# CMSC 631 – Program Analysis and Understanding
Fall 2010

## Analyze and Understanding Software

- 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



## Personnel

- Jeff Foster
  - Office: 4129 AVW
  - E-mail: jfoster at cs.umd.edu
  - Office hours: M 12-1, W 1-2
    - Or by appointment
- TA: Bao Nguyen
  - E-mail: baonn at cs.umd.edu
  - Office hours: M 10-11, Tu 1-3



## Prerequisite

- CMSC ~~430~~ 330 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

- 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
- On reserve in CS library



## Forum

- Web forum on CS dept server
  - See class web page for link

- Can use the forum to communicate with others
  - Questions about assignments and projects
  - Thoughts of general interest



## Expectations: Homework (30%)

- Written assignments
  - Short problem sets
- Programming assignments
  - Implement ideas from lecture
- Proofs in Coq
  - Solve problem sets using the Coq proof assistant
  - You will know immediately if you get it right!
- This is how you will learn things
  - Much more effective than listening to a lecture



## Late Policy on Assignments

- Programming/Coq assignments: Due at midnight
  - Submit via the submit server (see class web page)
- Written assignments: Due at start of class

- 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, but more during second half
  - 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 (35%)

- Class goal: Teach you how to do research
  - So you have to do research as part of the class
- Substantial research project (35% 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)
- Projects will happen across most of the semester
  - Assignments will taper off in second half



## 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**
**Spring 2009**

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 | e + e

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

| + | - | 0 | + |
|---|---|---|---|
| - | - | - | ? |
| 0 | - | 0 | + |
| + | ? | + | + |

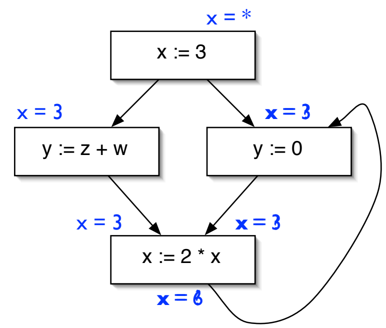
- Notice the need for ? value
  - Arises because of the abstraction

## 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



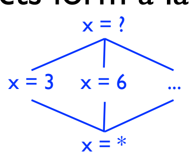
## Control-Flow Graph






## Lattices and Termination

- Dataflow facts form a lattice

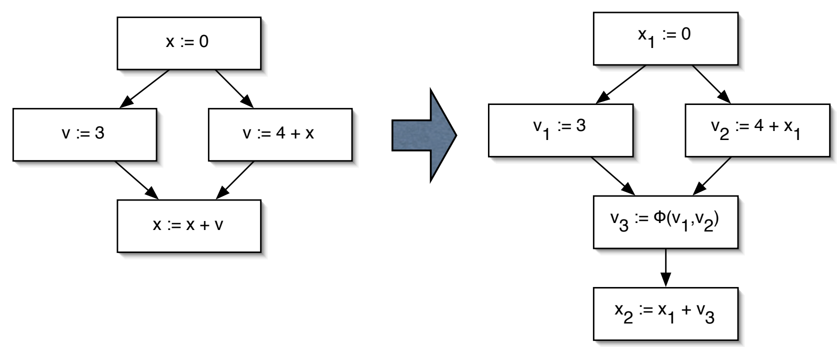



- Each statement has a transformation function
  - Out(S) = Gen(S) U (In(S) - Kill(S))
- Terminates because
  - Finite height lattice
  - Monotone transformation functions



## Static Single Assignment Form

- Transform CFG so each use has a single defn



## Lambda Calculus

- Three syntactic forms

| | |
|---|---|
| e ::= x | variable |
| \| $\lambda$x.e | function |
| \| e e | function application |

- One reduction rule
  - $(\lambda x.e_1)\ e_2 \to e_1[e_2\backslash x]$ (replace $x$ by $e_2$ 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 = $(\lambda x.\lambda y.x)\ b\ c \to (\lambda y.b)\ c \to b$
    - if false then b else c = $(\lambda x.\lambda y.y)\ b\ c \to (\lambda y.y)\ c \to c$

- 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* $[\![\ ]\!]$
  - Program fragment as argument and returns meaning
  - E.g., $[\![\ 3+4\ ]\!] = 7$
- Gets interesting when we try to find denotations of loops or recursive functions



## Denotational Semantics Example

- $b ::= \text{true} \mid \text{false} \mid b \vee b \mid b \wedge b$
- $e ::= 0 \mid 1 \mid \ldots \mid e + e \mid e * e$
- $s ::= e \mid \text{if } b \text{ then } s \text{ else } s$
- Semantics:
  - $[\![\ \text{true}\ ]\!] = \text{true}$
  - $[\![\ b1\ \ b2\ ]\!] = \begin{cases} \text{true} & \text{if } [\![b1]\!] = \text{true or } [\![b2]\!] = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$
  - $[\![\ \text{if } b \text{ then } s1 \text{ else } s2\ ]\!] = \begin{cases} [\![s1]\!] & \text{if } [\![b]\!] = \text{true} \\ [\![s2]\!] & \text{if } [\![b]\!] = \text{false} \end{cases}$



## 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 λ-calculus

$e ::= x \mid n \mid \lambda x{:}\tau.e \mid e\ e$

$\tau ::= \text{int} \mid \tau \to \tau$

$A \vdash e : \tau$ in type environment A, expression e has type $\tau$

$$\frac{}{A \vdash n : \text{int}} \qquad \frac{x \in \text{dom}(A)}{A \vdash x : A(x)}$$

$$\frac{A[\tau \backslash x] \vdash e : \tau'}{A \vdash \lambda x{:}\tau.e : \tau \to \tau'} \qquad \frac{A \vdash e1 : \tau \to \tau' \quad A \vdash e2 : \tau}{A \vdash e1\ e2 : \tau'}$$



## 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 reachablity and polymorphism
- Constraint-based analysis
- Alias and pointer analysis
- Region-based memory management
- Garbage collection
- More...



## Applications: Parsing

- Syntactic bug pattern checkers
  - ASTLog
  - PREFast
    - Buffer overflows! (sizeof() of wrong type in copy operations)
  - FindBugs
    - wait() not inside of a loop
    - Pointer to internal array returned (unsafe)
    - Dereference of null pointer



## Applications: Abstract Interp.

- Polyspace
  - Looks for race conditions, out-of-bounds array accesses, null pointer derefs, etc
  - Also includes arithmetic equation solver
- ASTREE
  - 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: Dataflow analysis

- Optimizing compilers
  - I.e., any good compiler
- ESP: Path-sensitive program cheker
  - Example: can check for correct file I/O properties, like files are opened for reading before being read
- LCLint: Memory error checker (plus more)
- Meta-level compilation: Checks lots of stuff
- ...

## 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
- Vault and Cyclone
  - Memory allocation and deallocation errors, library protocol errors, misuse of locks



## Applications: Symbolic Execution

- A symbolic executor is a language interpreter++
  - Rather than only work on concrete values, also works on symbolic values
  - Ex: y = fresh(); assert(f(y) == 2*y-1);
  - Solver conceptually "forks" on tests of symbolic values
- Uses SMT solver to check assertions, path feasiblity
  - SMT = Satisfiability Modulo Theory = SAT++
  - Solvers can solve very large instances, even though SAT theoretically intractable
- Very popular: DART, CUTE, EXE, KLEE, Otter, Rubyx, etc



## 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 instead of speed
  - Forget about full correctness, though
  - Scalability to large programs essential