CMSC 631 – Program Analysis and Understanding
Fall 2011
What you’ll learn

• Formal systems and notations
  ▪ Vocabulary for talking about programs

• Program analysis
  ▪ Automatic reasoning about source code

• Programming language features
  ▪ Affects programs and how we reason about them
How you’ll learn it

• Implement
  ▪ You will build some program analyzers as projects using the Objective Caml programming language

• Prove
  ▪ You will mechanize mathematics for reasoning about programs using the Coq proof assistant
    - Interactive theorem proving: ensures your proofs are actually correct, and therefore that you really understand

• Take it one step further: substantial final project
What you’ll gain

• Better programming ability
  ▪ By understanding programming languages deeply

• Mathematical methods and maturity
  ▪ Formalization and proof techniques will transfer to other areas

• How to use Coq and program in OCaml
  ▪ Write reliable code faster! Prove the four color theorem (with high assurance)!
Personnel

- Michael Hicks
  - Office: 4131 AVW (for now)
  - E-mail: mwh@cs.umd.edu
  - Office hours: 2 hours, TBD (requests?)
    - Or by appointment
Prerequisite

• CMSC 430 or equivalent
  - Ideas we will use in this class:
    - Parse trees/abstract syntax trees
    - BNF notation for grammars
    - Programming language maturity
      - Familiarity with several different languages/paradigms
    - General information about programming language design
  - Talk to me if you’re not sure
Textbooks

• No required textbooks
  ▪ But see web page for suggestions
  ▪ Recommended text:
    - Pierce, *Types and Programming Languages*
  ▪ A second book, also good:
    - Huth and Ryan, *Logic in Computer Science*

• Neither covers everything in the course
• Recommended two on reserve in CS library
Forum

• Piazza
  ▪ See class web page for link
  ▪ Need to sign up

• Can use piazza to ask and answer questions about lectures, assignments, etc.
  ▪ Please use this forum unless you have personal request (e.g., about your grade, an absence, etc.)
Expectations: Homework (40%)

• Programming assignments
  - Symbolic execution and type inference

• Proofs using Coq
  - From basic mathematics to
  - methods for expressing a program’s semantics to
  - methods for proving properties about programs
Late Policy on Assignments

• Programming/Coq assignments: Due at midnight
  ▪ Submit via the submit server (see class web page)

• No late submissions
  ▪ Contact me about extenuating circumstances
    - E.g., religious holidays
  ▪ Inform me as soon as possible
Expectations: Participation (10%)

• Will need to read some papers for class
  ▪ Scattered through the semester
  ▪ Should come prepared to contribute to discussion

• (Possible) student presentations of papers
  ▪ Read 1-2 papers on a topic
  ▪ Present (partial) lecture in class about the material
Expectations: Project (25%)

- Class goal: Teach you how to do research
  - So you have to do research as part of the class

- Substantial research project (25% of grade)
  - Any topic vaguely related to the class is acceptable
    - Will post some suggestions for projects later on
    - May also be able to share project with other class
  - Completed in groups of size 2 (possibly 1 or 3)

- Will occupy the latter 2/3 of semester
  - But will still have some Coq assignments
Expectations: Project (cont’d)

• Deliverables
  - Project proposal (one page) + talk with me
  - Project write-up
    - A conference-style paper (5-15 pages, as appropriate)
  - Implementation, if any
  - In-class presentation
    - 15-20 minutes, depending on # of projects

• In the past, several 631 projects led to papers
  - Not required (!), but possible
Expectations: Exam (25%)

• Final exam
  § Based on course assignments
  § Take home exam
    - The exam will be available for 96 hours
    - You pick a 48-hour window during that time during which to take the exam
  § Dates on class web page
Academic Dishonesty

• Don’t do it
CMSC 631 – Program Analysis and Understanding
Fall 2011

21 Ideas and Applications in Program Analysis in 40 Minutes
Abstract Interpretation

• Rice’s Theorem: Any non-trivial property of programs is undecidable
  ▪ Uh-oh! We can’t do anything. So much for this course...

• Need to make some kind of approximation
  ▪ Abstract the behavior of the program
  ▪ ...and then analyze the abstraction

• Seminal papers: Cousot and Cousot, 1977, 1979
Example

\[ e ::= n \mid e + e \]

\[ \alpha(n) = \begin{cases} 
- & n < 0 \\
0 & n = 0 \\
+ & n > 0 
\end{cases} \]

- Notice the need for ? value
- Arises because of the abstraction
Dataflow Analysis

• Classic style of program analysis

• Used in optimizing compilers
  ▪ Constant propagation
  ▪ Common sub-expression elimination
  ▪ Loop unrolling and code motion
  ▪ etc.

• Efficiently implementable
  ▪ At least, *intraprocedurally* (within a single proc.)
  ▪ Use bit-vectors, fixpoint computation
Control-Flow Graph

\[ x := 3 \]
if (!x) then
  \[ y := z + w \]
else
  L: \{ y := 0 \}
\[ x := 2 \ast x \]
if (!x) then goto L
Lattices and Termination

• Dataflow facts form a lattice

\[ \text{Out}(S) = \text{Gen}(S) \cup (\text{In}(S) - \text{Kill}(S)) \]

• Each statement has a transformation function

- \[ x = ? \]
- \[ x = 3 \]
- \[ x = 6 \]
- \[ \ldots \]
- \[ x = \ast \]

• Terminates because

- Finite height lattice
- Monotone transformation functions
Static Single Assignment Form

• Transform CFG so each use has a single defn
Lambda Calculus

• Three syntactic forms

  \[ e ::= x \quad \text{variable} \]
  \[ \lambda x.e \quad \text{function} \]
  \[ e_1 e_2 \quad \text{function application} \]

• One reduction rule

  \[
  (\lambda x.e_1) e_2 \rightarrow e_1[e_2/x] \quad \text{(replace } x \text{ by } e_2 \text{ in } e_1)\]

• Can represent any computable function!
Example

• Conditionals
  ▪ true = \( \lambda x.\lambda y.x \)    false = \( \lambda x.\lambda y.y \)
  ▪ if a then b else c = a b c
    - if true then b else c = (\( \lambda x.\lambda y.x \)) b c \( \rightarrow \) (\( \lambda y.b \)) c \( \rightarrow \) b
    - if false then b else c = (\( \lambda x.\lambda y.y \)) b c \( \rightarrow \) (\( \lambda y.y \)) c \( \rightarrow \) c

• Can also represent numbers, pairs, data structures, etc, etc.

• Result: Lingua franca of PL
ML: Meta-Language

- ML designed originally for theorem provers
  - But after a while, realized could be general-purpose

- Mostly-functional language
  - Similar to lambda-calculus
    - Mostly functional, encouraged not to use side-effects
    - Call-by-value

- We’ll use OCaml for programming assignments
Program Semantics

• To be able to analyze programs, we have to know what they mean
  ▪ Semantics comes from the Greek semaino, “to mean”

• Three styles of formal semantics
  ▪ Operational semantics (major focus)
    - Like an interpreter
  ▪ Denotational semantics
    - Like a compiler
  ▪ Axiomatic semantics
    - Based on what you can prove about programs
Operational Semantics

- Evaluation is described as transitions in some abstract machine
  - Example: Beta reduction from lambda calculus
    \[(\lambda x.e_1) e_2 \rightarrow e_1[e_2 \backslash x]\]
  - State of machine described by current expression

- There are different styles of abstract machines
  - Small-step (as above), big-step, etc

- The meaning of a program is its fully reduced form (a.k.a. a value)
Denotational Semantics

• The meaning of a program is defined as a mathematical object, e.g., a function or number

• Typically define an interpretation function $\llbracket \cdot \rrbracket$
  
  ▪ Program fragment as argument and returns meaning
  
  ▪ E.g., $\llbracket 3+4 \rrbracket = 7$

• Gets interesting when we try to find denotations of loops or recursive functions
Denotational Semantics Example

- \( b ::= \text{true} \mid \text{false} \mid b \lor b \mid b \land b \)
- \( e ::= 0 \mid 1 \mid \ldots \mid e + e \mid e \ast e \)
- \( s ::= e \mid \text{if } b \text{ then } s \text{ else } s \)
- Semantics:
  - \( \llbracket \text{true} \rrbracket = \text{true} \)
  - \( \llbracket b_1 \lor b_2 \rrbracket = \begin{cases} \text{true} & \text{if } \llbracket b_1 \rrbracket = \text{true} \text{ or } \llbracket b_2 \rrbracket = \text{true} \\ \text{false} & \text{otherwise} \end{cases} \)
  - \( \llbracket \text{if } b \text{ then } s_1 \text{ else } s_2 \rrbracket = \begin{cases} \llbracket s_1 \rrbracket & \text{if } \llbracket b \rrbracket = \text{true} \\ \llbracket s_2 \rrbracket & \text{if } \llbracket b \rrbracket = \text{false} \end{cases} \)
Axiomatic Semantics

• Operational and denotational semantics let us reason about the meaning of a program
  ▪ Are two programs equivalent? Does a program terminate? Does a program implement a particular specification

• Axiomatic semantics define a program’s meaning in terms of what one can prove about it
  ▪ Hoare, Dijkstra, Gries, others
Hoare Triples

• \{P\} S \{Q\}
  - If statement S is executed in a state satisfying precondition P, then S will terminate, and Q will hold of the resulting state
  - Partial correctness: ignore termination

• Weakest precondition for assignment
  - Axiom: \{Q[e/x]\} x := e \{Q\}
  - Example: \{y > 3\} x := y \{x > 3\}
Type Systems

• Machine represents all values as bit patterns
  ▪ Is 00110110111100101100111010101000
    - A signed integer? Unsigned integer? Floating-point number?
      Address of an integer? Address of a function? etc.

• Type systems allow us to distinguish these
  ▪ To choose operation (which + op), e.g., FORTRAN
  ▪ To avoid programming mistakes
    - E.g., don’t treat integer as a function address
Simply-typed $\lambda$-calculus

$$e ::= x | n | \lambda x:\tau.e | e \ e$$

$$\tau ::= \text{int} | \tau \rightarrow \tau$$

$$A \vdash e : \tau \quad \text{in type environment } A, \text{ expression } e \text{ has type } \tau$$

$$\begin{align*}
A \vdash n : \text{int} \\
A \vdash x : A(x)
\end{align*}$$

$$\begin{align*}
A[\tau\backslash x] \vdash e : \tau' \\
A \vdash \lambda x:\tau.e : \tau \rightarrow \tau'
\end{align*}$$

$$\begin{align*}
A \vdash e_1 : \tau \rightarrow \tau' \quad A \vdash e_2 : \tau \\
A \vdash e_1 \ e_2 : \tau'
\end{align*}$$
Subtyping

• Liskov:
  - If for each object \( o_1 \) of type \( S \) there is an object \( o_2 \) of type \( T \) such that for all programs \( P \) defined in terms of \( o_1 \), the behavior of \( P \) is unchanged when \( o_2 \) is substituted for \( o_1 \) then \( S \) is a subtype of \( T \).

• Informal statement
  - If anyone expecting a \( T \) can be given an \( S \) instead, then \( S \) is a subtype of \( T \).
Other Technologies and Topics

- Control-flow analysis
- CFL reachability and polymorphism
- Constraint-based analysis
- Alias and pointer analysis
- Region-based memory management
- Garbage collection
- More...
Applications: Dataflow analysis

- Optimizing compilers
  - I.e., any good compiler

- ESP: Path-sensitive program checker (Microsoft)
  - Example: can check for correct file I/O properties, like files are opened for reading before being read

- Meta-level compilation (Coverity)

- ...

Friday, September 2, 2011
Applications: Abstract Interp.

• Terminator (Microsoft)
  - Analyzes code to prove that it terminates (!)
  - Applied to device drivers for Windows kernel
    - Tricky part is reasoning about the heap

• ASTREE (INRIA and others)
  - Used to detect all possible runtime failures (divide by zero, null pointer deref, array out of bounds) on embedded code
  - Used regularly on Airbus avionics software
Applications: Symbolic Execution

- A symbolic executor is a language interpreter
  - Rather than only work on concrete values, also works on symbolic values
  - Ex: \( y = \text{fresh}(); \ assert(f(y) == 2*y-1); \)
  - Solver conceptually “forks” on tests of symbolic values
- Uses SMT solver to check assertions, path feasibility
  - SMT = Satisfiability Modulo Theory = SAT
  - Solvers can solve very large instances, even though SAT theoretically intractable (i.e., NP Hard)
- Very popular: DART, CUTE, EXE, KLEE, Otter, Rubyx, etc
  - SAGE tool in regular use at Microsoft for fuzz testing
Applications: Axiomatic Semantics

• Extended Static Checker
  ▪ Can perform deep reasoning about programs
  ▪ Array out-of-bounds
  ▪ Null pointer errors
  ▪ Failure to satisfy internal invariants

• Uses the Simplify theorem prover
Applications: Type Systems

• Type qualifiers
  - Format-string vulnerabilities, deadlocks, file I/O protocol errors, kernel security holes

• Jif (Java+Information Flow)
  - Annotate standard types with additional security labels, where type correctness implies correct protection of sensitive data
Conclusion

• PL has a great mix of theory and practice
  ▪ Very deep theory
  ▪ But lots of practical applications

• Recent exciting new developments
  ▪ Focus on program correctness (and security)
    - instead of speed
  ▪ Scalability to large programs
  ▪ In greater use in mainstream development