CMSC 631 – Program Analysis and Understanding
Fall 2003

Expectations: Readings

- Will read 1-2 papers for each class
  - Typically, papers available on the web
  - Otherwise will hand out photocopies in class
- Must participate in brief discussion on class wiki
  - Post a few sentences to a paragraph or two on
    - Main contributions/ideas in paper
    - Ideas that were unclear – what do I need to cover in class?
    - Relationship to other papers we’ve read
    - etc...

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

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

Expectations: Readings (cont’d)

- Must post comments by noon on day of class
  - First post! can just put up a summary
  - Later posts need to take earlier posts into account
  - Posting earlier is less work

- 20% of grade will be on class participation, including comments on wiki
  - Includes talk on selected paper in 2nd half of course

About Me

- Office: 4129 AVW
- E-mail: jfoster at cs.umd.edu
- Office hours: Tu 10am-11am, Fr 11am-12pm
**Expectations: Project**

- Class goal: Teach you how to do research
  - So you have to do research as part of the class
- Substantial research project (40% of grade)
  - Any topic vaguely related to the class 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
  - Turn in project write-up at end of semester
  - Give short (15-20 minute) presentation in class at end of semester

**Expectations: Exam**

- Final exam and/or midterm (40% of grade)
  - Will be some written assignments early on to prepare
  - List of material covered on exams later on

**Academic Dishonesty**

- Don’t do it

**Software Chat**

- Weekly meeting about programming languages, software engineering, and software systems
- Fridays at 3pm in CSIC 3120 this fall
  - Starting September 12th
- Topics include
  - Current research in the department
  - Practice talks
  - Interesting recent papers

**Administrivia**

- No class on Thursday, September 4

---

20 Ideas and Applications in Program Analysis and Understanding
Fall 2003

20 Ideas and Applications in Program Analysis and Understanding
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

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

Example

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

Dataflow Analysis

• Classic style of program analysis
• Used in optimizing compilers
  ▪ Constant propagation
  ▪ Common sub-expression eliminating
  ▪ Loop unrolling and code motion
  ▪ 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) 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

Properties of SSA

- Could increase CFG size dramatically
  - But in practice, linear growth

- Compact representation of def-use information
  - Makes other dataflow analyses easier
    - Dead code elimination: check if defn of $x_i$ never used
    - Constant propagation: initially assume edges unreachable
  - Abstractly: turns flow-sensitive problem into flow-insensitive problem

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

Simply-typed $\lambda$-calculus

$$
\begin{align*}
  e &::= x \mid \lambda x.e \mid e_1 e_2 \\
  \tau &::= \text{int} \mid \tau \rightarrow \tau \\
  A \vdash e : \tau & \quad \text{in type environment } A, \text{expression } e \text{ has type } \tau \\
  A \vdash n : \text{int} & \quad x \in \text{dom}(A) \\
  A \vdash x : A(x) & \\
  A[\tau\chi] \vdash e : \tau' & \quad A \vdash e_1 : \tau \rightarrow \tau' \quad A \vdash e_2 : \tau \\
  A \vdash \lambda x. e : \tau \rightarrow \tau' & \quad A \vdash e_1 e_2 : \tau'
\end{align*}
$$

Lambda Calculus

- Three syntactic forms
  - $e ::= x$ variable
  - $\lambda x.e$ function
  - $e_1 e_2$ function application

- One reduction rule
  - $(\lambda x.e_1) e_2 \rightarrow e_1[e_2/x] \quad (\text{replace } x \text{ by } e_2 \text{ in } e_1)$

- Can represent any computable function!

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

Type Inference

- One drawback to type checking
  - We need to come up with parameter types
  - What if we don’t want to write them down?

- Solution: Type inference
  - Given: A bare program (no type annotations)
  - Compute: A valid typing
    - Or determine that no valid typing exists
**Type Inference Rules**

- Introduce type variables to stand for unknowns
  \[ \tau ::= \alpha \mid \text{int} \mid \tau \to \tau \]
  
  \[ \quad \quad \quad \quad \quad \quad \quad \quad \quad\]

  \[ A \vdash n : \text{int} \]
  
  \[ A \vdash x : A(x) \quad x \in \text{dom}(A) \]

  \[ A[\alpha/x] \vdash e : \tau' \quad \alpha \text{ fresh} \]
  
  \[ A \vdash \lambda x.e : \alpha \to \tau' \]

**Unification**

- The side conditions in type inference rules are equality or unification constraints
- Can be solved in linear time
  - Though typical implementation is almost linear

  \[ \begin{align*}
  C U \{ \text{int} = \text{int} \} &\Rightarrow C \\
  C U \{ \alpha = \tau \} &\Rightarrow C[\tau/\alpha] \\
  C U \{ \tau = \tau' = \gamma \} &\Rightarrow C U \{ \tau = \tau' \} U \{ \gamma = \gamma' \}
  \end{align*} \]

  otherwise \Rightarrow \text{unsatisfiable}

**Problem with Monomorphic Typing**

- Assume simply-typed \( \lambda \)-calculus
  - Assume primitive types int and float

  \[ \text{let id = } \lambda x.x \text{ in } \quad \text{id} : \alpha \to \alpha \]
  
  \[ \text{id 3; } \quad \alpha = \text{int} \]
  
  \[ \text{id 3.14; } \quad \alpha = \text{float} \quad \text{inconsistent!} \]

- The \text{id} function doesn’t care what \( x \) is
  - But the type system does
  - A monomorphic type system rejects this program

**Hindley-Milner Polymorphism**

- Solution: Give \text{id} a parametric polymorphic type

  \[ \begin{align*}
  \text{let id = } \lambda x.x \text{ in } &\quad \text{id} : \forall \alpha. \alpha \to \alpha \\
  \text{id, 3; } &\quad \text{id} : \beta \to \beta \quad \beta = \text{int} \\
  \text{id, 3.14; } &\quad \text{id} : \gamma \to \gamma \quad \gamma = \text{float} 
  \end{align*} \]

- This is universal quantification
  - \text{id} works for any type of \( x \)
  - At each occurrence of \text{id}, we instantiate it with a fresh copy of its type
  - Just like inlining, but done in the type system

**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 \( T \).

**Subtyping Rule**

- Familiar for object-oriented programming
  - Usually use \text{subclass} instead of \text{subtype}

- Easily added to type checking, type inference

  \[ A \vdash e : \tau \]

  \[ \tau \leq \tau' \]

  \[ A \vdash e : \tau' \]
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
  - Axiomatic Semantics

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: \([y > 3] \ x := y \ {x > 3}\)

Denotational Semantics

- What mathematical structures do programs represent?
  - How do we reason compositionally about programs?
  - State: Variables → Values
  - \(\llbracket \rfloor\): Statement → (State → State)
  - \(\llbracket \text{skip} \rrbracket = \lambda s.s\)
  - \(\llbracket x := e \rrbracket = \lambda s.(v \ x)\) where \(v = \llbracket e \rrbracket s\)
  - \(\llbracket \text{if } \theta \text{ then } C \text{ else } C' \rrbracket = \lambda s.\text{if}(\llbracket \theta \rrbracket s, \llbracket C \rrbracket s, \llbracket C' \rrbracket s)\) where
    - \(\text{if}(\text{true}, v, v') = v\)
    - \(\text{if}(\text{false}, v, v') = v'\)

Loops in Denotational Semantics

- Loops are tricky:
  - Want \(\llbracket \text{while } B \text{ do } C \rrbracket\) to be defined in terms of \(B\) and \(C\)
  - \(\llbracket \text{while } B \text{ do } C \rrbracket = \lambda s.\text{if}(\llbracket B \rrbracket s, \llbracket C \rrbracket s, \llbracket C' \rrbracket s)\) if \(\llbracket B \rrbracket s = \text{false}\)
  - \(\llbracket \text{while } B \text{ do } C \rrbracket = \llbracket C \rrbracket; \{\llbracket B \rrbracket s = \text{true}\})\)
  - But that’s not compositional reasoning!
    - \(\text{while}\) is defined in terms of itself
  - Solution: Need fixpoint operation
    - Define domains on which minimal fixpoints exist
    - More on this later in the course

Operational Semantics

- Denotational semantics is great, but
  - It’s hard!
  - And often is more than we need
- Reason about programs via symbol pushing
  - Non-compositionality is OK
  - Trace computation step-by-step

Operational Semantic Rules

- A state is a tuple \(<S, C>\)
  - \(S\) : variables → values
  - \(C\) is a command
- Transitions from states to states
  - \(\langle S, \text{skip}; C \rangle \rightarrow \langle S, C \rangle\)
  - \(\langle S, x = v; C \rangle \rightarrow \langle S[v/x], C \rangle\)
  - \(\langle S, \text{while } B \text{ do } C; C' \rangle \rightarrow \langle S, C' \rangle\) if \(B = \text{false}\)
  - \(\langle S, \text{while } B \text{ do } C; C' \rangle \rightarrow \langle S, C; \text{while } B \text{ do } C; C' \rangle\) if \(B = \text{true}\)
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
```

(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?

Applications: Parsing

- Syntactic bug pattern checkers
  - ASTLog
  - PREFast
    - Buffer overflows! (sizeof() of wrong type in copy operations)
  - Pugh’s bug finder for Java
    - 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