Theorem Proving

CS 294-8
Lecture 9

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

- Software/hardware productivity tools
  - Hardware and software verification (or debugging)
  - Security protocol checking
- Automatic program synthesis from specifications
- Discovery of proofs of conjectures
  - A conjecture of Tani 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 dead

- Myth:
  - "Think of the peace of mind you will have when the verifier finally says "Verified", and you can relax in the mathematical certainty that no more errors exist"

- Answer:
  - Use instead to find bugs (like more powerful type checkers)
  - We should change "verified" to "Sorry, I can't find more bugs"

- Fact:
  - Many logical theories are undecidable or decidable by super-exponential algorithms
  - There are theorems with super-exponential proofs

- Answer:
  - Such limits apply to human proof discovery as well
  - If the smallest correctness argument of program P is huge then how did the programmer find it?
  - Theorems arising in PV are usually shallow but tedious
Program Verification

Opinion:
- Mathematicians do not use formal methods to develop proofs
- Why then should we try to verify programs formally?

Answer:
- In programming, we are often lacking an effective formal framework for describing and checking results
- Compare the statements
  - The area bounded by $y=0$, $x=1$ and $y=x^2$ is $1/3$
  - By sliding two circular lists we obtain another circular list with the union of the elements

---

Theorem Proving and Software

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

---

Overview of the Next Few Lectures

- Focus
  - Expose basic techniques useful for software debugging

- From programs to theorems
  - Verification condition generation

- From theorems to proofs
  - Theorem provers
  - Decision procedures

- Applications
  - Combining program analysis and decision procedures
Programs → Theorems. Axiomatic Semantics

- Consists of:
  - A language for making assertions about programs
  - Rules for establishing when assertions hold
- Typical assertions:
  - During the execution, only non-null pointers are dereferenced
  - This program terminates with $x = 0$
- Partial vs. total correctness assertions
  - Safety vs. liveness properties
  - Usually focus on safety (partial correctness)

Assertion or 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
- Semantics given in the context of the underlying logic
- We focus here on state-based assertions (for safety)
  - State = values of variables + contents of memory (+ past state)
  - Not allowed: "variable $x$ is live", "lock $L$ will be released"
  - "There is no correlation between the values of $x$ and $y$"

An Assertion Language

- We'll use a fragment of first-order logic:
  - $ \forall x \exists y \exists z \ldots (P(x, y, z, \ldots))$
  - $P(x, y, z, \ldots)$
  - $x, y, z, \ldots$ are variables
  - $P$ is a predicate
  - $\forall x$ means for all $x$
  - $\exists y$ means there exists a $y$
- All boolean expressions from our language are atomic
- Can have an arbitrary collection of function symbols
  - reachable($E_1$,$E_2$) - list cell $E_2$ is reachable from $E_1$
  - sort($A$, $L$, $H$) - 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

Handling Program State

- We cannot have side-effects in assertions
  - While creating the theorem we must remove side-effects!
  - But how do that when lacking precise aliasing information?
- Important technique #1: Postpone alias analysis
  - Model the state of memory as a symbolic mapping from addresses to values:
    - If $E$ denotes an address and $M$ a memory state then:
      - $\text{sel}(M, E)$ denotes the contents of memory cell $E$
      - $\text{upd}(M, E, V)$ denotes a new memory state obtained from $M$ by writing $V$ at address $E$

More on Memory

- We allow variables to range over memory states
  - So we can quantify over all possible memory states
- And we use the special pseudo-variable $\mu$ in assertions to refer to the current state of memory

- Example:
  - $\forall i \exists j : i \times j = \text{positive}(j, A) = 0$ - an expression in array $A$ are positive
- Defined inductively on the structure of statements

Hoare Triples

- Partial correctness: $\{ A \} s \{ B \}$
  - When you start $s$ in any state that satisfies $A$
  - $s$ the execution of $s$ terminates
  - It does as in a state that satisfies $B$
- Total correctness: $\{ A \} s \{ B \}$
  - When you start $s$ in any state that satisfies $A$
  - The execution of $s$ terminates and
  - It does as in a state that satisfies $B$
Hoare Rules

\begin{align*}
(A) \quad & \quad (C) \\
\frac{(A) s_1 (C)}{(A) s_2 (B)} \\
\frac{(A) s_1 (B)}{(A) \land \neg E \Rightarrow A) \quad \text{if } E \text{ then } s_1 \text{ else } s_2 (B)}
\end{align*}

\begin{align*}
(A) \quad & \quad (A) \quad (\land) \quad (\implies)
\frac{(E \land \neg E \Rightarrow A) \quad \text{if } E \text{ then } s_1 \text{ else } s_2 (B)}{(E \land \neg E \Rightarrow B)}
\end{align*}

\begin{align*}
(A) \quad & \quad (A) \quad (\land) \quad (\implies)
\frac{(E \land \neg E \Rightarrow A) \quad \text{if } E \text{ then } s_1 \text{ else } s_2 (B)}{(E \land \neg E \Rightarrow B)}
\end{align*}

Example: \{ A \} \ x := x + 2 \{ x := 5 \}. What is A?

General rule:

\[ (\forall x \in X) x := E (\forall x \in X) \]

Surprising how simple the rule is!

The key is to compute "backwards" the precondition from the postcondition.

Before Hoare:

\[ (A) \quad \text{if } E \text{ then } s_1 \text{ else } s_2 (B) \]

Hoare Rules: Assignment

But now try:

\[ \{ A \} \ x := y \land \ y = 5 \{ x \land x = 5 \} \]

\[ A \text{ ought to be } \"y = 5 \text{ or } x = y\" \]

The Hoare rule would give us:

\[ (* \land y = 5) \land (5 \land x = 5) \]

\[ = 5 \land y = 5 \land x = 5 \]

\[ (\text{we lost one case}) \]

How come the rule does not work?

Memory Aliasing

Consider again: \( \{ A \} \ x := 5 \{ x \land x \land y = 10 \} \)

We obtain:

\[ A = (* \land x = 10) \land (\text{upd}(x, 5)) \]

\[ = (\text{seq}(x, x) \land x \land y) = 10 \land (\text{upd}(x, 5)) \]

\[ = \text{seq}(\text{upd}(x, 5), x, x) \land x \land y = 10 \]

\[ = \text{if } x = y \text{ then } 5 + 5 = 10 \text{ else } 5 \land x \land y = 10 \]

\[ = x \land y \land x = y \]

\[ = 5 \land x \land y \land x = y \]

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:

\[ \text{seq}(\text{upd}(x, 5), x, x) \land x \land y = 10 \]

\[ = \text{if } x = y \text{ then } 5 + 5 = 10 \text{ else } 5 \land x \land y = 10 \]

\[ = x \land y \land x = y \]

\[ = 5 \land x \land y \land x = y \]

\[ = (**) \]

The meaning of "obvious" varies:

\[ \{ A \} \ x := x + 2 \{ x := 5 \}. \]

\[ \{ A \} \ x \land (\text{global}) \land (\text{local}) \land (\text{fresh}) \]

\[ \{ A \} \ x \land (\text{global}) \land (\text{local}) \land (\text{fresh}) \]

\[ \{ A \} \ x \land (\text{global}) \land (\text{local}) \land (\text{fresh}) \]

\[ \{ A \} \ x \land (\text{global}) \land (\text{local}) \land (\text{fresh}) \]
Weakest Preconditions

- Dijkstra's idea: To verify that \( \{ A \} \implies \{ B \} \)
  a) Find out all predicates \( A' \) such that \( \{ A' \} \implies \{ B \} \)
  b) Verify for one \( A' = \text{Pre}(s, B) \) that \( A \implies A' \)
- Predicates form a lattice:
  \[
  \begin{array}{cc}
  \text{false} & \implies & \text{true} \\
  \text{strong} & \implies & \text{weak} \\
  A & \implies & \text{weakest precondition: } WP(s, B) \\
  \end{array}
  \]
- Thus: compute \( WP(s, B) \) and prove \( A \implies WP(s, B) \)

Theorem Proving and Program Analysis (again)

- Predicates form a lattice:
  \( WP(s, B) = \text{lub}_P(\text{Pre}(s, B)) \)
- This is not obvious at all:
  - \( \text{lub}(P, P_2) = P_2 \)
  - \( \text{lub}(P \lor P_2, P) \)
  - But can we always write this with a finite number of \( \lor \)'
- Even checking implication can be quite hard
- Compare with program analysis in which lattices are:
  - Finite height and quite simple
  - Program verification is program analysis on the lattice of first order formulas

Weakest Preconditions (Cont.)

- What about loops?
  - Define a family of \( WP \)
    - \( WP(\text{while } E \implies S, B) = \text{weakest precondition on which the loop } \)
    - If it terminates in \( k \) or fewer iterations, it terminates in \( B \)
    - \( WP, E \implies WP(S, WP) \land -E \implies B \)
  - \( WP(\text{while } E \implies S, B) = \land_{k \geq 0} WP_k = \text{lub}(WP_k | k \geq 0) \)
  - Kind of hard to compute
  - Can we find something easier yet sufficient?

Not Quite Weakest Preconditions

- Recall what we are trying to do:
  \[
  \begin{array}{cc}
  \text{false} & \implies & \text{true} \\
  \text{strong} & \implies & \text{weak} \\
  A & \implies & \text{weakest precondition: } WQ(s, B) \\
  \end{array}
  \]
- We shall construct a verification condition: \( VQ(s, B) \)
  - The loops are annotated with loop invariants
  - \( VQ \) is guaranteed stronger than \( WP \)
  - But hopefully still weaker than \( A \): \( A \implies VQ(s, B) \implies WP(s, B) \)

Invariants Are Not Easy

- Consider the following code from QuickSort
  \[
  \begin{aligned}
  &\text{partition}(int *a, int L, int H, int *pivot) \{
  &\text{int } L = L; H = H; \\
  &\text{while } (L < H) \{
  &\text{while } (L < pivot) \{ L++; \\
  &\text{while } (H > pivot) \{ H--; \\
  &\text{if } (L > H) \{ \text{swap } (L) \text{ and } (H) \} \\
  &\text{return } L \\
  &\}
  &\}
  &\}
  &\}
  \end{aligned}
  \]
- Consider verifying only memory safety
- What is the loop invariant for the outer loop?