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 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 dead

Program Verification

- 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`"

Program Verification

- 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

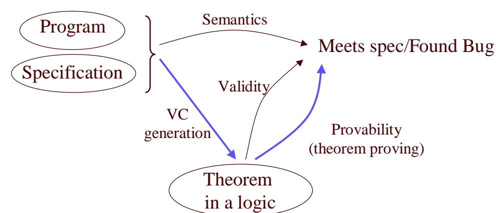
Program Verification

- 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 splicing two circular lists we obtain another circular list with the union of the elements

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



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

Theorem Proving	Program Analysis
Start from real code and face head-on issues like: • aliasing and side-effects • looping • data types and recursion • exceptions	
Most often used for sequential programs	
Ambitious: – Complex properties – Flow sensitive – Inter-procedural	Modest: – Simple properties – Flow insensitive – Intra-procedural
Semi-automatic	Automatic
Requires invariants and validates them	Discovers simple invariants

Theorem Proving	Model Checking
Operate on models of the software and reduce the checking problem to a decision problem in a logic	
Ambitious: – Starts from real code – Inter-procedural – Complex properties	More modest: – Models are heavily abstracted – Function calls, data structures are abstracted – Simpler properties (typically)
Model generated automatically	Model generated manually
Mostly sequential programs	Also concurrent programs
Requires invariants and validates them	Discovers invariants

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:

  Formulas $P ::= A \mid T \mid \bot \mid P_1 \wedge P_2 \mid \forall x.P \mid P_1 \Rightarrow P_2$

  Atoms $A ::= E_1 \leq E_2 \mid E_1 = E_2 \mid f(A_1,\dots,A_n) \mid \dots$
  - All boolean expressions from our language are atoms
- Can have an arbitrary collection of function symbols
  - $\text{reachable}(E_1,E_2)$ - list cell E_2 is reachable from E_1
  - sorted(A, L, H) - array A is sorted between L and H
  - ptr(E,T) - expression E denotes a pointer to T
  - E : 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 to 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:
  - sel(M,E) denotes the contents of memory cell
  - 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 μ in assertions to refer to the current state of memory
- Example:

  "$\forall i.\ i \geq 0 \wedge i < 5 \Rightarrow \text{sel}(\mu, A + i) > 0$" = $\text{allpositive}(\mu, A, 0, 5)$

  says that entries 0..4 in array A are positive

Hoare Triples

- Partial correctness: { A } s { B}
  - When you start s in any state that satisfies A
  - If the execution of s terminates
  - It does so 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 so in a state that satisfies B
- Defined inductively on the structure of statements

Hoare Rules

$$\frac{\{A\}\ s_1\ \{C\} \qquad \{C\}\ s_2\ \{B\}}{\{A\}\ s_1;\ s_2\ \{B\}}$$

$$\frac{\{A\}\ s_1\ \{B\} \qquad \{A'\}\ s_2\ \{B\}}{\{E \Rightarrow A \wedge \neg E \Rightarrow A'\}\ \text{if } E \text{ then } s_1 \text{ else } s_2\ \{B\}}$$

$$\frac{\{I \wedge E\}\ s\ \{I\} \qquad I \wedge \neg E \Rightarrow B}{\{I\}\ \text{while } E \text{ do } s\ \{B\}}$$

Hoare Rules: Assignment

- Example: { A } x := x + 2 {x >= 5 }. What is A?
- General rule:

$$\frac{}{\{B[E/x]\}\ x := E\ \{B\}}$$

- Surprising how simple the rule is !
- The key is to compute "backwards" the precondition from the postcondition
- Before Hoare:

$$\frac{}{\{A\}\ x := E\ \{\exists x'.\ A[x'/x] \wedge x = E[x'/x]\}}$$

Hoare Rules: Assignment

- But now try:
  { A } *x = 5 { *x + *y = 10 }
- A 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: Side-Effects

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

$$\frac{}{\{B[upd(\mu, E_1, E_2)/\mu]\}\ {*E_1} := E_2\ \{B\}}$$

- Important technique #1a: treat memory as a whole
- And (#1b) reason later about memory expressions with inference rules such as (McCarthy):

$$sel(upd(M, E_1, E_2), E_3) = \begin{cases} E_2 & \text{if } E_1 = E_3 \\ sel(M, E_3) & \text{if } E_1 \neq E_3 \end{cases}$$

Memory Aliasing

- Consider again: { A } *x = 5 { *x + *y = 10 }
- We obtain:
  A = (*x + *y = 10)[upd(μ, x, 5)/μ]
  = (sel(μ, x) + sel(μ, y) = 10) [upd(μ, x, 5)/μ]
  = sel(upd(μ, x, 5), x) + sel(upd(μ, x, 5), y) = 10 (*)
  = 5 + sel(upd(μ, x, 5), y) = 10
  = if x = y then 5 + 5 = 10 else 5 + sel(μ, y) = 10
  = x = y or *y = 5 (**)
- To (*) is theorem generation
- From (*) to (**) is theorem proving

Alternative Handling for Memory

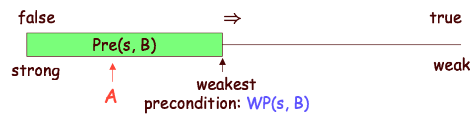
- Reasoning about aliasing can be expensive (NP-hard)
- Sometimes completeness is sacrificed with the following (approximate) rule:

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

- The meaning of "obvious" varies:
  - The addresses of two distinct globals are ≠
  - The address of a global and one of a local are ≠
- "PREfix" and GCC use such schemes

Weakest Preconditions

- Dijkstra's idea: To verify that { A } s { B}
  a) Find out all predicates A' such that { A'} s { B}
    - call this set Pre(s, B)
  b) Verify for one A' ∈ Pre(s, B) that A ⇒ A'
- Predicates form a lattice:



- Thus: compute WP(s, B) and prove A ⇒ WP(s, B)

Theorem Proving and Program Analysis (again)

- Predicates form a lattice:
  $WP(s, B) = lub_{\Rightarrow}(Pre(s, B))$
- This is not obvious at all:
  - $lub \{P_1, P_2\} = P_1 \vee P_2$
  - $lub\ PS = \vee_{P \in PS} P$
  - But can we always write this with a finite number of ∨ ?
- Even checking implication can be quite hard
- Compare with program analysis in which lattices are of finite height and quite simple

  Program Verification is Program Analysis on the lattice of first order formulas

Weakest Preconditions

- Computed by a backward reading of Hoare rules

$$\frac{\{A\}\ s_1\ \{C\} \qquad \{C\}\ s_2\ \{B\}}{\{\ A\ \}\ s_1;\ s_2\ \{B\}}$$

  - $wp(s_1;\ s_2,\ B) = wp(s_1,\ wp(s_2,\ B))$

$$\frac{}{\{\ B[E/x]\ \}\ x := E\ \{B\}}$$

  - $wp(x := E,\ B) = B[E/x]$

$$\frac{\{A\}\ s_1\ \{B\} \qquad \{A'\}\ s_2\ \{B\}}{\{\ E \Rightarrow A \wedge \neg E \Rightarrow A'\}\ \text{if } E \text{ then } s_1 \text{ else } s_2\ \{B\}}$$

  - $wp(\text{if } E \text{ then } s_1 \text{ else } s_2,\ B) = E \Rightarrow wp(s_1,\ B) \wedge \neg E \Rightarrow wp(s_2,\ B)$

Weakest Preconditions (Cont.)

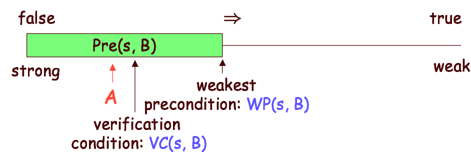
- What about loops?
- Define a family of WPs
  - WP_k(while E do S, B) = weakest precondition on which the loop <u>if</u> it terminates in k or fewer iterations, it terminates in B

  $WP_0 = \neg E \Rightarrow B$
  $WP_1 = E \Rightarrow WP(S, WP_0) \wedge \neg E \Rightarrow B$
  …

- $WP(\text{while } E \text{ do } S,\ B) = \wedge_{k \geq 0} WP_k = lub\ \{WP_k \mid k \geq 0\}$
  - Kind of hard to compute
  - Can we find something easier yet sufficient ?

Not Quite Weakest Preconditions

- Recall what we are trying to do:



- We shall construct a <u>verification condition</u>: VC(s, B)
  - The loops are annotated with loop invariants !
  - VC is guaranteed stronger than WP
  - But hopefully still weaker than A: A ⇒ VC(s, B) ⇒ WP(s, B)

Invariants Are Not Easy

- Consider the following code from QuickSort

```
int partition(int *a, int L0, int H0, int pivot) {
    int L = L0, H = H0;
    while(L < H) {
        while(a[L] < pivot) L ++;
        while(a[H] > pivot) H --;
        if(L < H) { swap a[L] and a[H] }
    }
    return L
}
```

- Consider verifying only memory safety
- What is the loop invariant for the outer loop ?