Lecture 4. Control Flow Analysis

Wei Le

2014.10

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

benchmark	size (LOC)	node	edge

The History of Control Flow Analysis

- 1970, Frances Allen, *Control Flow Analysis* – CFG
- 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?

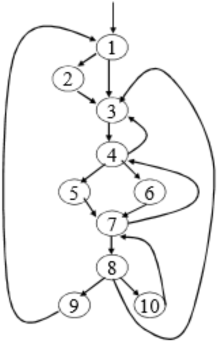
Dominance

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

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

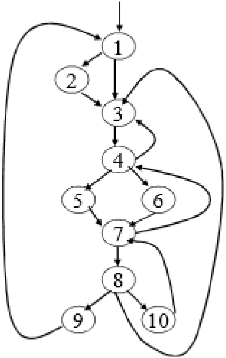
- Each node n has a unique *immediate dominator* m which is the last dominator of n on any path from the entry to n (m $idom$ n), $m \neq n$
- The immediate dominator m of n is the strict dominator of n that is closest to n

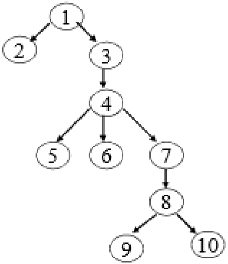
Dominator Example



Block	Dom	IDom
1	{1}	—
2	{1,2}	1
3	{1,3}	1
4	{1,3,4}	3
5	{1,3,4,5}	4
6	{1,3,4,6}	4
7	{1,3,4,7}	4
8	{1,3,4,7,8}	7
9	{1,3,4,7,8,9}	8
10	{1,3,4,7,8,10}	8

Dominator Tree



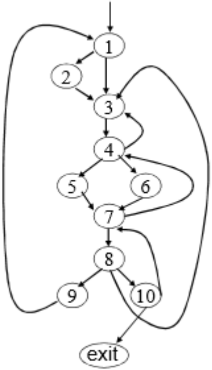


- In a dominator tree, a node's parent is its immediate dominator

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



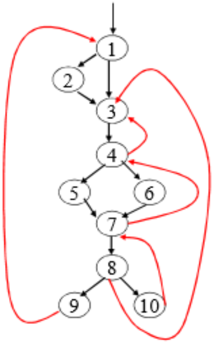
Block	Pdom	IPdom
1	{3,4,7,8,10,exit}	3
2	{2,3,4,7,8,10,exit}	3
3	{3,4,7,8,10,exit}	4
4	{4,7,8,10,exit}	7
5	{5,7,8,10,exit}	7
6	{6,7,8,10,exit}	7
7	{7,8,10,exit}	8
8	{8,10,exit}	10
9	{1,3,4,7,8,10,exit}	1
10	{10,exit}	exit

Natural Loops

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

- A back edge is an edge $a \to b$ 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



Back edge	Natural loop
10→7	{7,10,8}
7→4	{4,7,5,6 10,8}
4→3	{3,4,7,5,6,10,8}
8→3	
9→1	{1,9,8,7,5,6, 10,4,3,2}

- Why neither {3,4} nor {4,5,6,7} is a natural loop?

Inner 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++

```
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.

Compared to Static Dispatch

```
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 (*funcp)(double), double lo, double hi)
{
    double  sum = 0.0;

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

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

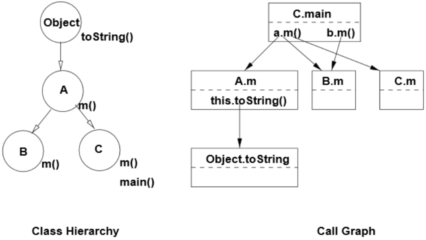
```
class A extends Object {
  String m() {
    return(this.toString());
  }
}

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

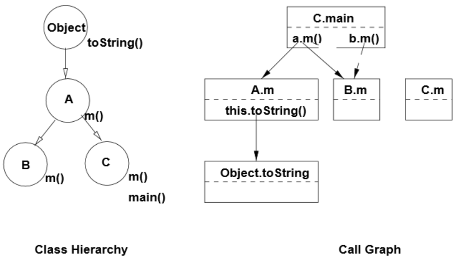
```
class A extends Object {
  String m() {
    return(this.toString());
  }
}

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

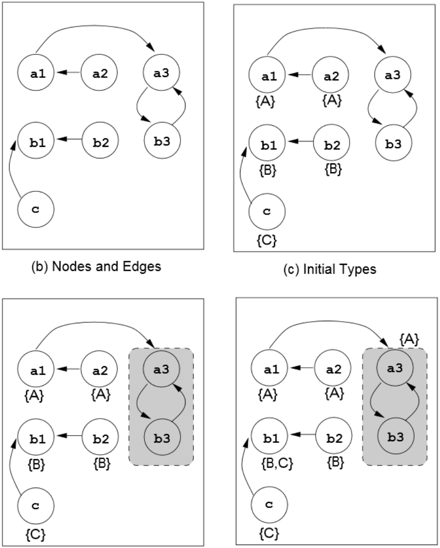
An Example - Variable Type Analysis

```
A 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;
```

(a) Program



(b) Nodes and Edges

(c) Initial Types

(d) Strongly-connected components

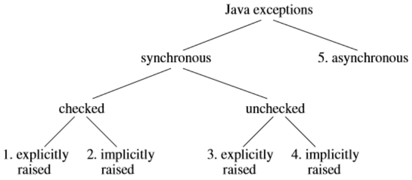
(e) final solution

Exceptional Handling: C++

```
try
{
    divide(10, 0);
}
catch(int i)
{
    if(i==DivideByZero)
    {
        cerr<<"Divide by zero error";
    }
}
```

Exceptional Handling: Java

```
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
   . . .
}
```



Exceptional Handling: Java

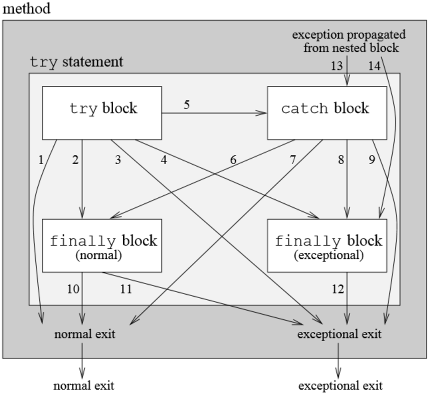
```java
public void openFile(){
    try {
        // constructor may throw FileNotFoundException
        FileReader reader = new FileReader("someFile");
        int i=0;
        while(i != -1){
            //reader.read() may throw IOException
            i = reader.read();
            System.out.println((char) 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
    }
}
```

```java
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]

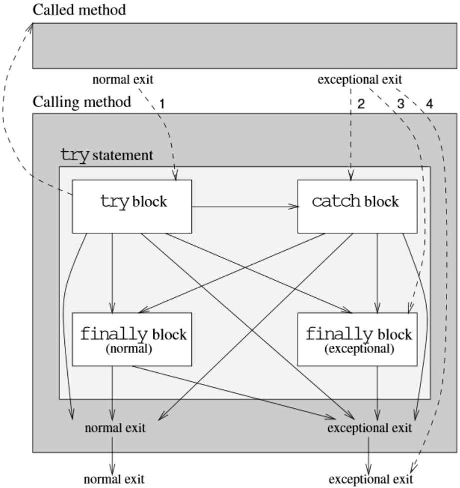
Subject		Number of classes	Number of methods	Methods with EH constructs
Name	Description			
antlr	Framework for compiler construction	175	1663	175 (10.5%)
debug	Sun's Java debugger	45	416	80 (19.2%)
jaba	Architecture for analysis of Java bytecode	312	1615	200 (12.4%)
jar	Sun's Java archive tool	8	89	14 (15.7%)
jas	Java bytecode assembler	118	408	59 (14.5%)
jasmine	Java Assembler Interface	99	627	54 (8.6%)
java_cup	LALR parser generator for Java	35	360	32 (8.9%)
javac	Sun's Java compiler	154	1395	175 (12.5%)
javadoc	Sun's HTML document generator	3	99	17 (17.2%)
javasim	Discrete event process-based simulation package	29	216	37 (17.1%)
jb	Parser and lexer generator	45	543	55 (10.1%)
jdk-api	Sun's JDK API	712	5038	582 (11.6%)
jedit	Text editor	439	2048	173 (8.4%)
jflex	Lexical-analyzer generator	54	417	31 (7.4%)
jlex	Lexical-analyzer generator for Java	20	134	4 (3.0%)
joie	Environment for load-time transformation of Java classes	83	834	90 (10.8%)
sablecc	Framework for generating compilers and interpreters	342	2194	106 (4.8%)
swing-api	Sun's Swing API	1588	12304	583 (4.7%)
Total		3951	30400	2467 (8.1%)

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]



1 called method propagates no exceptions

2 called method propagates exception; catch block in calling method handles exception

3 called method propagates exception; catch block in calling method does not handle exception; finally block specified in calling method

4 called method propagates exception; catch block in calling method does not handle exception; no finally block in calling method

Figure 9: Interprocedural control flow in exception-handling constructs .

```
public class VendingMachine {

  private int totValue;
  private int currValue;
  private int currAttempts;
  private Dispenser d;

  public VendingMachine() {
 1  totValue = 0;
 2  currValue = 0;
 3  currAttempts = 0;
 4  d = new Dispenser();
  }

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

  public void returnCoins() {
10  if ( currValue == 0 ) {
11    throw new ZeroValueException();
    }
12  showMsg( "Take your coins" );
13  currValue = 0;
14  currAttempts = 0;
  }
```

```
  public void vend( int selection ) {
15  if ( currValue == 0 ) {
16    throw new ZeroValueException();
    }
    try {
17    d.dispense( currValue, selection );
18    int bal = d.value( selection );
19    totValue += currValue - bal;
20    currValue = bal;
21    returnCoins();
    }
22  catch( SelectionException s ) {
23    currAttempts++;
24    if ( currAttempts < MAX_ATTEMPTS ) {
25      showMsg( "Enter selection again" );
      }
      else {
26      currAttempts = 0;
27      throw s;
      }
    }
28  catch( ZeroValueException z ) {
    }
  }
}
```

```
public class Dispenser {
  public void dispense( int currVal, int sel ) {
29  Exception e = null;
30  if ( sel < MIN_SELECTION || sel > MAX_SELECTION ) {
31    showMsg( "selection "+sel+" is invalid" );
32    e = new IllegalSelectionException();
    }
    else {
33    if ( !available( sel ) ) {
34      showMsg( "selection "+sel+" is unavailable" );
35      e = new SelectionNotAvailableException();
      }
      else {
36      int val = value( sel );
37      if ( currVal < val ) {
38        e = new IllegalAmountException( val-currVal );
        }
      }
    }
39  if ( e != null ) {
40    throw e;
    }
41  showMsg( "Take selection" );
  }
}
```

```
  public static void main() {
42  VendingMachine vm = new VendingMachine();
43  while ( true ) {
      try {
        try {
44        switch( action ) {
45          case INSERT: vm.insert( coin );
46          case VEND:   vm.vend( selection );
47          case RETURN: vm.returnCoins();
          }
        }
48      catch( SelectionException s ) {
49        showMsg( "Transaction aborted" );
50        vm.returnCoins();
        }
51      catch( IllegalCoinException i ) {
52        showMsg( "Illegal coin" );
53        vm.returnCoins();
        }
54      catch( IllegalAmountException i ) {
55        int val = i.getValue();
56        showMsg( "Enter more coins"+val );
        }
      }
57    catch( ZeroValueException z ) {
58      showMsg( "Value is zero.  Enter coins" );
      }
    }
```

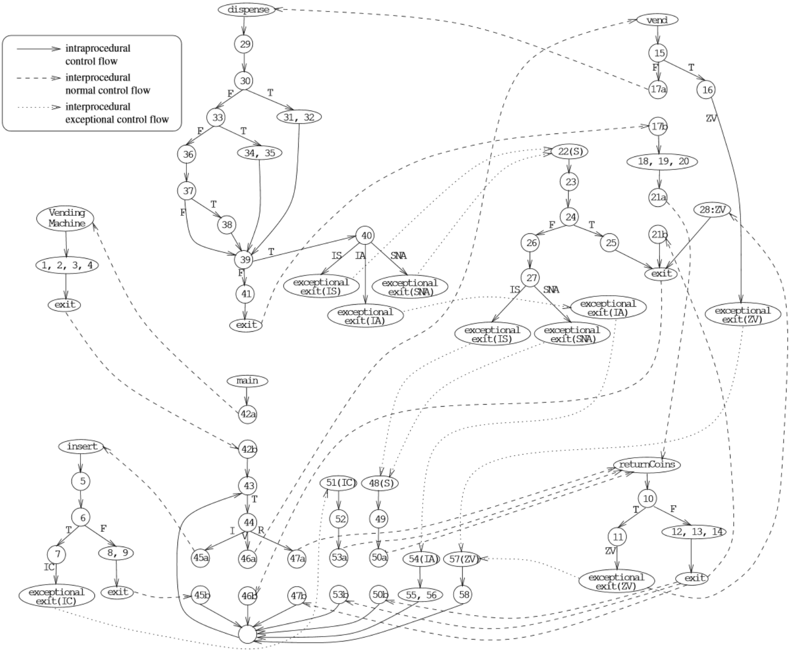


Figure 10: ICFG for the vending-machine program.

David F. Bacon and Peter F. Sweeney.
Fast static analysis of c++ virtual function calls.
SIGPLAN Not., 31(10):324–341, October 1996.

David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers.
Call graph construction in object-oriented languages.
In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 108–124, New York, NY, USA, 1997. ACM.

Mary W. Hall and Ken Kennedy.
Efficient call graph analysis.
ACM Lett. Program. Lang. Syst., 1(3):227–242, September 1992.

Jon Loeliger, Robert Metzger, Mark Seligman, and Sean Stroud.
Pointer target tracking—an empirical study.
In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 14–23, New York, NY, USA, 1991. ACM.

Saurabh Sinha and Mary Jean Harrold.
Analysis and testing of programs with exception handling constructs.
IEEE Trans. Softw. Eng., 26(9):849–871, September 2000.

Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin.
Practical virtual method call resolution for java.
SIGPLAN Not., 35(10):264–280, October 2000.