures

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:

```daml
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))
}
```
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;
  }
}
Mutual recursion example

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);
    }
}
Binary Search Example

```plaintext
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;
}
```
Binary Search (2)

```java
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;
}
```
var low, high := 0, a.Length;
while (low < high)
invariant 0 <= low <= high <= a.Length;
invariant forall i :: 0 <= i < a.Length &&
    !(low <= i < high) ==> a[i] != value;
{
    var mid := (low + high) / 2;
    if (a[mid] < value) {
        low := mid + 1; }
    else if (value < a[mid]) {
        high := mid; }
    else {
        return mid; }
}
return -1;
}
Relative Completeness

• For every command $S$ and assertion $B$
  – there exists an assertion $A$, such that
    $A = \text{wp}(S, B)$ (Theorem 7.5)
  – $\vdash \{\text{wp}(S, B)\} S \{B\}$ (Lemma 7.6)

• Theorem 7.7: The proof system is relatively complete
  – $\vdash \{P\} S \{Q\}$ implies $\vdash \{P\} S \{Q\}$
Verification Conditions

• Generate assertions that describe the partial correctness of the program
• Use automatic theorem provers to show partial correctness
• Existing tools ESC/Java, Spec#, Dafny
WP Compound statements

- \( \text{wp} \) (skip, Q) = Q
- \( \text{wp}(X := e, Q) = Q[e / x] \)
- \( \text{wp}(S_1; S_2, Q) = \text{wp} \left[ S_1 \right] (\text{wp} \left[ S_2 \right] (Q)) \)
- \( \text{wp}(\text{if } B \text{ then } S_1 \text{ else } S_2, Q) = (B \land \text{wp}(S_1, Q) \lor (\neg B \land \text{wp}(S_2, Q)) \)
- \( \text{wp}(\text{while } B \text{ do } \{I\} S, Q) = I \)
VC rules

\[ VC_{\text{gen}}(\{P\} S \{Q\}) = P \rightarrow \text{wp}(S, Q) \land \bigwedge VC_{\text{aux}}(S, Q) \]

\[ VC_{\text{aux}}(S, Q) = \{\} \text{ (for any atomic statement)} \]

\[ VC_{\text{aux}}(S_1; S_2, Q) = VC_{\text{aux}}(S_1, \text{wp}(S_2, Q)) \cup VC_{\text{aux}}(S_2, Q) \]

\[ VC_{\text{aux}}(\text{if } C \text{ then } S_1 \text{ else } S_2, Q) = VC_{\text{aux}}(S_1, Q) \cup VC_{\text{aux}}(S_2, Q) \]

\[ VC_{\text{aux}}(\text{while } B \text{ do } \{I\} S, Q) = VC_{\text{aux}}(S, I) \cup \{I \land B \rightarrow \text{wp}(S, I)\} \cup \{I \land \neg B \rightarrow Q\} \]
Soundness and completeness

• The set of rules of Hoare logic is sound: any derivable triple is valid

• Completeness: The set of rules of Hoare logic is relatively complete
  – if the language of state predicates is expressive enough, then any valid Hoare triple \{P\} S \{Q\} can be derived using the rules
Verification Success Stories

• Comcert C compiler verification
• Sel4 Kernel verification
• The IronClad/IronFleet Projects Microsoft Research
  – End-to-end verification of a distributed system with Dafny
Projects with Dafny

• Verify a real algorithm/data structure
• Examples
  • Simplified DPLL without learned clause
    • Two watched literal invariant
  • A dynamic programming algorithm
  • Your own algorithm with complex loops/data structures
Projects with axiomatic verification

• Develop a verifier for a small Python subset
  • Enough to cover few assignments in Mavo
Summary

- Axiomatic semantics provides an abstract semantics
- Can be used to explain programming
- Extensions
  - Procedures
  - Concurrency
  - Events
  - Rely/Guarantee
  - Heaps
- Can be automated
- But requires considerable effort