# Techniques for Automated Deduction

CS 294-4
Lecture 1



## Course Administration

- Please write down your name, email address
- Time: Tuesday, Thursday 3:30-5:00pm
- Office hours: Monday 9-11am + by appt.
- Web page:
  http://www.cs.berkeley.edu/~necula/autded

## 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
  - Computing theory: decidability and complexity results
- 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 (i.e. debugging)
    - An extension of type checkers
- Security checkers
  - Security protocols
- Automatic program synthesis from specifications
  - Constraint-based programming
  - 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

## 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:
  - 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 correctness of program P is huge then how did the programmer find it?
  - We only want machines to find proofs that humans can find
  - 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:
  - We are not looking for proofs from first principles
  - Compare the statements
    - Show that the area bounded by y=0, x=1 and $y=x^2$ is 1/3
    - Show that by splicing two circular lists we obtain another circular list with the union of the elements
  - In programming, we are often lacking an effective formal framework for describing and checking results

## 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:
  - Indeed, there usually are as many bugs in the specification as in the program
  - Still redundancy turns many bugs into inconsistencies
  - We are interested in partial specifications
    - An index is within bounds
    - A lock is released
- Discovering specifications is harder than proving their correctness !

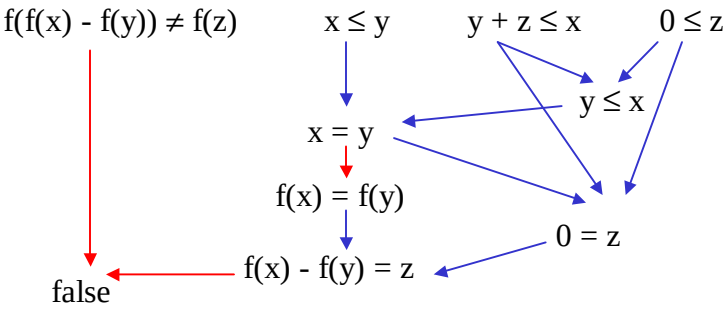


## Coursework

- Attend lectures
- Course Project
- A few homeworks (?)
- Prepare 40 minute lecture
  - Temporal logic, linear logic, belief logics, BDDs, arithmetic decision procedures
- Please register
  - For grade: must do project
  - For S/U: no project
- Course can 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:



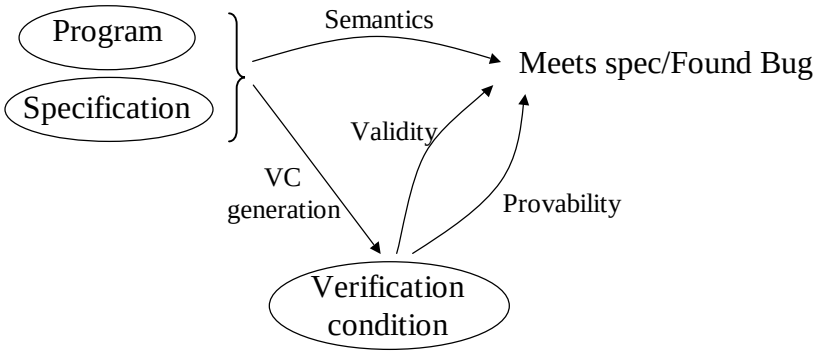



## Course Project (II)

- We develop together the core of the theorem prover
- Each group develops a decision procedure
  - Example: arithmetic, equality, typing, etc.
  - Range from 400 lines to 2000 lines
  - Groups of 2-3
- In Objective CAML (dialect of ML)
  - Will give tutorial if needed
  - Will provide infrastructure (pretty printing, etc.)
- Test cases:
  - Proof-carrying code for Java type safety
  - Translation validation of GCC

## Course Overview

- Focus on automated deduction for software debugging






## Course Overview (II)

- We will discuss fundamentals of logic
  - Propositional calculus
    - Syntax
    - Semantics
    - Deduction systems
    - Automated proof methods
  - 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 more trusting of theorem provers
  - Helps debug the theorem prover
  - Produce 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 invariant generation
- Any systematic method for generating (correct) invariants induces a method for proving invariants
- We will look at several methods
  - Abstract interpretation
  - Induction iteration

## An Imperative Programming Language

- Syntax:
  - L-values
    L ::= x | *E
  - Expressions:
    E ::= L | n | $E_1$ + $E_2$ | $E_1$ = $E_2$ | $E_1$ ≥ $E_2$ | …
  - 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($E_1$,… ,$E_n$) | return E
  - Programs:
    P ::= sequence of f($x_1$,… ,$x_n$) = 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
  - As in Pascal, Visual 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
    $\rho$ : Var $\rightarrow$ Value
  - Store: a mapping from addresses to values
    $\sigma$ : Addr $\rightarrow$ Value



## State Manipulation

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

## Evaluation of Expressions

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

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

$$\frac{\rho, \sigma \vdash E_1 \Downarrow n_1 \qquad \rho, \sigma \vdash E_2 \Downarrow n_2}{\rho, \sigma \vdash E_1 + E_2 \Downarrow n_1 \oplus n_2}$$

$$\frac{\rho, \sigma \vdash E \Downarrow a}{\rho, \sigma \vdash *E \Downarrow \sigma(a)}$$



## Evaluation of Commands (I)

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

$$\rho, \sigma, \text{skip} \Downarrow \rho, \sigma$$

$$\frac{\rho, \sigma, C_1 \Downarrow \rho', \sigma' \qquad \rho', \sigma', C_2 \Downarrow \rho'', \sigma''}{\rho, \sigma, C_1; C_2 \Downarrow \rho'', \sigma''}$$

## Evaluation of Commands (II)

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

$$\frac{\rho,\sigma \vdash E \Downarrow v}{\rho,\sigma, x := E \Downarrow \rho[x:=v],\sigma}$$

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



## Evaluation of Commands (III)

$$\frac{\rho,\sigma \vdash E \Downarrow 1 \quad \rho,\sigma, C_1 \Downarrow \rho',\sigma'}{\rho,\sigma, \text{if } E \text{ then } C_1 \text{ else } C_2 \Downarrow \rho',\sigma'}$$

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

## Evaluation of Commands (IV)

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

$$\frac{\rho, \sigma \vdash E \Downarrow 1 \quad \rho, \sigma, C \Downarrow \rho', \sigma' \quad \rho', \sigma', \text{while } E \text{ do } C \Downarrow \rho'', \sigma''}{\rho, \sigma, \text{while } E \text{ do } C \Downarrow \rho'', \sigma''}$$



## Evaluation of Commands (V)

$$\frac{f(x_1, \ldots, x_n) = C_f \in \text{Program} \qquad \rho, \sigma, \text{let } x_1 = E_1 \text{ in} \ldots \text{let } x_n = E_n \text{ in let } f = 0 \text{ in } C_f; x := f \Downarrow \rho', \sigma'}{\rho, \sigma, x := f(E_1, \ldots, E_n) \Downarrow \rho', \sigma'}$$