Outline

- Logistics
  - www.cs.rpi.edu/~milanova/csci4450/

- Program analysis, introduction

- Course topics, tools and homework

- Introduction to Dataflow analysis

Spring 18 CSCI 4450/6450, A Milanova

Logistics

- Course webpage
  - http://www.cs.rpi.edu/~milanova/csci4450
- Announcements – check regularly
- Schedule, Notes, Reading
  - Schedule, lecture slides and assigned reading
- Homework
  - Announcement and instructions when new homework assignment is on
- Submitty
  - All homework assignments, homework submission and grades
  - New this term! Submitty discussion board! Thanks Prof. Cutler!

Logistics

- Recommended reading
  - Principles of Program Analysis by Flemming Nielson, Hanne Riis Nielson, and Chris Hankin.
  - Types in 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 must be completed individually unless otherwise noted
- Quizzes are in-class, open-notes, and may be completed individually or in small groups
- Makeup policy is stricter than my usual policy on quizzes: you’ll need an excuse note to makeup a quiz
Late Homework

- Homework assignments must be submitted in Submitty by 2pm 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 run-time 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 is a form 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

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
- Improving security and privacy
- Education. Submitty uses static analysis!

Examples of Properties Deducible by Static Analysis

- Can \( x \) ever be null at program point \( i: \text{main}() \)?
- Can \( y \) be different than 1 at program point \( i: x = y*10 \)?
- Can \( n \) at \( x[n] \) 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?
  \( x \neq 1 \land x \neq -2 \)
- 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

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
  - E.g., in previous example the analysis that reports y in \( \{1, 2\}\) is safe, but so is the one 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*10, while in fact there is an execution path that sets \( y \) to 10.
  - If the optimizing compiler changes z=y*10 to z=10, the program produces incorrect result along the path when \( y \) is 10!

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” null-pointer dereferences
    - Safe analysis A reports 100 potential null-pointer dereferences (all 10 “true” bugs and 90 “false-positives”).
    - Unsafe analysis B reports 10 potential null-pointer dereferences (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 $\{1, 2, 21\}$.
  - Typically, we use the term precision with safe analysis (safe analysis has 100% recall)
  - Wide spectrum of analyses and tradeoff between cost and precision

Outline

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

Course Topics

- Dataflow analysis
  - Lattices, transfer functions, dataflow frameworks
  - Classical analyses: points-to analysis
- Abstract interpretation (powerful formalism, generalization of DF)
  - Abstract vs. concrete semantics
  - Galois connections

Course Topics

- Types and type-based analysis
  - Simply typed Lambda calculus
  - Type systems and type soundness
  - Hindley-Milner type inference
  - Types for imperative languages
  - Pluggable types
  - Type-based information flow analysis (aka taint analysis)

Course Topics

- Axiomatic Semantics
  - You know already: Hoare logic!
  - Logics to specify assertions (as you know them, $P$ and $Q$ in $\{ P \}$ code $\{ Q \}$)
  - SMT solvers and proving Hoare triples
  - Symbolic execution

Tools

- Soot
- Haskell
- Checkers
- Z3
There will be 8-9 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!

HW1
- Practice dataflow analysis and abstract interpretation

HW2-HW4
- Implement classical OO analyses in Soot: the CHA, RTA, and XTA family of analyses
- Implement static instrumentation

HW5-HW6
- Practice functional programming
- Implement a type inference engine in Haskell

HW7-HW8
- Implement a pluggable type system, JWild, to infer Java Wildcards in Checkers or Soot
- Write a paper about it!

HW9
- Learn about SAT and SMT solvers
- Implement a tiny program verifier in Z3, or
- Encode JWild constraints in an SAT/SMT solver to infer an optimal JWild type assignment

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 graphs

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

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]  a[i]
7. t4 = addr(a) – 4
8. t5 = i * 4
9. t6 = t4[t5]  a[i]
10. t7 = t3 * t6  a[i]*a[i]
11. t8 = sum + t7 increment sum
12. if i > n goto 15 increment loop counter
13. i = i + 1
14. goto 3
```

Control Flow Graph (CFG)

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

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

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
11. t8 = sum + t7
12. sum = t8
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
2a t1 = addr(a) - 4
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. ...

Strength Reduction

1. sum = 0
2. i = 1
2a t1 = addr(a) - 4
3. if i > n goto 15
2b t2 = i*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. ...

After Invariant Code Motion

1. sum = 0
2a t1 = addr(a) - 4
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. ...

Spring 18 CSCI 4450/6450, A Milanova
**After Strength Reduction**

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

**Test Elision and Induction Variable Elimination**

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

**After Test Elision and Induction Variable Elimination**

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

**Constant Propagation and Dead Code Elimination**

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

**New Control Flow Graph**

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

**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

---

Spring 18 CSCI 4450/6450, A Milanova
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

Aside: in Principles of Software, we built a CFG from structural IR:

\[
S ::= x = y \text{ op } z | S;S | \text{ if (b) then } S \text{ else } S | \text{ while (b) } S
\]

**Step 1. Partition Code Into Basic Blocks**

1. Determine the **leader** statements:
   - (i) First program statement
   - (ii) Targets of conditional or unconditional goto's
   - (iii) Any statement following 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 sequence

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. Find the Leader Statements**

1. \( \text{sum} = 0 \)
2. \( i = 1 \)
3. \( \text{if } i > n \text{ goto 15} \)
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. \( t_6 = t_5[t_5] \)
10. \( t_7 = t_3*t_6 \)
11. \( t_8 = \text{sum} + t_7 \)
12. \( \text{sum} = t_8 \)
13. \( i = i + 1 \)
14. \( \text{goto 3} \)
15. ...

**Question. Add Control Flow Edges**

1. \( \text{sum} = 0 \)
2. \( i = 1 \)
3. \( \text{if } i > n \text{ goto 15} \)
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. \( t_6 = t_5[t_5] \)
10. \( t_7 = t_3*t_6 \)
11. \( t_8 = \text{sum} + t_7 \)
12. \( \text{sum} = t_8 \)
13. \( i = i + 1 \)
14. \( \text{goto 3} \)
15. ...

**Next Class**

- Dataflow analysis
- Four classical dataflow analysis problems