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:

\[
\begin{align*}
  f(f(x) - f(y)) &\neq f(z) \\
x &\leq y \\
0 &\leq z \\
\end{align*}
\]

```
false
```

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**

![Diagram]

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
    \[ L ::= x \mid \ast E \]
  - Expressions:
    \[ E ::= L \mid n \mid E_1 + E_2 \mid E_1 = E_2 \mid E_1 \geq E_2 \mid \ldots \]
  - Commands:
    \[ C ::= \text{skip} \mid C_1 ; C_2 \mid \text{let } x = E \text{ in } C \mid L := E \mid \]
    \[ \text{if } E \text{ then } C_1 \text{ else } C_2 \mid \text{while } E \text{ do } C \mid \]
    \[ L := f(E_1, \ldots, E_n) \mid \text{return } E \]
  - Programs:
    \[ P ::= \text{sequence of } f(x_1, \ldots, x_n) = C \]

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
  - As in Pascal, Visual Basic
Operational Semantics

- Values (results of evaluating expressions):
  \[ V ::= n \quad \text{(integer literals)} \]
  \[ | a \quad \text{(addresses)} \]
- A command changes the evaluation state
- State: two components
  - Environment: a mapping from local variables to values
    \[ \rho : \text{Var} \rightarrow \text{Value} \]
  - Store: a mapping from addresses to values
    \[ \sigma : \text{Addr} \rightarrow \text{Value} \]

State Manipulation

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

\[
\begin{align*}
\text{if } s, \sigma; E \Downarrow \nu \\
\text{then } s, \sigma \times \nu E(x) \\
\text{else } s, \sigma \Downarrow \mu
\end{align*}
\]

\[
\begin{align*}
\text{if } s, \sigma; E_1 \Downarrow n_1 \\
\text{then } s, \sigma; E_2 \Downarrow n_2 \\
\text{else } s, \sigma; E \Downarrow n_1 \oplus n_2
\end{align*}
\]

\[
\begin{align*}
\text{if } s, \sigma; E_1 \Downarrow \nu \alpha \\
\text{then } s, \sigma; \beta \Downarrow \nu \alpha(a)
\end{align*}
\]

Evaluation of Commands (I)

\[
\begin{align*}
\text{if } s, \sigma; C \Downarrow s', \sigma'
\end{align*}
\]

\[
\begin{align*}
\text{while } s, \sigma; \text{skip} \Downarrow s, \sigma
\end{align*}
\]

\[
\begin{align*}
\text{if } s, \sigma; C_1 \Downarrow s', \sigma'; \text{then } s, \sigma; C_2 \Downarrow s'', \sigma''
\end{align*}
\]
Evaluation of Commands (II)

\[
\begin{align*}
\frac{\sigma, \Gamma \vdash E \psi \nu \quad \sigma[x := \nu], \Gamma \vdash \psi', \sigma'}{\sigma, \Gamma, \text{let } x = E \quad \psi', \sigma'} & \quad \text{fresh} \\
\frac{\sigma, \Gamma \vdash E \psi \nu}{\sigma, x := E \quad \sigma[x := \nu], \sigma'} \\
\frac{\sigma, \Gamma \vdash E_1 \psi \alpha \quad \sigma, \Gamma \vdash E_2 \psi \nu}{\sigma, \Gamma, \ast E_1 := E_2 \quad \psi', \sigma[x := \alpha \nu]} & \quad \text{where } E_1 \in \epsilon
\end{align*}
\]

Evaluation of Commands (III)

\[
\begin{align*}
\frac{\sigma, \Gamma \vdash E_1 \psi \nu}{\sigma, \Gamma \vdash \text{if } E \text{ then } C_1 \text{ else } C_2 \psi', \sigma'} & \quad \text{where } E_1 \in \epsilon
\end{align*}
\]
Evaluation of Commands (IV)

\[
\begin{align*}
\text{let } \text{E do } C \downarrow & s', f \\
\text{let } s, s', f \downarrow & s', g \\
\text{while } E \text{ do } C \downarrow & s', g
\end{align*}
\]

Evaluation of Commands (V)

\[
\begin{align*}
\text{let } x_1 = E_1 \text{ in } \cdots \text{ let } x_n = E_n \text{ in } \text{let } f = 0 \text{ in } C_f; x := f \downarrow & s', g', 0'
\end{align*}
\]