

CMSC 631 – Program Analysis and Understanding Spring 2006

About this Class

- Topic: Analyzing and understanding software
- Three main focus areas:
 - Static analysis
 - Automatic reasoning about source code
 - Formal systems and notations
 - Vocabulary for talking about programs
 - 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 1-2pm, Tu 11am-12pm
 - Or by appointment

Prerequisite

- CMSC 430 or equivalent compiler class
 - Ideas we will use in this class:
 - Parse trees/abstract syntax trees
 - BNF notation for grammars
 - Type checking (usually not much covered in compilers class)
 - Data flow analysis (sometimes not covered in compilers class)
 - Tools like yacc and lex may be useful for your project
 - We won't use most of the other material
 - So even if you haven't taken compilers class, you may be OK
 - Talk to me if you're not sure

Textbooks

- No required textbooks
- Two recommended texts
 - Pierce, *Types and Programming Languages*
 - 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
 - <https://forum.cs.umd.edu/forumdisplay.php?f=22>
- Can use the forum to communicate with others in the class and ask questions of general interest

Expectations: Homework

- Two kinds of assignments:
 - Programming assignments (20% of grade)
 - Every two weeks
 - Implement the ideas we see in lecture
 - Written assignments (10% of grade)
 - Every week
 - Short problem sets
- This is how you will learn things
 - Much more effective than listening to a lecture

Late Policy on Assignments

- Programming Assignments: Due at midnight
- Written assignments: Due at beginning of class
 - No late submissions
- Contact me about extenuating circumstances
 - E.g., religious holidays
 - Inform me as soon as possible

Expectations: Participation

- Will need to read some papers for class
 - More later on in the semester
 - Should come prepared to contribute to discussion
- Student presentations later in the semester
 - Read 1-2 papers on a topic
 - Present a lecture in class about the material
- 10% of grade on class participation

Expectations: Project

- 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 1 or 2
- This will consume second-half of semester
 - Will ease up on homeworks, reading

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

Expectations: Exam

- Final exam (25% of grade)
 - Based on written and programming assignments
 - Take-home or in-class (we'll vote at the end of the semester)

Academic Dishonesty

- Don't do it

Software Chat

<http://www.cs.umd.edu/projects/softchat>

- Weekly meeting about programming languages, software engineering, and software systems
- Mondays at 11am in 3118 CSIC this fall
 - Starting Monday
- Topics include
 - Current research in the department
 - Practice talks
 - Interesting recent papers

CMSC 631 – Program Analysis and Understanding Spring 2006

20 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

$e ::= n \mid e + e$

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

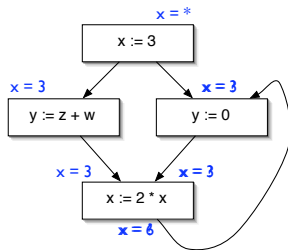
+	-	0	+
-	-	-	?
0	-	0	+
+	?	+	+

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

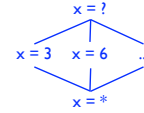
Dataflow Analysis

- Classic style of program analysis
- Used in optimizing compilers
 - Constant propagation
 - Common sub-expression elimination
 - etc.
- Efficiently implementable
 - At least, interprocedurally (within a single proc.)
 - Use bit-vectors, fixpoint computation

Control-Flow Graph



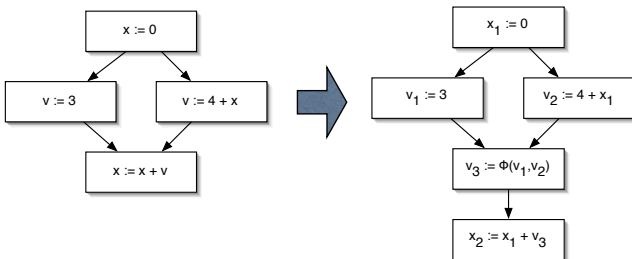
Lattices and Termination



- Dataflow facts form a lattice
- Each statement has a transformation function
 - $Out(S) = Gen(S) \cup (In(S) - Kill(S))$
- 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
 - $(\lambda x. e_1) e_2 \rightarrow e_1[e_2/x]$ (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 = $(\lambda x. \lambda y. x) b c \rightarrow (\lambda y. b) c \rightarrow b$
 - 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

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

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

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

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

$$\frac{}{A \vdash n : \text{int}}$$

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

$$\frac{A[\tau/x] \vdash e : \tau'}{A \vdash \lambda x:\tau. e : \tau \rightarrow \tau'}$$

$$\frac{A \vdash e_1 : \tau \rightarrow \tau' \quad A \vdash e_2 : \tau}{A \vdash e_1 e_2 : \tau'}$$

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 .

Axiomatic Semantics

- Old idea: Shouldn't just hack up code, try to prove programs are correct
- Proofs require reasoning about the meaning of programs
- First system: Formalize program behavior in logic
 - 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: $\{y > 3\} x := y \{x > 3\}$

Model Checking

- Technique for validating hardware
 - Lots of parallelism (concurrency), but
 - Not a lot of structure (e.g., no dynamic allocation)
- Example: mutual-exclusion protocol

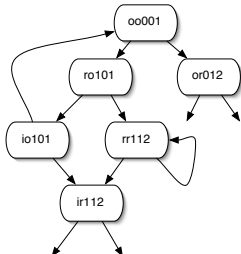
```
loop
  out: x1 := 1; last := 1
  req: await x2 = 0 or last = 2
  in:  x1 := 0
end loop
```

```
loop
  out: x2 := 1; last := 2
  req: await x1 = 0 or last = 1
  in:  x2 := 0
end loop
```

(Example from Henzinger)

Transition Graph

- Program defines a state graph
 - State = (pc1, pc2, x1, x2, last)
 - Is any bad state (iiXXX) reachable from the start state?



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: 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.

- Everything!
- But in particular, Polyspace
 - Looks for race conditions, out-of-bounds array accesses, null pointer dereferences, non-initialized data access, etc.
 - Also includes arithmetic equation solver

Applications: Dataflow analysis

- Optimizing compilers
 - I.e., any good compiler
- ESP: Path-sensitive program checker
 - 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: Symbolic Evaluation

- PREFix
 - Finds null pointer dereferences, array-out-of bounds errors, etc.
 - Used regularly at Microsoft
- Also ESP

Applications: Model Checking

- SLAM and BLAST
 - Focus on device drivers: lock/unlock protocol errors, and other errors sequencing of operations
- Uses alias analysis, predicate abstraction, and more

Applications: Axiomatic Semantics

- Extended Static Checker
 - Can perform deep reasoning about programs
 - Array out-of-bounds
 - Null pointer errors
 - Failure to satisfy internal invariants
- Based on theorem proving

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

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