Program Analysis

- Static analysis: inspect programs at compile-time
- Extract information about program execution
  - Characterize dynamic program executions
- Use analysis results for:
  - Optimizations and transformations
  - Program verification
  - Error detection
  - Program understanding

Static vs. Dynamic

- Static analysis:
  - Work done at compile-time
  - Characterizes all executions
  - Conservative: approximates concrete program states
- Dynamic analysis:
  - Run-time overhead
  - Characterizes one or a few executions
  - Precise: knows the concrete program state
  - Can’t “look into the future”

Classifying Program Analyses

- Lots of approaches to static analysis
  - How do they compare to each other?
  - What distinguishes them?
- Main aspects of program analyses:
  - What information are we interested in?
  - What program constructs?
  - How does the analysis work?
  - How much user interaction?
  - Is the analysis sound?

Analysis Information

- Figure out “facts” about the program execution
- Facts typically talk about:
  - The values in the memory
    - Constant propagation: \( x = 5 \)
    - Points-to analysis: \( x \) points to \( y \)
    - Types: value of \( x \) is an integer
    - Verification: the result of \( \text{fact}(n) = n! \)
  - Events during program execution
    - Liveness: variable \( x \) never used in the future
    - Temporal properties, e.g. lock-unlock property
Analysis Information

• How much information depends on the client
• E.g., program verification: show lack of errors
• What is an error?
  – Type error?
  – Memory error?
  – Incorrect result?

Where Do Facts Hold?

• Facts hold:
  – Either locally (e.g., at a particular program points)
  – Or globally (throughout the program. E.g., types)
• Program points approximate sets of points in dynamic execution traces
• Can refine program points using:
  – The calling stack when the execution reaches a point
  – The program path that lead to a point

Program Constructs

pointers recursive structures functions higher-order functions
arrays destructive updates control constructs
virtual calls objects threads
exceptions inheritance machine code

Analysis Techniques

• Dataflow analysis, Abstract interpretation
  – Flow-sensitive: track facts through the control flow
• Type systems
  – Check or infer types for program expressions
  – Typically flow-insensitive
• Constraint methods
  – Reduce the analysis problem to a set of constraints
  – Examples: set constraints, linear systems, boolean formulas, etc.
  – Separates specification from implementation
• Model checking
  – Check properties expressed as temporal logic formulas
• Theorem proving
  – Use logical deduction to prove facts

Abstractions

• Analyses must use abstractions
  – Model computation in the program
  – Model program state
    • describe unbounded sets of unbounded states
    • Finite, tractable abstractions are desirable
• Examples:
  – Dataflow, AI, CFGs, SSA, lattices
  – Model checking: transition systems, temporal logic formulas
  – Type systems: type abstraction, typing rules (type constraints)
  – Constraint methods: constraints
  – Theorem proving: theorems
User Interaction

• Three ways users can interact with analyses:
  – Help the analysis: annotations, specifications
    • Typical example: types
    • Best way to help the analysis: provide information at procedure boundaries, loop invariants (Hoare-style)
  – Help the analysis: interactive
    • Provide help while the analysis runs
  – Tell the analysis what to compute: parameterization
    • User tells what facts the analysis should compute/verify
    • Example: finite state machine models

Proving Soundness

• How do I know that the analysis is sound?
  – Define program semantics
  – AI framework: show that abstract transformer yields conservative results
  – Fairly straightforward for standard compiler analyses
  – Type systems: progress + preservation

• Another approach:
  – Define abstraction
  – Automatically build sound analyses for that abstraction

Soundness

• Soundness: analysis conservatively approximates all program executions
• Unsound analyses: might miss some facts
  – “false negatives” = “missed facts”
  – “false positives” = “facts that never occur”

• Is soundness desirable?
  – Definitely for analyses, transformations, verification
  – Error-detection is a different story
    • Unsound analyses okay
    • Unsound analyses can prove the presence of errors, not their absence

• Sources of unsoundness:
  – Treatment of aliasing, loops, recursion, type-unsafe constructs

Efficiency and Scalability

• Analyses can be expensive
  – E.g., inter-procedural, flow-sensitive analyses

• Ways to make an analysis scalable:
  – Reduce precision
  – Request user annotations
  – Be unsound

This Course

• Programming paradigms and constructs:
  – Focus on analyses for imperative languages
  – Look at: inter-procedural analysis, OO features, pointers, recursive structures, machine code, threads

• Analysis Techniques:
  – Mainly dataflow, AI, type systems, constraint methods

• Bug-finding tools:
  – Including unsound analyses

• Automatic generation of static analyses

Course Structure

• Read significant/recent papers in the area
  – 35 minutes paper presentation
  – 25 minutes discussions

• Background
  – Dataflow analysis, optimizations (CS412)
  – Type systems (CS411, CS611)

• Requirements
  – Attend seminars
  – Read all papers, engage in discussions
  – Present 1-2 papers, start discussions
  – Do an implementation project
    • Or write a survey in a sub-area
A Flavor of Static Analysis

• Can an analysis determine that your program builds a tree? (not a DAG or a cyclic graph)

• Why should I care?
  – Program understanding/verification
  – Can parallelize programs with tree structures
  – Check memory safety

Example

rotate(tree * t) {
  tree *x = t->left;
  t->left = x->right;
  x->right = t;
  return x;
}

• Can the compile automatically prove that this code preserves the tree shape? How?

Example

rotate(tree * t) {
  tree *x = t->left;
  t->left = x->right;
  x->right = t;
  return x;
}

• Shape analysis
  – Uses an abstraction that tracks reference counts
  – Tree if all reference count are equal to 1

Find Bugs

rotate(tree * t) {
  tree *x = t->left;
  t->left = x->right;
  x->left = t;
  return x;
}

• Change "x->right" with "x->left"
• What goes wrong?

Materials

• Book:
  "Principles of Program Analysis",
  by Nielson, Nielson, Hankin, Springer 1999

• Web site
  http://www.cs.cornell.edu/courses/cs711

• Next time: Inter-procedural analysis
  "Precise Inter-Procedural Dataflow Analysis via Graph Reachability"
  by Reps, Horwitz, Sagiv, POPL’95