Techniques for Automated Deduction

CS 294-3
Lecture 1

Course Administration

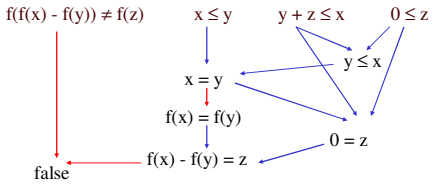
- Why this course?
- Please write down your name, email address
- Time: Tuesday, Thursday 12:30-2:00pm
- Office hours: Tuesday 2-3, and by appointment
- Web page:
  http://www.cs.berkeley.edu/~necula/autded

Coursework

- Attend lectures
  - Occasionally help by scribing lecture notes
- Course Project …
- Prepare a lecture based on a couple of papers
  - Modal logics (temporal, linear, belief), BDDs, SAT-based proving
- Please register
  - For grade: must do project
  - For P/F: no project
- Course **cannot** be used for software breadth req.

Course Project

- Develop an automatic theorem prover
  - Use Nelson-Oppen cooperating decision procedures
  - We'll be able to mix-and-match decision procedures
  - Example: equality + arithmetic. Prove the unsatisfiability of:



Course Project (II)

- I will provide the core of the theorem prover
- Each group develops a decision procedure
  - Example: arithmetic, equality, typing, randomized, etc.
  - Range from 400 lines to 2000 lines
  - Groups of 2-3, several separate provers
- In Objective CAML (dialect of ML)
  - Will provide infrastructure (pretty printing, etc.)
- Test cases:
  - from BLAST

Automated Deduction: Historical Perspective

- Automated deduction
  = logical deduction performed by machine
  - As simple as type checking or as complex as proving mathematical conjectures
- 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

History

- Program verification is almost as old as programming (e.g., "Checking a Large Routine", Turing 1949)
- In the late '60s, Floyd had rules for flow-charts and Hoare for structured languages
- Since then, there have been axiomatic semantics for substantial languages, and many applications

Applications

- Software/hardware productivity tools
  - Hardware and software verification (i.e. debugging)
  - Security protocol checking
  - ➢ An extension of type checkers
- Automatic program synthesis from specifications
  - Using formal methods to select components from a library
- Discovery of proofs of conjectures
  - A conjecture of Tarski was proved by machine (1996)
  - There are effective geometry theorem provers

Hoare Said

- "Thus the practice of proving programs would seem to lead to solution of three of the most pressing problems in software and programming, namely, reliability, documentation, and compatibility. However, program proving, certainly at present, will be difficult even for programmers of high caliber; and may be applicable only to quite simple program designs."

C.A.R Hoare,
"An Axiomatic Basis for Computer Programming",
1969

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 believed that program verification was dead
  - In the last 5 years we have seen renewed interest

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:
  - This is not the purpose of PV.
  - We use PV to find bugs,
  - We should change "`verified`" to "`Sorry, I can't find more bugs`"
  - Just like we use type-checkers
  - Think of PV and stronger (and harder) type checking

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

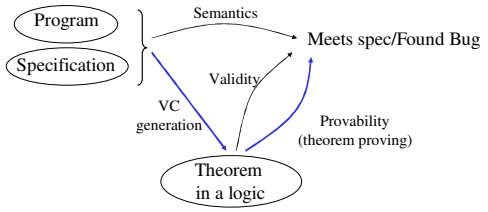
Program Verification

- Opinion:
  - Mathematicians do not use formal methods to develop proofs
    - Correctness of a theorem is established by a social process
  - 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
  - But redundancy exposes many bugs as inconsistencies
  - We are often interested in partial specifications
    - An index is within bounds, a lock is released, etc.

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 meanings
  - Operational semantics
- From programs to theorems
  - Axiomatic semantics
  - Weakest preconditions, verification condition generation
- From theorems to proofs
  - Theorem provers
  - Decision procedures
- Applications
  - Combining program analysis and decision procedures

Course Overview (II)

- We will discuss fundamentals of logic
  - Propositional calculus
    - Syntax
    - Semantics
    - Deduction systems
    - Automated proof methods
  - (maybe) Variations: classical, intuitionistic, modal
  - First-order logic
    - Same structure
- And we will discuss theories + decision procedures
  - Arithmetic, equality, arrays, linked data structures

Course Overview (III)

- For all proof methods we will explore strategies for proof generation
- Advantages of proof generation
  - No need to trust theorem provers
  - Helps debug the theorem prover
  - Proofs act as easy-to-check witnesses (proof-carrying code)
- Challenges of proof generation
  - Many decision procedures do not follow directly an axiomatization
  - Proofs are produced by "coding the correctness argument" for the decision procedure

Course Overview (IV)

- The hardest part of program verification is <u>invariant generation</u>
- Any systematic method for generating (correct) invariants induces a method for proving invariants
- We will look at several methods
  - Abstract interpretation
  - Induction iteration
  - Model checking

An Imperative Programming Language

- Syntax:
  - L-values
    L ::= x | *E
  - Expressions:
    E ::= L | n | $E_1 + E_2$ | $E_1 = E_2$ | $E_1 \geq E_2$ | …
  - Commands:
    C ::= skip | C_1; C_2 | let x = E in C | L := E |
    if E then C_1 else C_2 | while E do C |
    L := $f(E_1,\ldots,E_n)$
  - Programs:
    P ::= sequence of $f(x_1,\ldots,x_n)$ = C

Programming Language Notes

- Simple variables with integer and pointer values
- Only structured control flow (no goto)
- No constructs for allocation/deallocation of locations
- Call-by-value semantics
- Return values by assignment to special variable
  - E.g., in function f, we return 5 by "f := 5"
  - As in Basic

Operational Semantics

- Values (results of evaluating expressions):
  v ::= n (integer literals)
  | a (addresses)
- A command changes the evaluation state
- State: two components
  - Environment: a mapping from local variables to values
    ρ : Var $\rightarrow$ Value
  - Store: a mapping from addresses to values
    σ : Addr $\rightarrow$ Value

State Manipulation

- Accessing state
  - $\rho(x)$ - the value of variable x in the environment ρ
  - $\sigma(a)$ - the content of store σ at address a
- Updating state: changes the environment or the store
  - $\rho[x := v]$ - an environment like ρ but with x mapped to v
  - $\sigma[a := v]$ - a store like σ but with a mapped to v

Evaluation of Expressions

- Define an evaluation judgment for expressions

$$\rho, \sigma \vdash E \Downarrow v$$

  - In environment ρ and store σ, the value of E is v

$$\frac{}{\rho, \sigma \vdash v \Downarrow v} \qquad \frac{}{\rho, \sigma \vdash x \Downarrow \rho(x)}$$

$$\frac{\rho, \sigma \vdash E_1 \Downarrow n_1 \quad \rho, \sigma \vdash E_2 \Downarrow n_2}{\rho, \sigma \vdash E_1 + E_2 \Downarrow n_1 + n_2} \qquad \frac{\rho, \sigma \vdash E \Downarrow a}{\rho, \sigma \vdash *E \Downarrow \sigma(a)}$$

Evaluation of Commands (I)

- Define an evaluation judgment for commands

$$\rho, \sigma \vdash C \Downarrow \rho', \sigma'$$

  - In environment ρ and store σ, the command C terminates with new environment ρ' and new store σ'

$$\frac{}{\rho, \sigma \vdash \text{skip} \Downarrow \rho, \sigma} \qquad \frac{\rho, \sigma \vdash C_1 \Downarrow \rho_1, \sigma_1 \quad \rho_1, \sigma_1 \vdash C_2 \Downarrow \rho_2, \sigma_2}{\rho, \sigma \vdash C_1;C_2 \Downarrow \rho_2, \sigma_2}$$

$$\frac{\rho, \sigma \vdash E \Downarrow v}{\rho, \sigma \vdash x := E \Downarrow \rho[x := v], \sigma} \qquad \frac{\rho, \sigma \vdash E_1 \Downarrow a \quad \rho, \sigma \vdash E_2 \Downarrow v}{\rho, \sigma \vdash *E_1 := E_2 \Downarrow \rho, \sigma[a := v],}$$

$$\frac{\rho, \sigma \vdash E \Downarrow v \quad \rho[x := v], \sigma \vdash C \Downarrow \rho', \sigma' \quad x \text{ is fresh}}{\rho, \sigma \vdash \text{let } x := E \text{ in } C \Downarrow \rho', \sigma'}$$

Evaluation of Commands (II)

$$\frac{\rho, \sigma \vdash E \Downarrow 0 \quad \rho, \sigma \vdash C_1 \Downarrow \rho', \sigma'}{\rho, \sigma \vdash \text{if } E \text{ then } C_1 \text{ else } C_2 \Downarrow \rho', \sigma'} \qquad \frac{\rho, \sigma \vdash E \Downarrow n \neq 0 \quad \rho, \sigma \vdash C_2 \Downarrow \rho', \sigma'}{\rho, \sigma \vdash \text{if } E \text{ then } C_1 \text{ else } C_2 \Downarrow \rho', \sigma'}$$

$$\frac{\rho, \sigma \vdash E \Downarrow 0}{\rho, \sigma \vdash \text{while } E \text{ do } C \Downarrow \rho, \sigma}$$

$$\frac{\rho, \sigma \vdash E \Downarrow n \neq 0 \quad \rho, \sigma \vdash C \Downarrow \rho', \sigma' \quad \rho', \sigma' \vdash \text{while } E \text{ do } C \Downarrow \rho'', \sigma''}{\rho, \sigma \vdash \text{while } E \text{ do } C \Downarrow \rho'', \sigma''}$$

Evaluation of Commands

A function call

$$x := f(E_1, \ldots, E_n)$$

is interpreted as

$$\text{let } x_1 = E_1 \text{ in } \ldots \text{ let } x_n = E_n \text{ in } f := 0;\ C_f;\ x := f$$

for a function defined as

$$f(x_1, \ldots, x_n) = C_f$$

$$\frac{\begin{array}{l} f(x_1, \ldots, x_n) = C_f \in \text{Program} \\ \rho, \sigma \vdash E_i \Downarrow v_i \quad (i = 1, \ldots, n) \\ \rho[x_1 := v_1]\ldots[x_n := v_n][f := 0], \sigma \vdash C_f \Downarrow \rho', \sigma' \qquad x_1, \ldots, x_n \text{ are fresh} \end{array}}{\rho, \sigma \vdash x := f(E_1, \ldots, E_n) \Downarrow \rho'[x := \rho'(f)], \sigma'}$$