Research in General

- Identifying and defining problems is as important as finding a solution
  - Formulate the problem
  - Why it is important
  - Why the solution potentially works
  - What are the tradeoffs

- Where to find important problems:
  - Internet of things: device is resource constrained, connected to the Internet
  - Big data
  - Obsolete data format
Status Check

- Install Soot and LLVM
- Read the tutorial
- Try running some examples
- Start writing your own analysis
SOOT and LLVM Exercise: Weeks 6-7 (Due Oct 10)

Assignment

- Generate Callgraphs, CFGs, and Dependence Graphs for 2 programs of Java and 2 programs of C/C++
- Visualization using Dotty (recommended)

Deliverables

- Soot and LLVM code
- A report (within 2 page) on:
  - Your experience with SOOT and LLVM, what you like, what you don’t like, how long it takes you to learn, what are the most challenges
  - Data collected from the graph, see the following table.
  - Examples: part of the graph

<table>
<thead>
<tr>
<th>benchmark</th>
<th>size (LOC)</th>
<th>node</th>
<th>edge</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The History of Control Flow Analysis

- Turing award for pioneering contributions to the theory and practice of optimizing compiler techniques, awarded 2006
What is Control Flow Analysis (CFA)?

- Determining the execution order of program statements or instructions
- Control flow graph (CFG) specifies all possible execution paths
- Important control flow constructs (program constructs important to control flow)
  - basic block: a basic block is a maximal sequence of consecutive statements with a single entry point, a single exit point, and no internal branches
  - loops
  - method calls: program analysis to identify the receiver of the function calls – e.g., virtual functions, function pointers: abstract interpretation, type systems and constraint solving
  - exception handling
What CFG implies?

- CFGs are commonly used to propagate information between nodes (Dataflow analysis)
- The existence of back edges / cycles in flow graphs indicates that we may need to traverse the graph more than once:
  - Iterative algorithms: when to stop? How quickly can we stop?
Dominance

Node $d$ of a CFG *dominates* node $n$ if every path from the entry node of the graph to $n$ passes through $d$, noted as $d$ dom $n$

- $\text{Dom}(n)$: the set of dominators of node $n$
- Every node dominates itself: $n \in \text{Dom}(n)$
- Node $d$ *strictly dominates* $n$ if $d \in \text{Dom}(n)$ and $d \neq n$
- Dominance’ based loop recognition: entry of a loop dominates all nodes in the loop

- Each node $n$ has a unique *immediate dominator* $m$ which is the last dominator of $n$ on any path from the entry to $n$ ($m$ idom $n$), $m \neq n$
- The immediate dominator $m$ of $n$ is the strict dominator of $n$ that is closest to $n$
Dominator Example

<table>
<thead>
<tr>
<th>Block</th>
<th>Dom</th>
<th>IDom</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1}</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>{1,2}</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>{1,3}</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>{1,3,4}</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>{1,3,4,5}</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>{1,3,4,6}</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>{1,3,4,7}</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>{1,3,4,7,8}</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>{1,3,4,7,8,9}</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>{1,3,4,7,8,10}</td>
<td>8</td>
</tr>
</tbody>
</table>
In a dominator tree, a node’s parent is its immediate dominator.
Post-Dominance

- Node $d$ of a CFG post dominates node $n$ if every path from $n$ to the exit node passes through $d$ ($d \ pdom \ n$)
- $Pdom(n)$: the set of post dominators of node $n$
- Every node post dominates itself: $n \in Pdom(n)$
- Each node $n$ has a unique immediate post dominator
Post-Dominator Example

```
<table>
<thead>
<tr>
<th>Block</th>
<th>Pdom</th>
<th>IPdom</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{3,4,7,8,10,exit}</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>{2,3,4,7,8,10,exit}</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>{3,4,7,8,10,exit}</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>{4,7,8,10,exit}</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>{5,7,8,10,exit}</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>{6,7,8,10,exit}</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>{7,8,10,exit}</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>{8,10,exit}</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>{1,3,4,7,8,10,exit}</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>{10,exit}</td>
<td>exit</td>
</tr>
</tbody>
</table>
```
Natural Loops

Use dominators to discover loops for optimization, implemented in current compiler optimizations

- A back edge is an edge $a \rightarrow b$ whose head $b$ dominates its tail $a$.
- A loop must have a single entry point called header. This entry node dominates all nodes in the loop.
- There must be a back edge that enters the loop header. Otherwise, it is not possible for the flow of control to return to the header directly from the "loop".
- A natural loop consisting of all nodes $x$, where $b$ dom $x$ and there is a path from $x$ to $b$ not containing $b$
Natural Loop Example

Why neither \(\{3,4\}\) nor \(\{4,5,6,7\}\) is a natural loop?
An inner loop is a loop that contains no other loops

- Good optimization candidate
- The inner loop of the previous example: 7,8,10
Dynamic Dispatch problems

- Function pointers
- Object oriented languages
- Functional languages

Problem: which implementation of the function will be invoked at the callsite
Dynamic Dispatch Problems in C++

```cpp
class A {
public:
    virtual void f();
    // ...
};

class B: public A {
public:
    virtual void f();
    // ...
};

int main()
{
    A *pa = new B();
    // pa->f();
    // ...
}
```

To which implementation the call f bound to? Dynamic dispatch: the binding is determined at runtime, based on the input of the program and execution paths.
int main()
{
    A a; // An A instance is created on the stack
    B b; // A B instance, also on the stack

    a = b; // Only the A part of 'b' is copied into a.

    a.f(); // Static dispatch. This determines the binding
             // of f to A's f and this is done at compile time.
}

Static dispatch: the binding is determined at the compiler time.
Function Pointers

```c
#include <math.h>
#include <stdio.h>

// Function taking a function pointer as an argument
double compute_sum(double (*funcep)(double), double lo, double hi)
{
    double sum = 0.0;

    // Add values returned by the pointed-to function '*funcep'
    for (int i = 0; i <= 100; i++)
    {
        double x, y;

        // Use the function pointer 'funcep' to invoke the function
        x = i/100.0 * (hi - lo) + lo;
        y = (*funcep)(x);
        sum += y;
    }
    return (sum/100.0);
}

int main(void)
{
    double (*fp)(double); // Function pointer
    double sum;

    // Use 'sin()' as the pointed-to function
    fp = sin;
    sum = compute_sum(fp, 0.0, 1.0);
    printf("sum(sin): %f\n", sum);

    // Use 'cos()' as the pointed-to function
    fp = cos;
    sum = compute_sum(fp, 0.0, 1.0);
    printf("sum(cos): %f\n", sum);
    return 0;
}
```

Control flow analysis: determining dataflow or values of variables
An Overview of Research

Between 1990-2000:

- **Class hierarchy analysis** (newly defined types) and rapid type analysis (RTA) (analyzing instantiation of the object) – resolve 71% virtual function calls [1]
- Theoretical framework for call graph constructions for object-oriented programs [2]
- Pointer target tracking [4]
- Callgraph analysis [3]
- Variable type and declared type analysis [6] ...
An Example - Class Hierarchy Analysis

```java
class A extends Object {
    String m() {
        return(this.toString());
    }
}
class B extends A {
    String m() { ... }
}
class C extends A {
    String m() { ... }
    public static void main(...) {
        A a = new A();
        B b = new B();
        String s;
        ...
        s = a.m();
        s = b.m();
    }
}
(a) Example Program
```

(b) Class Hierarchy and Call Graph
An Example - Rapid Type Analysis

class A extends Object {
    String m() {
        return(this.toString());
    }
}

class B extends A {
    String m() { ... }
}

class C extends A {
    String m() { ... }
    public static void main(...) {
        A a = new A();
        B b = new B();
        String s;

        ...  
        s = a.m();
        s = b.m();
    }
}

(a) Example Program

(b) Class Hierarchy and Call Graph
An Example - Variable Type Analysis

(a) Program

A a1, a2, a3;
B b1, b2, b3;
C c;

a1 = new A();
a2 = new A();
b1 = new B();
b2 = new B();
c = new C();

a1 = a2;
a3 = a1;
a3 = b3;
b3 = (B) a3;
b1 = b2;
b1 = c;

(b) Nodes and Edges

(c) Initial Types

(d) Strongly-connected components

(e) final solution
try
{
    divide(10, 0);
}
catch(int i)
{
    if(i==DivideByZero)
    {
        cerr<<"Divide by zero error";
    }
}
try {
    // guarded section
    . . .
}
catch (ExceptionType1 t1) {
    // handler for ExceptionType1
    . . .
}
catch (ExceptionType2 t2) {
    // handler for ExceptionType2
    . . .
}
. . .
catch (Exception e) {
    // handler for all exceptions
    . . .
}
finally {
    // cleanup code
    . . .
}
```java
public void openFile()
{
    try {
        // constructor may throw FileNotFoundException
        FileReader reader = new FileReader("someFile");
        int i = 0;
        while (i != -1) {
            // reader.read() may throw IOException
            i = reader.read();
            System.out.println((char) i);
        }
        reader.close();
        System.out.println("--- File End ---");
    } catch (FileNotFoundException e) {
        // do something clever with the exception
    } catch (IOException e) {
        // do something clever with the exception
    }
}

public void openFile()
{
    FileReader reader = null;
    try {
        reader = new FileReader("someFile");
        int i = 0;
        while (i != -1) {
            i = reader.read();
            System.out.println((char) i);
        }
    } catch (IOException e) {
        // do something clever with the exception
    } finally {
        if (reader != null) {
            try {
                reader.close();
            } catch (IOException e) {
                // do something clever with the exception
            }
        }
    }
    System.out.println("--- File End ---");
}
```
Frequency of Occurrence of Exception Handling Statements in Java [5]

<table>
<thead>
<tr>
<th>Subject</th>
<th>Description</th>
<th>Number of classes</th>
<th>Number of methods</th>
<th>Methods with EH constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>Framework for compiler construction</td>
<td>175</td>
<td>1663</td>
<td>175 (10.5%)</td>
</tr>
<tr>
<td>debug</td>
<td>Sun's Java debugger</td>
<td>45</td>
<td>416</td>
<td>80 (19.2%)</td>
</tr>
<tr>
<td>jaba</td>
<td>Architecture for analysis of Java bytecode</td>
<td>312</td>
<td>1615</td>
<td>200 (12.4%)</td>
</tr>
<tr>
<td>jar</td>
<td>Sun's Java archive tool</td>
<td>8</td>
<td>89</td>
<td>14 (15.7%)</td>
</tr>
<tr>
<td>jas</td>
<td>Java bytecode assembler</td>
<td>118</td>
<td>408</td>
<td>59 (14.5%)</td>
</tr>
<tr>
<td>jasmine</td>
<td>Java Assembler Interface</td>
<td>99</td>
<td>627</td>
<td>54 (8.6%)</td>
</tr>
<tr>
<td>java_cup</td>
<td>LLR parser generator for Java</td>
<td>35</td>
<td>360</td>
<td>32 (8.9%)</td>
</tr>
<tr>
<td>javac</td>
<td>Sun's Java compiler</td>
<td>154</td>
<td>1395</td>
<td>175 (12.5%)</td>
</tr>
<tr>
<td>javadoc</td>
<td>Sun's HTML document generator</td>
<td>3</td>
<td>99</td>
<td>17 (17.2%)</td>
</tr>
<tr>
<td>javasim</td>
<td>Discrete event process-based simulation package</td>
<td>29</td>
<td>216</td>
<td>37 (17.1%)</td>
</tr>
<tr>
<td>jdb</td>
<td>Parser and lexer generator</td>
<td>45</td>
<td>543</td>
<td>55 (10.1%)</td>
</tr>
<tr>
<td>jdk-api</td>
<td>Sun's JDK API</td>
<td>712</td>
<td>5038</td>
<td>582 (11.6%)</td>
</tr>
<tr>
<td>jedit</td>
<td>Text editor</td>
<td>439</td>
<td>2048</td>
<td>173 (8.4%)</td>
</tr>
<tr>
<td>jflex</td>
<td>Lexical-analyzer generator</td>
<td>54</td>
<td>417</td>
<td>31 (7.4%)</td>
</tr>
<tr>
<td>jlex</td>
<td>Lexical-analyzer generator for Java</td>
<td>20</td>
<td>134</td>
<td>4 (3.0%)</td>
</tr>
<tr>
<td>jole</td>
<td>Environment for load-time transformation of Java classes</td>
<td>83</td>
<td>834</td>
<td>90 (10.8%)</td>
</tr>
<tr>
<td>sablecc</td>
<td>Framework for generating compilers and interpreters</td>
<td>342</td>
<td>2194</td>
<td>106 (4.8%)</td>
</tr>
<tr>
<td>swing-api</td>
<td>Sun's Swing API</td>
<td>1588</td>
<td>13304</td>
<td>583 (4.7%)</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>3951</td>
<td>30400</td>
<td>2467 (8.1%)</td>
</tr>
</tbody>
</table>
Analysis and Testing Program With Exception Handling Constructs [5]

1. try block raises no exception
2. try block raises no exception; finally block specified
3. try block raises exception; catch block does not handle exception; no finally block
4. try block raises exception; catch block does not handle exception; finally block specified
5. try block raises exception; catch block handles exception
6. catch block handles exception; finally block specified
7. catch block handles exception; no finally block
8. catch block handles exception, raises another exception; finally block specified
9. catch block handles exception; raises another exception no finally block
10. finally block raises no exception
11. finally block raises exception
12. finally block propagates previous exception, or raises another exception
13. nested block propagates exception; catch block handles exception
14. nested block propagates exception; catch block does not handle exception; finally block specified
Analysis and Testing Program With Exception Handling Constructs [5]

Figure 9: Interprocedural control flow in exception-handling constructs.
public class VendingMachine {
    private int tolValue;
    private int currValue;
    private int currAttempts;
    private Dispenser d;

    public VendingMachine() {
        tolValue = 0;
        currValue = 0;
        currAttempts = 0;
        d = new Dispenser();
    }

    public void insert(Coin coin) {
        int value = valueOf(coin);
        if (value == 0) {
            throw new IllegalCoinException();
        }
        currValue += value;
        showMsg("current value = " + currValue);
    }

    public void returnCoins() {
        if (currValue == 0) {
            throw new ZeroValueException();
        }
        showMsg("Take your coins");
        currValue = 0;
        currAttempts = 0;
    }

    public void vend(int selection) {
        if (currValue == 0) {
            throw new ZeroValueException();
        }
        d.dispense(currValue, selection);
        int bal = d.valueOf(selection);
        currValue -= bal;
        currValue = bal;
        returnCoins();
    }

    public void main() {
        VendingMachine vm = new VendingMachine();
        while (true) {
            try {
                switch (action) {
                    case INSERT: vm.insert(coin);
                    case VEND: vm.vend(selection);
                    case RETURN: vm.returnCoins();
                }
            } catch (SelectionException s) {
                showMsg("Transaction aborted");
                vm.returnCoins();
            } catch (IllegalCoinException i) {
                showMsg("Illegal coin");
                vm.returnCoins();
            } catch (IllegalAmountException i) {
                int val = 1.getValue();
                showMsg("Enter more coins" + val);
            } catch (ZeroValueException z) {
                showMsg("Value is zero. Enter coins");
            }
        }
    }
}

public class Dispenser {
    public void dispense(int sel) {
        Exception e = null;
        if (sel < MIN_SELECTION || sel > MAX_SELECTION) {
            showMsg("selection " + sel + " is invalid");
            e = new IllegalArgumentException();
        } else {
            if (!available(sel)) {
                showMsg("selection +sel+ is unavailable");
                e = new SelectionNotAvailableException();
            } else {
                int val = value(sel);
                if (currVal < val) {
                    e = new IllegalAmountException(val-currVal);
                }
            }
        }
        if (e != null) {
            throw e;
        }
        showMsg("Take selection");
    }
}
Figure 10: ICFG for the vending-machine program.
David F. Bacon and Peter F. Sweeney.  
Fast static analysis of C++ virtual function calls.  

David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers.  
Call graph construction in object-oriented languages.  

Mary W. Hall and Ken Kennedy.  
Efficient call graph analysis.  

Jon Loeliger, Robert Metzger, Mark Seligman, and Sean Stroud.  
Pointer target tracking—an empirical study.  

Saurabh Sinha and Mary Jean Harrold.  
Analysis and testing of programs with exception handling constructs.  