Program Analysis
(CSCI-4450/CSCI-6450)
Spring 2020

www.cs.rpi.edu/~milanova/csci4450/
Ana Milanova, Lally 314, milanova@cs.rpi.edu
Office hours: Wednesdays Noon-2:00PM or by appointment

Outline

- Logistics
  www.cs.rpi.edu/~milanova/csci4450/
- Program analysis, introduction
- Course topics, tools, and homework
- Introduction to dataflow analysis

Logistics

- Course webpage
  http://www.cs.rpi.edu/~milanova/csci4450
- Schedule, Notes, Reading
  - Schedule, lecture slides and assigned reading
  - Homework
    - Announcement and instructions when new homework assignment is on
  - Submitty
    - All homework submission and grades, forum, announcements
    - Check forum regularly for announcements

Logistics

- Recommended reading
  - Principles of Program Analysis by Flemming Nielson, Hanne Riis Nielson, and Chris Hankin.
  - Types and Programming Languages, by Benjamin C. Pierce.
  - MIT’s Open Courseware Class on Program Analysis
  - Papers and lecture notes

Logistics

- Syllabus
  www.cs.rpi.edu/~milanova/csci4450/syllabus.htm
  Topics, outcomes, policies and grading
- Take-home final: 25%
- Homework: 40%
- Paper presentation: 10%
- In-class quizzes (8): 20%
- Attendance and participation: 5%

Logistics

- Assignments are to be completed individually unless otherwise specified
- Quizzes are in-class, open-notes, and may be completed individually or in small groups
  - Makeup policy: you'll need an excuse note to makeup a quiz
Late Homework

- Homework assignments must be submitted in Submitty by noon on the due date
- You have 6 late days for the semester, with a max of 2 late days per assignment
- Exceptions to policy granted with an excuse note by your Class dean

Program Analysis

- Tools and techniques that help us reason about the runtime behavior of the program
  - Dynamic analysis – during program execution
    - Static instrumentation
    - Dynamic (binary) instrumentation (DBI)
  - Static analysis – before program execution
    - E.g., Java compiler’s definite-assignment-check
    - E.g., Type checking and type inference are forms of static analysis
    - E.g., Dafny-style verification
    - And many, many more!

Our main focus is Static Analysis

- Many techniques
  - Decades of research and Turing Awards!
  - Dataflow analysis and abstract interpretation
    - Kildall ’73, Kam and Ullman ’77, Cousot & Cousot ’77
  - Types and type-based analysis
    - Following Backus “Can Programming…” ’78
  - Axiomatic semantics (i.e., Hoare Logic)
    - C.A.R. Hoare’s “An Axiomatic Basis for Computer Programming”, ’69
  - Symbolic execution
    - Lori Clarke “A system to generate test data and symbolically execute programs” ’76

Static Analysis

- What this course is mostly about
  - How can we define the meaning of programs
  - How can we model behavior of programs, and prove theorems about programs
  - How can we use and build tools that automatically reason about programs
- Many applications

Applications

- Compiler optimization, traditional application
  - We’ll start with dataflow analysis
- Finding bugs, verifying the absence of bugs
- Designing languages that prevent bugs
- Refactoring and testing
- Improving energy efficiency
- Security and privacy
- Education. Submitty uses static analysis!

Examples of Properties Deducible by Static Analysis

- Can \( x \) ever be null at program point \( i: \mathbf{x}.\mathsf{m}() \)?
- Can \( y \) be different than 1 at program point \( i: \mathbf{x} = y \cdot 10 ? \)
- Can \( \text{buf}[n] \) cause out-of-bounds access?
- Does an app leak private data (e.g., phone number, phone identifier, location) to ad networks?
  - Answer: Yes!
Examples of Properties Deducible by Static Analysis

- What inputs avoid divide-by-zero at $x/y$?
  
  ```c
  // {x!=1 && x!=-2}
  y = x + 4;
  if (x > 0) {
    y = x*x - 1;
  } else {
    y = y + x;
  }
  // {y!=0}
  x = x/y;
  ```

- Formalism of Axiomatic Semantics (Hoare logic)
  
  - Different from dataflow and types
  - Allows us to specify program behavior with preconditions and postconditions that form logical assertions
  - Support complex logics
  - Enables reasoning about correctness

Examples of Properties Deducible by Static Analysis

```c
void fill_bowl(char* ingredients, char* bowl, int bowl_size) {
  printf("How many ingredients do you want to put in the bowl (max %u)\n", (bowl_size-1));
  int number;
  scanf("%u", &number);
  if (number > bowl_size) number = bowl_size - 1;
  // Copy at most bowl_size characters into the buffer
  for (int i=0; i <= number; i++) {
    bowl[i] = ingredients[i];
  }
}
```

Nature of Static Analysis

- To remain computable, static analysis must approximate. It is undecidable to find exactly what happens at runtime
  
  - Typically, analysis errs on the safe (sound) side --- that is, it over-approximates
    - E.g., analysis reports that $x$ at $x.m()$ may be null, even though it cannot ever be null
    - A type system rejects correct programs
  - Sometimes, analysis is unsafe (unsound) --- that is, it under-approximates

Nature of Static Analysis, cont.

- A static analysis is said to be safe (also, sound, correct) if it over-approximates, that is, takes into account every execution path and possibly some spurious paths
  
  - E.g., in previous example an analysis that reports that $y$ in $\{1, 2\}$ is safe, but so is an analysis that reports $y$ in $\{1, 2, 21\}$
Analysis Safety

- Safety is crucial when analysis enables compiler optimizations. Why?
  - E.g., an unsafe analysis may report that \( y \) is always 1 at \( z = y \times 10 \), while in fact there is an execution path where \( y \) is 2.
  - If the optimizing compiler changes \( z = y \times 10 \) to \( z = 10 \), then the optimized program produces an incorrect result when control takes the path where \( y \) is 2!

Analysis Safety

- Safety is often relinquished when analysis is used in static debugging tools. Why?
  - E.g., suppose we have a piece of code that contains 10 “true” buffer overflows
    - Safe analysis A reports 100 buffer overflows (all 10 “true” bugs plus 90 “false-positives”).
    - Unsafe analysis B reports 10 buffer overflows (8 “true” and 2 false-positives).
  - Which one would you take?

Analysis Precision

- Analysis precision refers to how “close” results are to what happens at runtime
  - E.g., in our running example, the analysis that reports \( y \) in \( \{1, 2\} \) is more precise than the one that reports \( y \) in \( \{1, 2, 21\} \).
  - Wide spectrum of static analyses and tradeoff between cost and precision

Outline

- Logistics
  - [www.cs.rpi.edu/~milanova/csci4450/](http://www.cs.rpi.edu/~milanova/csci4450/)
- Static analysis, introduction
- Course topics, tools and homework
- Introduction to Dataflow analysis

Course Topics

- Dataflow analysis
  - Control-flow graph, 4 classical dataflow analyses
  - Lattices, transfer functions, dataflow frameworks
  - Classical analyses: points-to analysis

- Abstract interpretation (a more powerful formalism than dataflow)
  - Abstract vs. concrete semantics
  - Galois connections

- Types and type-based analysis
  - Simply typed Lambda calculus
  - Type systems and type soundness
  - Simple type inference
  - Hindley Milner type inference

- Types for imperative languages
- Pluggable types
- Type-based information flow analysis (aka taint analysis)
Course Topics

- Axiomatic Semantics
  - You know this already: Hoare logic!
  - Logics to specify assertions (as you know them, \( P \) and \( Q \) in \( \{ P \} \) code \( \{ Q \} \))
  - SMT solvers and backwards reasoning (also known as Verification Condition Generation)

- Symbolic execution
  - SMT solvers and forward reasoning

Tools and Languages

- Soot
- Z3
- Ghidra and Klee
- Java
- C
- Haskell
- OCaml

Homework Assignments

- There will be 7-8 homework assignments
  - Each makes about 5-6% of your grade
  - Larger assignments are broken into 2-3 parts
  - Some are individual, some are team assignments

- Submitty!

Homework Assignments

- HW1
  - Problem set to practice dataflow analysis

- HW2-HW4
  - Classical OO analyses in Soot: the CHA, RTA, and XTA family of analyses

- HW5-HW6
  - Problem set to practice abstract interpretation/ type inference concepts
  - Implement simple type inference (and maybe Hindley Milner) in Haskell

Homework Assignments

- HW7-HW8
  - Implement a tiny C-program verifier using Z3
  - Implement a symbolic execution engine

- HW8-9
  - Can be larger projects you can define yourself

Dataflow Analysis
Key Papers

- Gary Kildall, “A Unified Approach to Global Program Optimization”, POPL 1973

Outline

- Motivation and origin of dataflow analysis: compiler optimization
- Overview of the compiler
- Classical compiler optimizations
- Control flow graph
- Reading:
  - Dragon Book, Chapter 9.1

Overview of the Compiler

- Phases of the compiler
  - Lexical Analyzer (scanner)
  - Syntax Analyzer (parser)
  - Semantic Analyzer and Intermediate Code Generator
  - Machine-Independent Code Optimizer
  - Code Generator
  - Machine-Dependent Code Optimizer

Overview of the Compiler

- An optimization is a semantics-preserving transformation

Classical Compiler Optimizations

- We will show the classical optimizations using an example Fortran loop
- Opportunities for optimization due to automatic generation of intermediate code

```
  sum = 0
  do 10 i = 1, n
    10 sum = sum + a[i]*a[i]
```

Three Address Code

Intermediate Representation (IR)

```
1. sum = 0    initialize sum
2. i = 1      initialize loop counter
3. if i > n goto 15 loop test, check for limit
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
8. t5 = i * 4
9. t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
12. sum = t8  increment sum
13. i = i + 1 increment loop counter
14. goto 3    increment loop counter
15. ...
```
Control Flow Graph (CFG)

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) – 4
5. t2 = i*4
6. t3 = t1[t2]
7. t4 = addr(a) – 4
8. t5 = i*4
9. t6 = t4[t5]
10. t7 = t3*t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15. ...

Common Subexpression Elimination

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) – 4
5. t2 = i*4
6. t3 = t1[t2]
7. t4 = addr(a) – 4
8. t5 = i*4
9. t6 = t4[t5]
10. t7 = t3*t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
13. i = i + 1
14. goto 3
15. ...

After Common Subexpression Elimination

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) – 4
5. t2 = i*4
6. t3 = t1[t2]
10a t7 = t3*t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. ...

Copy Propagation

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) – 4
5. t2 = i*4
6. t3 = t1[t2]
10a t7 = t3*t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
13. i = i + 1
14. goto 3
15. ...

After Copy Propagation

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) – 4
5. t2 = i*4
6. t3 = t1[t2]
10a t7 = t3*t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. ...

Invariant Code Motion

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) – 4
5. t2 = i*4
6. t3 = t1[t2]
10a t7 = t3*t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
13. i = i + 1
14. goto 3
15. ...

Copy Propagation

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) – 4
5. t2 = i*4
6. t3 = t1[t2]
10a t7 = t3*t3
10a t7 = t3*t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
13. i = i + 1
14. goto 3
After Invariant Code Motion

1. sum = 0
2. i = 1
2a t1 = addr(a) - 4
3. if i > n goto 15
5. t2 = i * t1
6. t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. ...

Strength Reduction

1. sum = 0
2. i = 1
2a t1 = addr(a) - 4
2b t2 = i * 4
3. if i > n goto 15
5. t2 = t3 * t7
6. t3 = t1[t2]
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. ...

After Strength Reduction

1. sum = 0
2. i = 1
2a t1 = addr(a) - 4
2b t2 = i * 4
2c t9 = n * 4
3. if t2 > t9 goto 15
6. t3 = t1[t2]
10a t7 = t3 * t7
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. ...

Test Elision and Induction Variable Elimination

1. sum = 0
2. i = 1
2a t1 = addr(a) - 4
2b t2 = i * 4
2c t9 = n * 4
3. if t2 > t9 goto 15
6. t3 = t1[t2]
10a t7 = t3 * t7
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. ...

After Test Elision and Induction Variable Elimination

1. sum = 0
2. i = 1
2a t1 = addr(a) - 4
2b t2 = i * 4
2c t9 = n * 4
3a if t2 > t9 goto 15
6. t3 = t1[t2]
10a t7 = t3 * t7
11a sum = sum + t7
11b t2 = t2 + 4
13. i = i + 1
14. goto 3a
15. ...

Constant Propagation and Dead Code Elimination

1. sum = 0
2. i = 1
2a t1 = addr(a) - 4
2b t2 = i * 4
2c t9 = n * 4
3a if t2 > t9 goto 15
6. t3 = t1[t2]
10a t7 = t3 * t7
11a sum = sum + t7
11b t2 = t2 + 4
14. goto 3a
15. ...

Spring 20 CSCI 4450/6450, A Milanova
New Control Flow Graph

1. sum = 0
2. t1 = addr(a) - 4
3. t9 = n * 4
4. t2 = 4
5. if t2 > t9 goto 11
6. t3 = t1[t2]
7. t7 = t3 * t3
8. sum = sum + t7
9. t2 = t2 + 4
10. goto 5

Classical Compiler Optimizations

To summarize
- Common subexpression elimination
- Copy propagation
- Strength reduction
- Test elision and induction variable elimination
- Constant propagation
- Dead code elimination
- Dataflow analysis enables these optimizations

Building the Control Flow Graph

Build the CFG from linear 3-address code:

- Step 1: partition code into basic blocks
  - Basic blocks are the nodes in the CFG
- Step 2: add control flow edges

Question. Find the Leader Statements

- sum = 0
- i = 1
- if i > n goto 15
- t1 = addr(a) - 4
- t2 = i * 4
- t3 = t1[t2]
- t4 = addr(a) - 4
- t5 = i * 4
- t6 = t5[t5]
- t7 = t3*t6
- t8 = sum + t7
- sum = t8
- i = i + 1
- goto 3

Step 1. Partition Code Into Basic Blocks

1. Determine the leader statements:
   (i) First program statement
   (ii) Targets of goto’s
   (iii) Any statement that follows a goto
2. For each leader, its basic block consists of the leader and all statements up to, but not including, the next leader or the end of the program

Step 2. Add Control Flow Edges

- There is a directed edge from basic block B_1 to block B_2 if B_2 can immediately follow B_1 in some execution
- Determine edges as follows:
  - There is an edge from B_1 to B_2 if B_2 follows B_1 in three-address code, and B_1 does not end in an unconditional goto
  - There is an edge from B_1 to B_2 if there is a goto from the last statement in B_1 to the first statement in B_2
Question. Add Control Flow Edges

1. \( \text{sum} = 0 \)
2. \( i = 1 \)
3. if \( i > n \) goto 15
4. \( t_1 = \text{addr}(a) - 4 \)
5. \( t_2 = i*4 \)
6. \( t_3 = t_1[t_2] \)
7. \( t_4 = \text{addr}(a) - 4 \)
8. \( t_5 = i*4 \)
9. \( t_6 = t_3[t_5] \)
10. \( t_7 = t_3[t_6] \)
11. \( t_8 = \text{sum} + t_7 \)
12. \( \text{sum} = t_8 \)
13. \( i = i + 1 \)
14. goto 3

Next Class

- Dataflow analysis
- Four classical dataflow analysis problems