CSC2125 – Advanced Topics in Software Engineering: Program Analysis and Understanding
Fall 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

Readings

• Supplemental readings from classical papers and from recent advances

Preparation

• A course in compilers would be helpful
• A course in model-checking would be most helpful

Expectations

■ Periodic written assignments (not graded)
  - Short problem sets
  - This is how you will learn things
    - Much more effective than listening to a lecture
■ Course participation (discussion of written assignments and course material)
■ Presentation of part of course material
■ Presentation of one application

What this course is about?

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

• $e ::= n \mid e + e$

\[\begin{array}{c|c|c|c}
+ & 0 & + \\
- & - & ? \\
= & 0 \cdot 0 & 0 & + \\
\end{array}\]

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

• 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
  ■ 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

\[\begin{array}{ccc}
\text{x = 3} & \Rightarrow & \text{x = 0} \\
\text{x = 6} & \Rightarrow & \text{x = 3} \\
\end{array}\]

• Each statement has a transformation function
  ■ $\text{Out(S)} = \text{Gen(S)} \cup (\text{In(S)} - \text{Kill(S)})$
• Terminates because
  ■ Finite height lattice
  ■ Monotone transformation functions

Static Single Assignment Form

• Transform CFG so each use has a single defn

\[\begin{array}{ccc}
\text{v = 3} & \Rightarrow & \text{v = 4 + x} \\
\end{array}\]
**Lambda Calculus**

- Three syntactic forms
  - \( e ::= x \) variable
  - \( \lambda x.e \) function
  - \( e_1 e_2 \) 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
  - \( \text{true} = \lambda x.\lambda y.x \quad \text{false} = \lambda x.\lambda y.y \)
  - \( \text{if a then b else c} = a \ b \ c \)
    - 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

**Type Systems**

- Machine represents all values as bit patterns
  - Is 00110110111100101100111010101000

- 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**

- \( e ::= x | n | \lambda x : \tau . e | e_1 e_2 \)

- \( \tau ::= \text{int} | \tau \rightarrow \tau \)

- Axioms
  - \( e : \tau \) in type environment \( A \), expression \( e \) has type \( \tau \)
    - \( A\vdash n : \text{int} \)
    - \( x \in \text{dom}(A) \)
    - \( A\vdash x : A(x) \)
    - \( A\vdash e_1 : \tau \rightarrow \tau ' \)
    - \( A\vdash e_2 : \tau \)
    - \( A\vdash \lambda x : \tau . e : \tau \rightarrow \tau ' \)

**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: \((y > 3) \ x := y \{x > 3\}\)

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: 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: Symbolic Evaluation

• PREFIX
  ■ Finds null pointer dereferences, array-out-of bounds errors, etc.
  ■ Used regularly at Microsoft

  • Also ESP

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: Model Checking

• SLAM, BLAST, Yasm
  ■ Focus on device drivers: lock/unlock protocol errors, and other errors sequencing of operations

  • Uses alias analysis, predicate abstraction, analysis of recursive functions…
Applications: Axiomatic Semantics

- Extended Static Checker and Spec#
  - 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

- Source: Jeff Foster's course in Univ. of Maryland

Possible Course Syllabus

- Week 1: Introduction, course setup
- Week 2: Dataflow analysis
- Week 3: More dataflow. PA as MC of AI, monotone frameworks
- Week 4: Program semantics (Schmidt), worklist algorithms
- Week 5: Interprocedural analysis, context sensitive analysis
  - (Pnueli), Bebob, Reps/Sagiv
- Week 6: Abstract Interpretation
- Week 7: More abstract interpretation (widening, shape analysis)
- Week 8: Lambda calculus, Type systems
- Week 9: Type systems (Cont'd), powersets
- Week 10: Axiomatic semantics
- Week 10: Axiomatic semantics, weakest precondition, C#, ESC/Java
- Week 12: Applications: Slicing and testcase generation
- Week 13: Applications: Security analysis

Introduction to the actual material

- Data-flow analysis – reaching definitions
  - From Chapter 1 of textbook
  - Slides 15, 18-37

- Abstract interpretation
  - From Chapter 1 of textbook
  - Slides 58-71