Theorem Proving
Theorem Proving: Historical Perspective

• Theorem proving (or automated deduction)
  = logical deduction performed by machine

• At the intersection of several areas
  - Mathematics: original motivation and techniques
  - Logic: the framework and the meta-reasoning techniques

• One of the most advanced and technically deep fields of computer science
  - Some results as much as 75 years old
  - Automation efforts are about 40 years old
Applications

• Hardware and software verification (or debugging)

• Automatic program synthesis from specifications

• Discovery of proofs of conjectures
  - A conjecture of Tarski was proved by machine (1996)
  - There are effective geometry theorem provers
Program Verification

• Fact: mechanical verification of software would improve software productivity, reliability, efficiency

• Fact: such systems are still in experimental stage
  - After 40 years!
  - Research has revealed formidable obstacles
  - Many believe that program verification is extremely difficult
Program Verification

• Fact:
  - Verification is done with respect to a specification
  - Is the specification simpler than the program?
  - What if the specification is not right?

• Answer:
  - Developing specifications is hard
  - Still redundancy exposes many bugs as inconsistencies
  - We are interested in partial specifications
    • An index is within bounds, a lock is released
Theorem Proving and Software

Program

Semantics

Validity

Meets spec/Found Bug

VC generation

Provability (theorem proving)

Theorem in a logic

• Soundness:
  - If the theorem is valid then the program meets specification
  - If the theorem is provable then it is valid

Specification
<table>
<thead>
<tr>
<th>Theorem Proving</th>
<th>Program Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start from real code and face head-on issues like:</strong></td>
<td><strong>Most often used for sequential programs</strong></td>
</tr>
<tr>
<td>- aliasing and side-effects</td>
<td>- Ambitious:</td>
</tr>
<tr>
<td>- looping</td>
<td>- Complex properties</td>
</tr>
<tr>
<td>- data types and recursion</td>
<td>- Flow sensitive</td>
</tr>
<tr>
<td>- exceptions</td>
<td>- Inter-procedural</td>
</tr>
<tr>
<td></td>
<td><strong>Modest:</strong></td>
</tr>
<tr>
<td></td>
<td>- Simple properties</td>
</tr>
<tr>
<td></td>
<td>- Flow insensitive</td>
</tr>
<tr>
<td></td>
<td>- Intra-procedural</td>
</tr>
<tr>
<td><strong>Ambitious:</strong></td>
<td><strong>Semi-automatic</strong></td>
</tr>
<tr>
<td>- Complex properties</td>
<td><strong>Automatic</strong></td>
</tr>
<tr>
<td>- Flow sensitive</td>
<td><strong>Requires invariants and validates them</strong></td>
</tr>
<tr>
<td>- Inter-procedural</td>
<td><strong>Discovers simple invariants</strong></td>
</tr>
</tbody>
</table>
## Theorem Proving vs. Type Systems

<table>
<thead>
<tr>
<th>Theorem Proving</th>
<th>Type Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start from real code and face head-on issues like:</td>
<td></td>
</tr>
<tr>
<td>• aliasing and side-effects</td>
<td></td>
</tr>
<tr>
<td>• looping</td>
<td></td>
</tr>
<tr>
<td>• data types and recursion</td>
<td></td>
</tr>
<tr>
<td>• exceptions</td>
<td></td>
</tr>
<tr>
<td>Mostly for sequential programs and some concurrent</td>
<td></td>
</tr>
<tr>
<td><strong>Ambitious:</strong></td>
<td><strong>Fairly Modest:</strong></td>
</tr>
<tr>
<td>• Complex properties (fairly arbitrary predicates over state space)</td>
<td>- Verify type properties where loops, aliases, exceptions etc do not cause big problems</td>
</tr>
<tr>
<td>• Flow sensitive</td>
<td>- Flow insensitive</td>
</tr>
<tr>
<td>• Inter-procedural</td>
<td>- Inter-procedural</td>
</tr>
<tr>
<td><strong>Semi-automatic</strong></td>
<td><strong>Semi-automatic</strong></td>
</tr>
<tr>
<td>Requires invariants and validates them</td>
<td>Requires invariants and validates them</td>
</tr>
</tbody>
</table>
Overview of the Next Few Lectures

• Focus
  
  *Expose basic theorem proving techniques useful for software debugging*

• From programs to theorems
  - Verification condition generation

• From theorems to proofs
  - Theorem provers
  - Decision procedures

• Applications
• **Consists of:**
  - A language for writing specifications about programs
  - Rules for establishing when specifications hold

• **Typical specifications:**
  - During the execution, only non-null pointers are dereferenced
  - This program terminates with \( x = 0 \)

• **Partial vs. total correctness specifications**
  - Safety vs. liveness properties
  - Usually focus on safety (partial correctness)
Specification Languages

• **Must be easy to use and expressive (conflicting needs)**
  - Most often only expressive 😞

• **Typically they are extensions of first-order logic**
  - Although higher-order or modal logics are also used

• **We focus here on state-based specifications (safety)**
  
  State = values of variables + contents of heap (+ past state)
  - Not allowed: “variable x is live”, “lock L will be released”, “there is no correlation between the values of x and y”
A Specification Language

• We’ll use a fragment of first-order logic:
  Formulas \( P ::= A | \text{true} | \text{false} | P_1 \land P_2 | P_1 \lor P_2 | \neg P | \forall x.P \)
  
  Atoms \( A ::= E_1 \leq E_2 | E_1 = E_2 | f(A_1,\ldots,A_n) | \ldots \)
  
  – All boolean expressions from our language are atoms

• Can have an arbitrary collection of predicate symbols
  
  – \text{reachable}(E_1,E_2) - list cell \( E_2 \) is reachable from \( E_1 \)
  
  – \text{sorted}(a, L, H) - array \( a \) is sorted between \( L \) and \( H \)
  
  – \text{ptr}(E,T) - expression \( E \) denotes a pointer to \( T \)
  
  – \( E : \text{ptr}(T) \) - same in a different notation

• An assertion can hold or not in a given state
  
  – Equivalently, an assertion denotes a set of states
Hoare Triples

• Partial correctness: \{ P \} s \{ Q \}
  - When you start s in any state that satisfies P
  - If the execution of s terminates
  - It does so in a state that satisfies Q

• Total correctness: [ P ] s [ Q ]
  - When you start s in any state that satisfies P
  - The execution of s terminates and
  - It does so in a state that satisfies Q

• Defined inductively on the structure of statements
Hoare Rules

\[
\begin{align*}
\{P\} & \quad s_1 \quad \{Q\} \quad \{Q\} \quad s_2 \quad \{R\} \\
\hline
\{P\} & \quad s_1; \quad s_2 \quad \{R\}
\end{align*}
\]

\[
\begin{align*}
\{P_1\} & \quad s_1 \quad \{Q\} \quad \{P_2\} \quad s_2 \quad \{Q\} \\
\hline
\{E \implies P_1 \land \neg E \implies P_2\} & \quad \text{if } E \text{ then } s_1 \text{ else } s_2 \quad \{Q\}
\end{align*}
\]

\[
\begin{align*}
\{I \land E\} & \quad s \quad \{I\} \\
\hline
\{I\} & \quad \text{while } E \text{ do } s \quad \{Q\}
\end{align*}
\]
Hoare Rules: Assignment

• Example: \( \{P\} x := x + 2 \{x \geq 5\} \). What is \( P \)?
• General rule:

\[
\{ Q[E/x] \} x := E \{Q\}
\]

• Surprising how simple the rule is!
• The key is to compute “backwards” the precondition from the postcondition
• Before Hoare:

\[
\{ P \} x := E \{ \% x'. P[x'/x] \& \& x = E[x'/x] \}
\]
Hoare Rules: Examples

• What is \( P \) in

- \( \{ P \} \ x := x + y \ \{ x = 0 \} \)
- \( \{ P \} \ y := 0 \ \{ y = 0 \} \)
- \( \{ P \} \ *x := 5 \ \{ *x + *y = 10 \} \)
Hoare Rules: Assignment

• But now try:
  \[ \{ P \} *x := 5 \{ *x + *y = 10 \} \]

• \( P \) ought to be “\( *y = 5 \) or \( x = y \)”

• The Hoare rule would give us:
  \[ (*x + *y = 10)[5/*x] \]
  \[ = 5 + *y = 10 \]
  \[ = *y = 5 \] (we lost one case)

• How come the rule does not work?
Hoare Rules: Assignment

• But now try:

\{ P \} \ast x := 5 \{ \ast x + \ast y = 12 \}

• \( P \) ought to be "\( \ast y = 7 \) and \( x \neq y \)"

• The Hoare rule would give us:

\[(\ast x + \ast y = 12)[5/\ast x]\]

\[= 5 + \ast y = 12\]

\[= \ast y = 7 \quad \text{(wrong)}\]

• How come the rule does not work?
Handling Program State

• Hoare rules assume absence of aliases
• But real programs have aliases!

• Important technique #1: Model the state of heap as a symbolic mapping from addresses to values

- If $E$ denotes an address and $\mu$ a heap state then:
  - $\text{sel}(\mu, E)$ denotes the contents of memory cell
  - $\text{upd}(\mu, E, V)$ denotes a new heap state obtained from $\mu$ by writing $V$ at address $E$
More on Memory

- We allow variables to range over heap states
  - So we can quantify over all possible heap states

- And we use the special pseudo-variable $\mu$ in assertions to refer to the current state of the heap

- Example:

  \[
  \forall i. i \geq 0 \land i < 5 \Rightarrow \text{sel}(\mu, A + i) > 0 \]
  \[
  \text{allpositive}(\mu, A, 0, 5)
  \]

  says that entries $0..4$ in array $A$ are positive
Hoare Rules: Side-Effects

- To correctly model writes we use memory expressions
  - A memory write changes the value of memory

\[
\{ Q[\text{\textup{upd}(\mu, E_1, E_2)}/\mu] \} *E_1 := E_2 \{Q\}
\]

- Reason later about memory expressions with inference rules such as (McCarthy):

\[
\text{sel}(\text{\textup{upd}(\mu, E_1, E_2), E_3}) = \begin{cases} 
E_2 & \text{if } E_1 = E_3 \\
\text{sel}(\mu, E_3) & \text{if } E_1 \neq E_3
\end{cases}
\]
Memory Aliasing

• Consider again: \( \{ A \} \times := 5 \{ \times + \gamma = 10 \} \)

• We obtain:
  \[
  A = (\times + \gamma = 10)[\text{upd}(\mu, \times, 5)/\mu] = \text{sel}(\text{upd}(\mu, \times, 5), \times) + \text{sel}(\text{upd}(\mu, \times, 5), \gamma) = 10
  \]
  \[
  = 5 + \text{sel}(\text{upd}(\mu, \times, 5), \gamma) = 10
  \]
  \[
  = \text{if } x = y \text{ then } 5 + 5 = 10 \text{ else } 5 + \text{sel}(\mu, \gamma) = 10
  \]
  \[
  = x = y \text{ or } x = 5
  \]

• To (*) is theorem generation

• From (*) to (**) is theorem proving
Memory Aliasing

• Consider again: \{ A \} *x := 5 \{ *x + *y = 12 \}
• We obtain:

\[
A = (\ast x + \ast y = 12)[\text{upd}(\mu, x, 5)/\mu]
= (\text{sel}(\mu, x) + \text{sel}(\mu, y) = 12)[\text{upd}(\mu, x, 5)/\mu]
= \text{sel}(\text{upd}(\mu, x, 5), x) + \text{sel}(\text{upd}(\mu, x, 5), y) = 12 \quad (*)
= 5 + \text{sel}(\text{upd}(\mu, x, 5), y) = 12
= \text{if } x = y \text{ then } 5 + 5 = 12 \text{ else } 5 + \text{sel}(\mu, y) = 12
= \text{if } x = y \text{ then } \text{false} \text{ else } 5 + \text{sel}(\mu, y) = 12
= x \neq y \text{ and } \ast y = 7 \quad (**)

• To (*) is theorem generation
• From (*) to (**) is theorem proving
Alternative Handling for Memory

- Reasoning about aliasing can be expensive (NP-hard)

- Sometimes completeness is sacrificed with the following (approximate) rule:

\[ \text{sel}(\text{upd}(\mu, E_1, E_2), E_3) = \begin{cases} 
E_2 & \text{if } E_1 = \text{(obviously) } E_3 \\
\text{sel}(\mu, E_3) & \text{if } E_1 \neq \text{(obviously) } E_3 \\
p & \text{otherwise (p is a fresh new parameter)}
\end{cases} \]

- The meaning of “obvious” varies:
  - The addresses of two distinct globals are \(\neq\)
  - The address of a global and one of a local are \(\neq\)
  - “PREfix” and GCC use such schemes
Hoare Rules: Examples

• Consider

  - \( \{ x = 2 \} \quad x := x + 1 \quad \{ x < 5 \} \)
  
  - \( \{ x < 2 \} \quad x := x + 1 \quad \{ x < 5 \} \)
  
  - \( \{ x < 4 \} \quad x := x + 1 \quad \{ x < 5 \} \)

• They all have correct preconditions
• But the last one is the most general (or weakest) precondition
Dijkstra’s Weakest Preconditions

• Consider \( \{ P \} s \{ Q \} \)
• Predicates form a lattice:

\[
\begin{array}{ccc}
\text{false} & \uparrow & \text{true} \\
\text{valid preconditions} & \uparrow & \text{weak} \\
\text{strong} & \uparrow & \text{weakest} \\
\end{array}
\]

precondition: \( WP(s, Q) \)

• To verify \( \{ P \} s \{ Q \} \)
• compute \( WP(s, Q) \) and prove \( P \models WP(s, Q) \)
Weakest Preconditions

• Computed by a backward reading of Hoare rules

\[
\frac{\{P\} \, s_1 \, \{Q\} \quad \{Q\} \, s_2 \, \{R\}}{\{P\} \, s_1; \, s_2 \, \{R\}}
\]

\[
- \text{wp}(s_1; \, s_2, \, R) = \text{wp}(s_1, \, \text{wp}(s_2, \, R))
\]
Weakest Preconditions

• Computed by a backward reading of Hoare rules

\[
\{Q[E/x]\} x := E \{Q\}
\]

- \(wp(x := E, Q) = Q[E/x]\)
Weakest Preconditions

• Computed by a backward reading of Hoare rules

\[
\begin{array}{c}
\{ P_1 \} \; s_1 \; \{ Q \} \; \quad \{ P_2 \} \; s_2 \; \{ Q \} \\
\hline
\{ E \vdash P_1 \land \neg E \vdash P_2 \} \; \text{if} \; E \; \text{then} \; s_1 \; \text{else} \; s_2 \; \{ Q \}
\end{array}
\]

\[ wp(\text{if} \; E \; \text{then} \; s_1 \; \text{else} \; s_2, \; Q) = (E \vdash wp(s_1, \; Q)) \land (\neg E \vdash wp(s_2, \; Q)) \]
Weakest Preconditions: Exercises

• \( WP(\ y++; x++, \ x=y )? \)

• \( WP(\ if \ x=y \ then \ x := x \ else \ x := y, \ x=y )? \)

• \( WP(\ while \ x>0 \ do \ x--, \ x=0 )? \)
Weakest Preconditions (Cont.)

• What about loops?
• Define a family of WPs
  – \( WP_k(\text{while } E \text{ do } S, Q) \) = weakest precondition on which
    • if the loop terminates in \( k \) or fewer iterations,
    • it terminates in \( Q \)

\[
WP_0 = (\neg E \Rightarrow Q) \\
WP_1 = (E \Rightarrow WP(s, WP_0)) \land (\neg E \Rightarrow Q)
\]

\[
WP_{\geq 0} = WP_0 \land WP_1 \land \ldots
\]

• \( WP(\text{while } E \text{ do } S, B) = K_{k=0} WP_k \)
  - Kind of hard to compute
  - Can we find something easier yet sufficient?
Not Quite Weakest Preconditions

• Recall what we are trying to do:

\[ \begin{align*}
\text{false} & \quad \uparrow \quad \uparrow \\
\text{valid preconditions} & \quad \uparrow \\
\text{strong} & \quad \uparrow \\
\text{P} & \quad \text{precondition: } WP(s, Q) \\
\text{verification condition: } VC(s, Q) & \\
\text{weak} & \quad \uparrow \\
\text{true} & \\
\end{align*} \]

• We shall construct a verification condition: \( VC(s, Q) \)
  - require loops annotated with loop invariants!
  - \( VC \) is guaranteed stronger than \( WP \)
  - But hopefully still weaker than \( P: P \vdash VC(s, Q) \vdash WP(s, Q) \)
Verification Condition Generation

- Computed in a manner similar to WP
- Except the rule for while:

\[ VC(\text{while}_I E \text{ do } s, Q) = I \land (\forall x_1...x_n. I \Rightarrow (E \Rightarrow VC(s, I) \land \neg E \Rightarrow Q)) \]

- \( I \) is the loop invariant (provided externally)
- \( x_1, ..., x_n \) are all the variables modified in \( s \)
- The \( \forall \) is similar to the \( \forall \) in mathematical induction:

\[ P(0) \land \forall n \in \mathbb{N}. P(n) \Rightarrow P(n+1) \]
Next Thursday

- Arnaud Venet and Guillaume Brat on NASA's *C Global Surveyor* in Crown College, room 105, 2-3:45pm.

- In class: How to *prove* verification conditions.

- Reading:
  - *Guarded commands, nondeterminacy and formal derivation of programs.*
  - *Chaff: Engineering an Efficient SAT Solver.*

- Homework 2 due

- Presentations
  - some suggestions on web page
  - other ideas are welcome