Lecture 4. Control Flow Analysis

Wei Le

2016.2

Agenda: Concepts and Algorithms

- CFG: Control flow graphs
- ICFG: Interprocedural control flow graphs
- Loops
- Call graph
- Exception flow
- More challenges

Assignment: Research Proposal

- 1 page proposal: problem definition, research questions, ideas and approaches, expected deliverables
- team up to 3 members
- version 1 due Feb 29, 11:59 pm
- version 2 due Mar 7, 11:59 pm

History of Control Flow Analysis

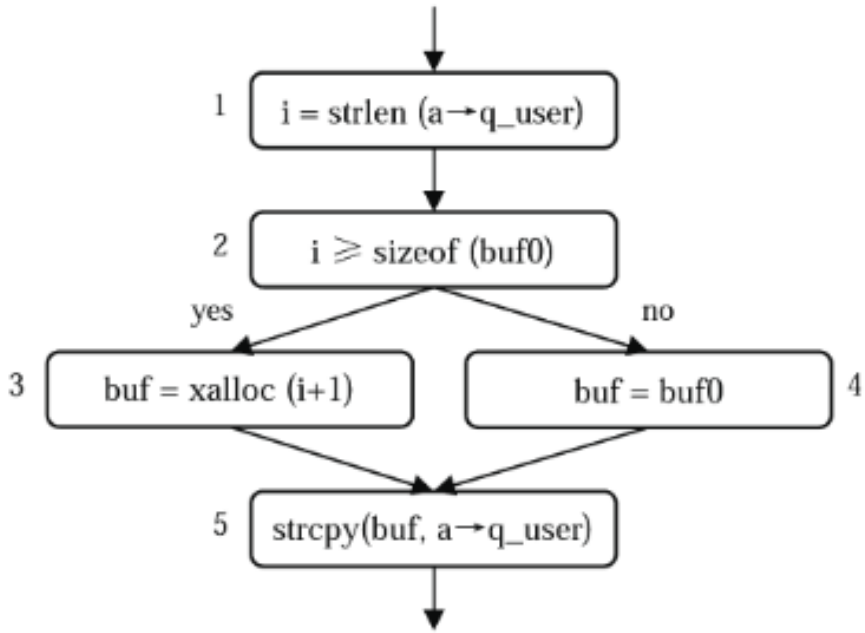
- 1970, Frances Allen, Her 1970 papers, "Control Flow Analysis" and "A Basis for Program Optimization" established "intervals" as the context for efficient and effective data flow analysis and optimization
- Turing award for pioneering contributions to the theory and practice of optimizing compiler techniques, awarded 2006

Control Flow Graphs

Control Flow Graphs

- Control flow analysis aims to determine the execution order of program statements or instructions
- **Basic block**: a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed).
- **Control flow graph (CFG)** is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths.
- a control flow graph specifies all possible execution paths

Control Flow Graphs: An Example



Program Paths and Traces

- **Path**: a sequence of node on the CFG (static), including an entry node and an exit node; **path segment**: a subsequence of nodes along the path
- **Trace**: a sequence of instructions performed during execution (dynamic)

ICFG: Interprocedural Control Flow Graphs

- CFG: representing control flow for a single procedure
- ICFG: representing control flow for a program

ICFG

Interprocedural Conditional Branch Elimination by Rastislav Bodk, Rajiv Gupta and Mary Lou Soffa

- a graph that combines CFGs of all program procedures by connecting procedure entries and exits with their call sites
- each procedure can have multiple procedure entry nodes and multiple procedure exit nodes
- the successors of a call site node are the procedure entry node, and the associated call site exit nodes.
- ICFG in a normal call site form, where 1) each call site node has a single procedure entry successor 2) each call site exit node has exactly one call site predecessor and one procedure exit predecessor (we assume that the above nodes are dummy nodes with no program statements)

ICFG

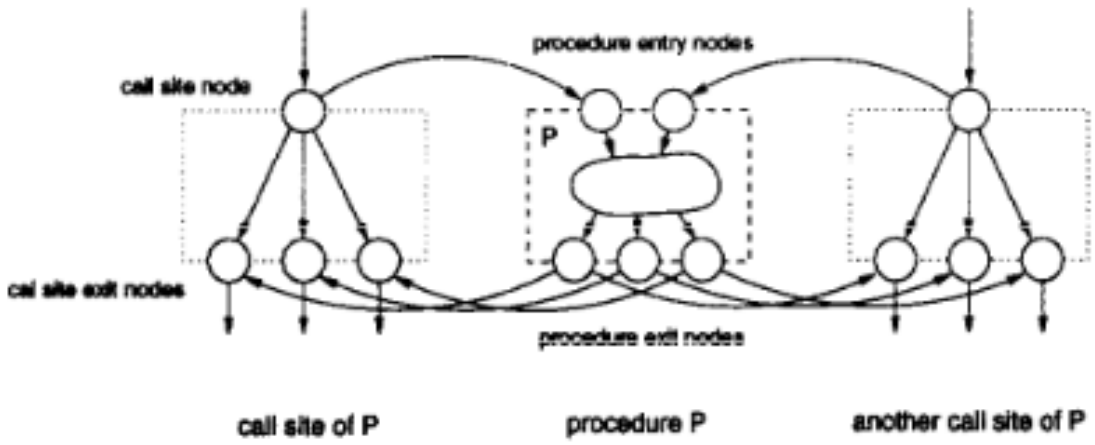


Figure 3: Interprocedural CFG in call site normal form.

ICFG

Demand-Driven Computation of Interprocedural Data Flow by Evelyn Duesterwald, Rajiv Gupta and Mary Lou Soffa

- A program consisting of a set of (recursive) procedures is represented as an interprocedural control flow graph $\{G_1, ..., G_k\}$ where $G_p = (N_p, E_p)$ is a directed graph representing procedure p
- Nodes in N_p represent the statements in p and edges in E_p represent the transfer of control among statements
- Two distinguished nodes r_p and e_p represent the unique entry and exit nodes of p
- The set $E = \bigcup\{E_i | 1 \leq i \leq k\}$ denotes the set of all edges in G, $N = \bigcup\{N_i | 1 \leq i \leq k\}$ denotes the set of all nodes in G
- $N_{call} \subset N$ denotes the set of nodes representing procedure calls (call sites) and for each node $n \in N_{call}$, $call(n)$ denotes the procedure called from n

ICFG

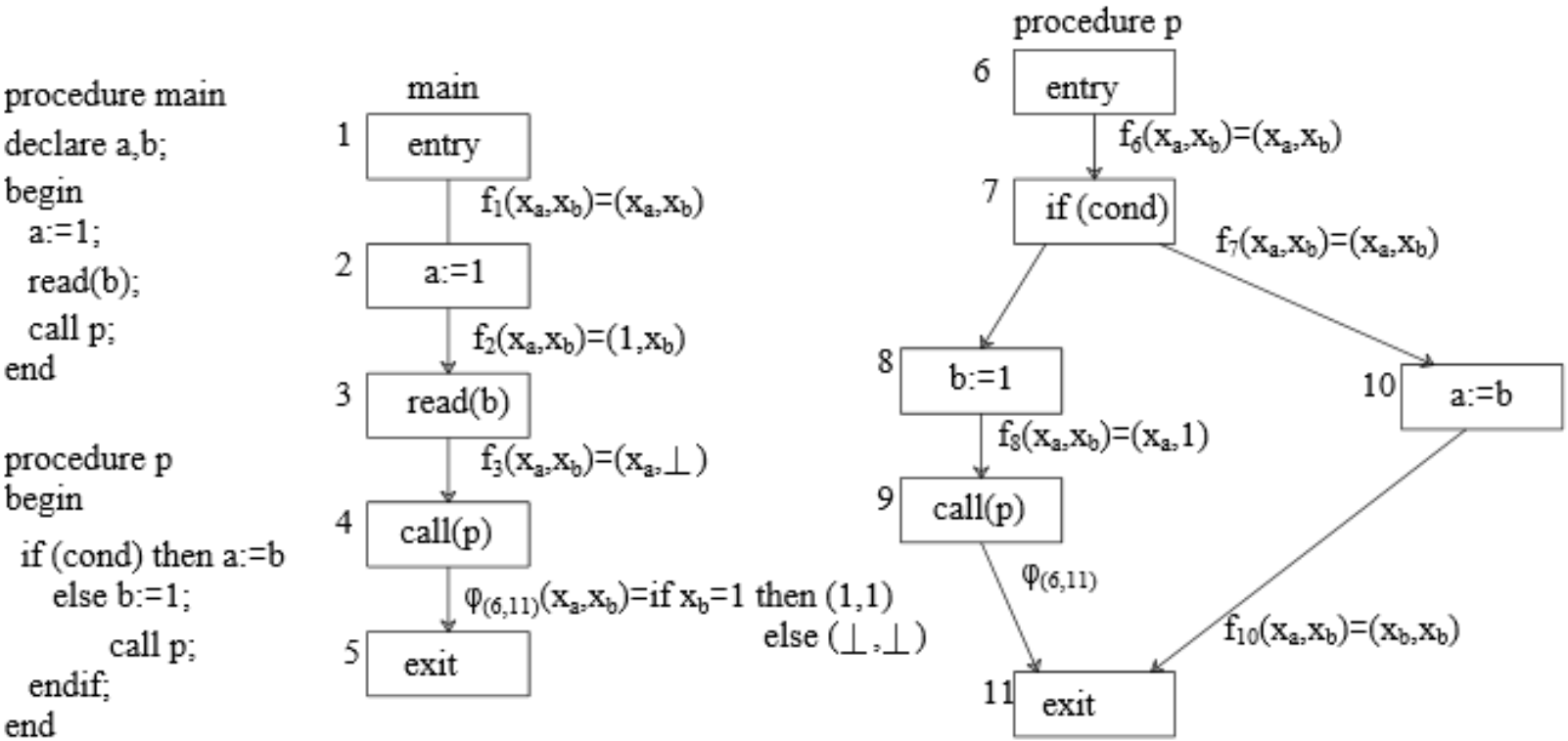


Figure 1: An IFG with the local flow functions for copy constant propagation.

Loops

Loops

Most of the execution time is spent in loops - the 90/10 law, which states that 90% of the time is spent in 10% of the code, and only 10% of the time in the remaining 90% of the code. Loop is a graph theoretical term:

- **dominance**, **dominator** and **postdominators** – relations between nodes on the CFG
- **head**, **back edge**, **reducibility**
- **natural loops**
- **single loops**, **nested loops** and **inner loop**

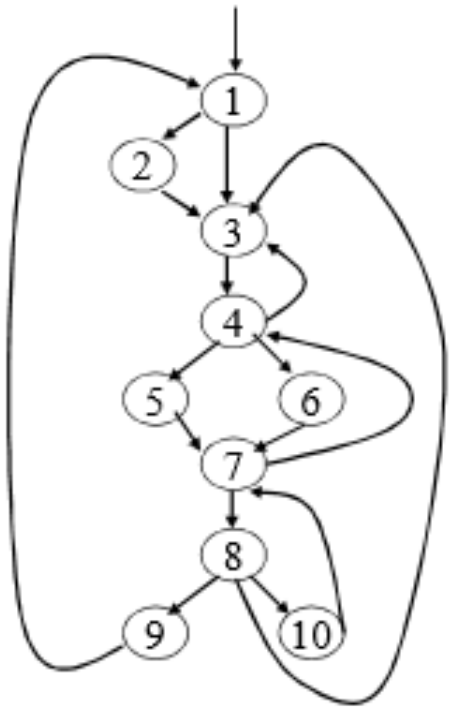
Dominance

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 (except the entry node) 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

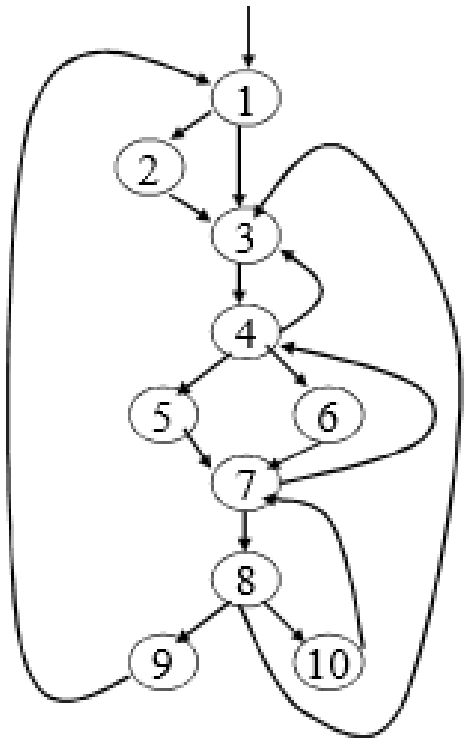


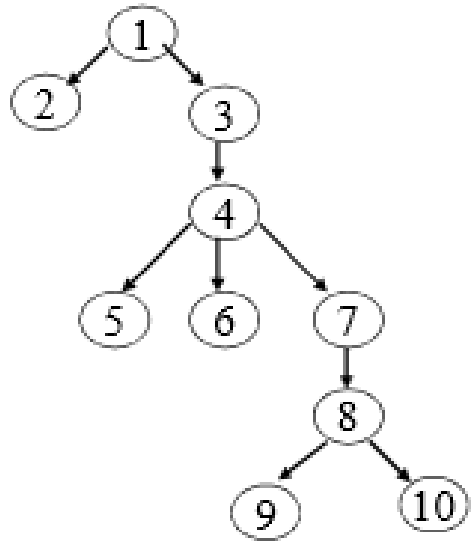
Block	Dom	IDom
1	{1}	—
2	{1,2}	1
3	{1,3}	1
4	{1,3,4}	3
5	{1,3,4,5}	4
6	{1,3,4,6}	4
7	{1,3,4,7}	4
8	{1,3,4,7,8}	7
9	{1,3,4,7,8,9}	8
10	{1,3,4,7,8,10}	8

Properties of Dominators

- Reflexive: a dom a
- Transitive: if a dom b and b dom c then a dom c
- Antisymmetric: if a dom b and b dom a then b=a

Dominator Tree



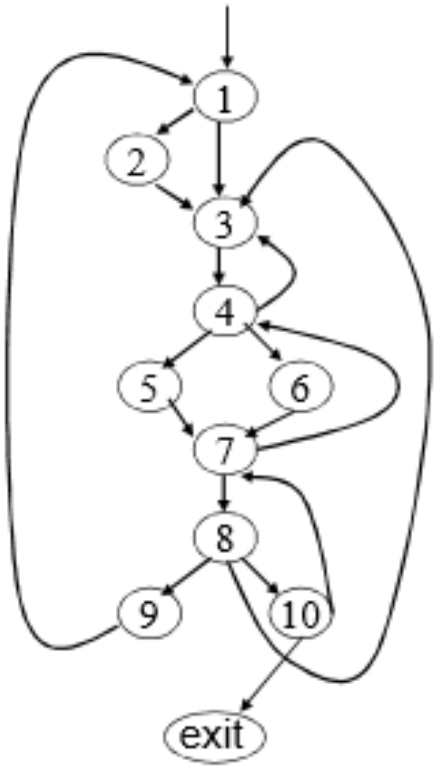


- In a dominator tree, a node's parent is its immediate dominator

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 (except the exit node) has a unique **immediate post dominator**

Post-Dominator Example

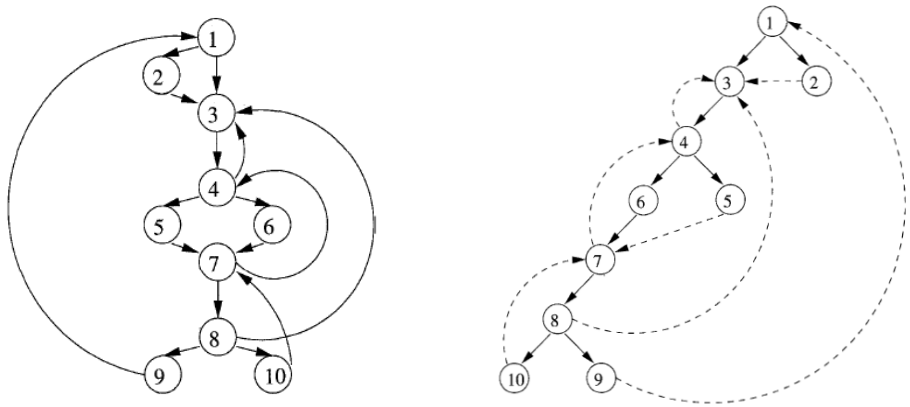


Block	Pdom	IPdom
1	{3,4,7,8,10,exit}	3
2	{2,3,4,7,8,10,exit}	3
3	{3,4,7,8,10,exit}	4
4	{4,7,8,10,exit}	7
5	{5,7,8,10,exit}	7
6	{6,7,8,10,exit}	7
7	{7,8,10,exit}	8
8	{8,10,exit}	10
9	{1,3,4,7,8,10,exit}	1
10	{10,exit}	exit

Retreating Edge: an Example

- Depth-First Ordering
  - A **depth-first search** of a graph visits all the nodes in the graph once, by starting at the entry node and visiting the nodes as far away from the entry node as quickly as possible.
  - A **preorder traversal** visits a node before visiting any of its children, which it then visits recursively in left-to-right order.
  - A **postorder traversal** visits a node's children, recursively in left-to-right order, before visiting the node itself.
  - A **depth-first ordering** is the reverse of a postorder traversal. That is, in a depthfirst ordering, we visit a node, then traverse its rightmost child, the child to its left, and so on.
- Retreating edge: in a depth first traversal of a graph, the retreating edge $m \rightarrow n$ connects a decedent m to its ancestor n ($dfn[m] \geq dfn[n]$) [dragon book p.662]

Retreating Edge: an Example



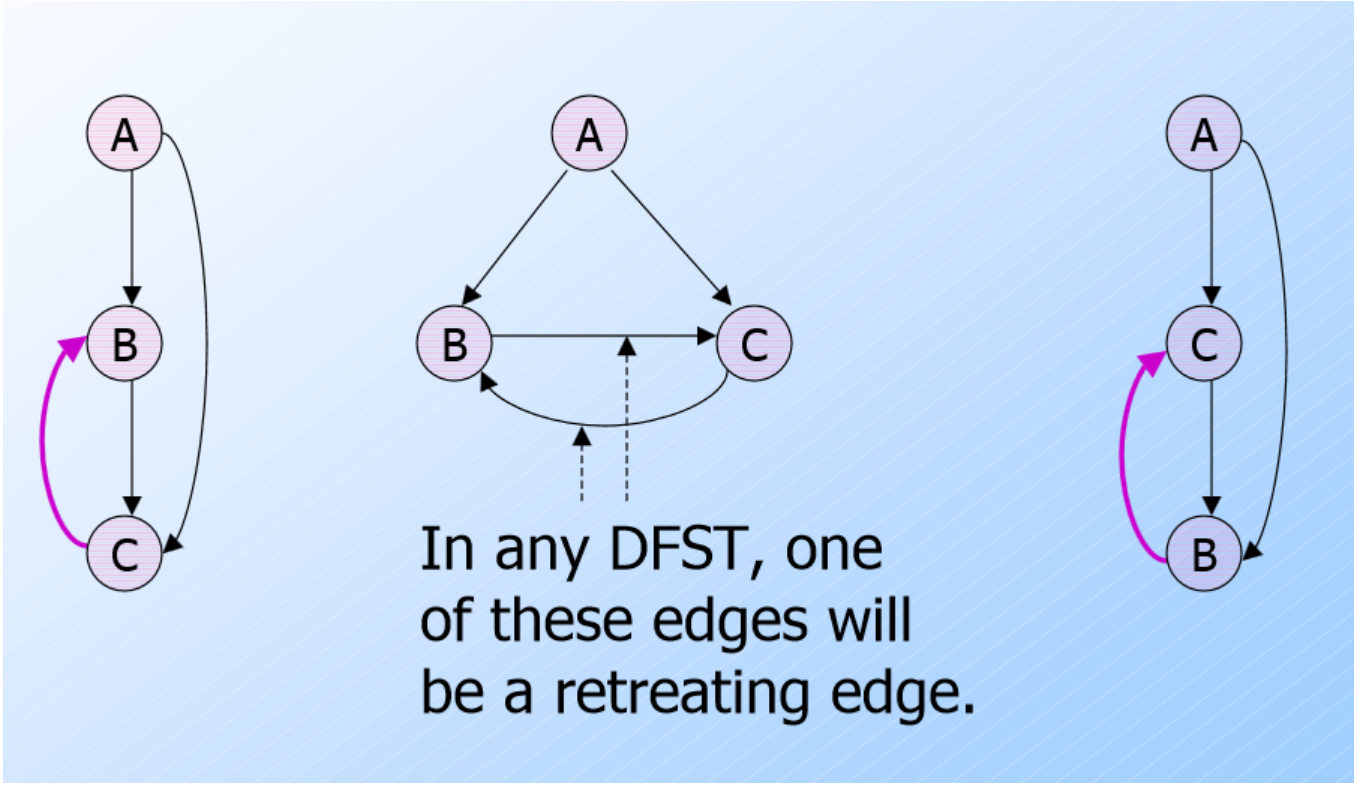
A depth-first traversal of the tree is given by:
$1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10$, then back to 8, then to 9. We go back to 8 once more, retreating to 7, 6, and 4, and then forward to 5. We retreat from 5 back to 4, then back to 3 and 1 . From 1 we go to 2, then retreat from 2, back to 1 , and we have traversed the entire tree.

- The preorder sequence for the traversal is thus 1 , 3, 4, 6 , 7, 8, 10, 9 , 5 , 2.
- The postorder sequence for the traversal of the tree is 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.
- The depth-first ordering, which i s the reverse of the postorder sequence, is 1 , 2, 3, 4, 5, 6, 7, 8, 9, 10.

Back Edge and Reducibility

- **Back edge**: head (ancestor) dominates its tail (decedent)
- A flow graph is **reducible** if every retreating edge in a flow graph is a back edge: take any DFST for the flow graph, remove the back edges, the result should be acyclic

Irreducible Graph



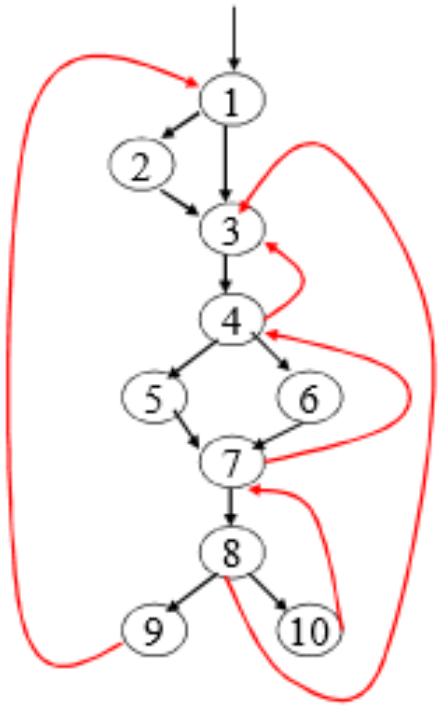
Reducibility in Practice

- If you use only while-loops, for-loops, repeat-loops, if-then(-else), break, and continue, then your flow graph is reducible.
- Some languages only permit procedures with reducible flowgraphs (e.g., Java)
- GOTO Considered Harmful: can introduce irreducibility
  - FORTRAN
  - C
  - C++

Natural Loops (Reducible Loops)

- Flowgraph is reducible iff all loops in it **natural**
- Single entry node (d)
  - no jumps into middle of loop
  - d dominates all nodes in loop
- Requires back edge into loop header ($n \rightarrow d$)
  - n is tail, d is head
  - single entry point
- Natural loop of back edge ($n \rightarrow d$):
  - d+ {all nodes that can reach n without touching d }

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

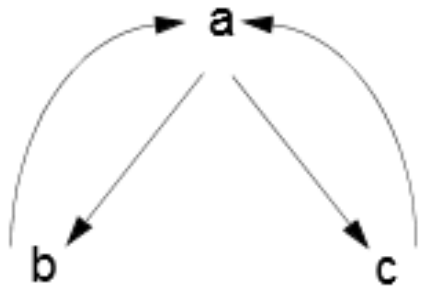
If two loops do not have the same header:

- disjoint
- one is entirely contained (nested within) the other

An **inner loop** is a loop that contains no other loops

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

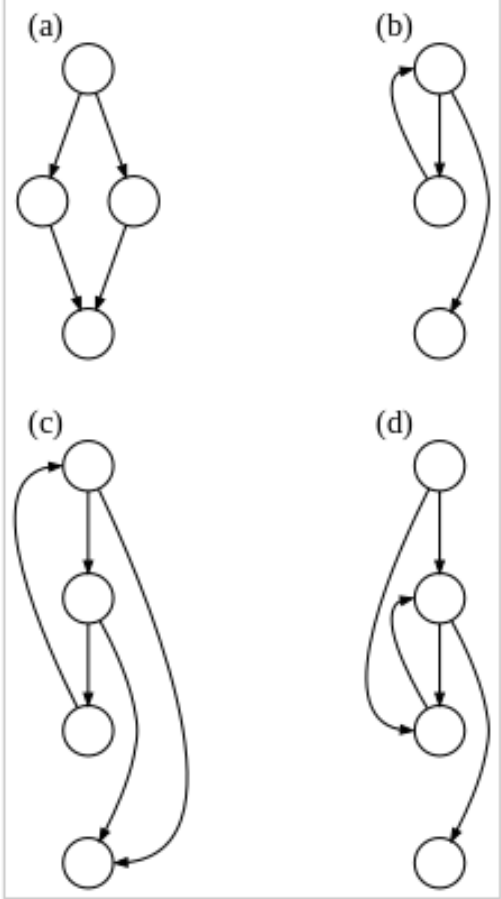
If two loops share the same header: hard to tell which is the inner loop



Algorithm to Find Natural Loops

- Find the dominator relations in a flow graph
- Identify the back edges
  - perform a depth first traversal
  - for each retreating edge $t \rightarrow h$, check if h is in t's dominator's list
- Find the natural loop associated with the back edge
  - delete h from the flow graph
  - find all the nodes that can reach t (those nodes plus h from the natural loop of $t \rightarrow h$), e.g., using a dept-first backward traversal

Summary: Examples of CFGs



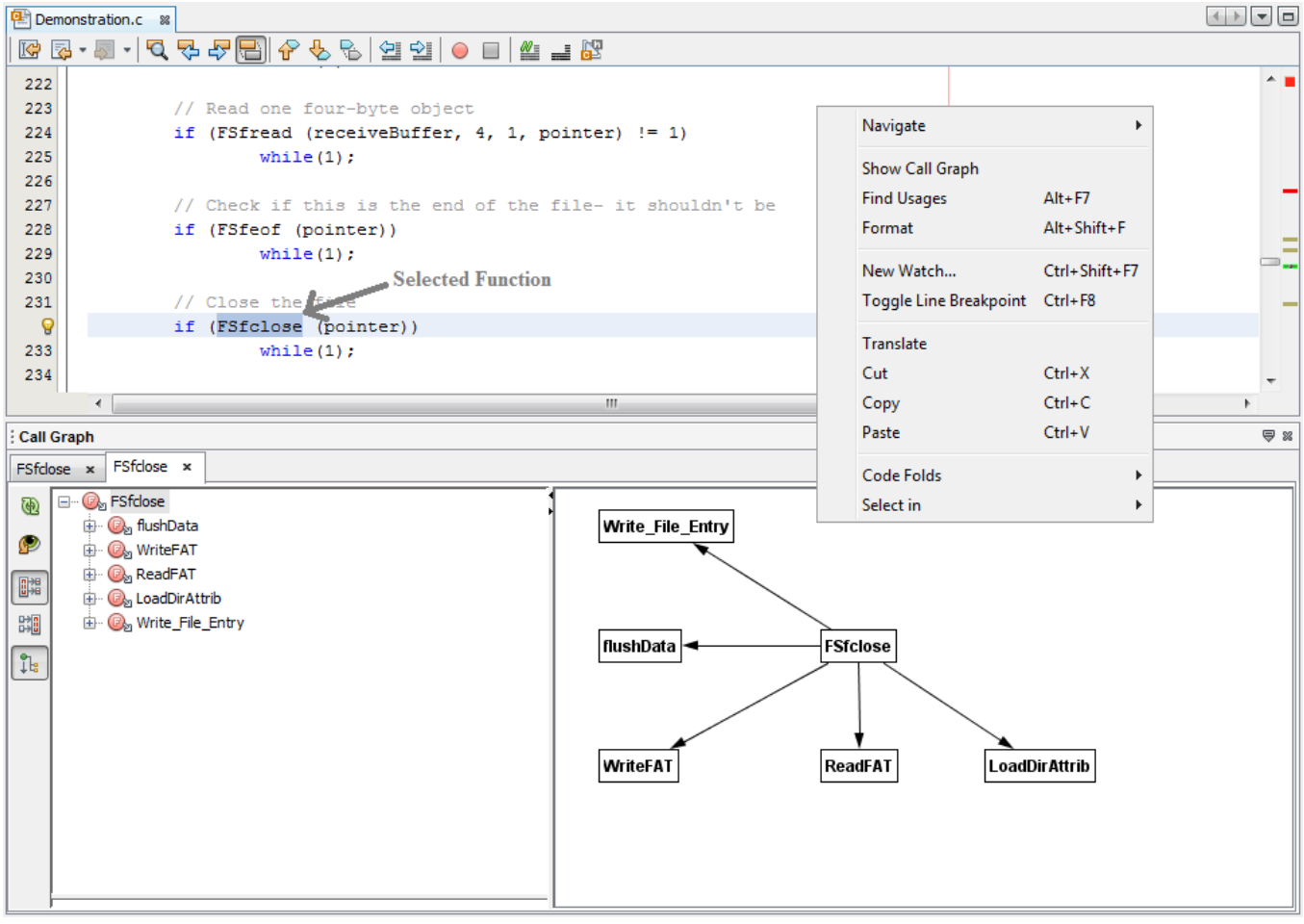
Some CFG examples:
(a) an if-then-else
(b) a while loop
(c) a natural loop with two exits, e.g. while with an if...break in the middle; non-structured but reducible
(d) an irreducible CFG: a loop with two entry points, e.g. goto into a while or for loop

Call Graph

- **Call graph** is a directed graph that represents the calling relationships between program procedures

Call Graphs

Call Graph: Example



Challenge: Dynamic Dispatch

- Which implementation of the function will be invoked at the callsite?
- The binding is determined at runtime, based on the input of the program and execution paths.

- Function pointers
- Object oriented languages
- Functional languages

Dynamic Dispatch: Example

```
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?

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

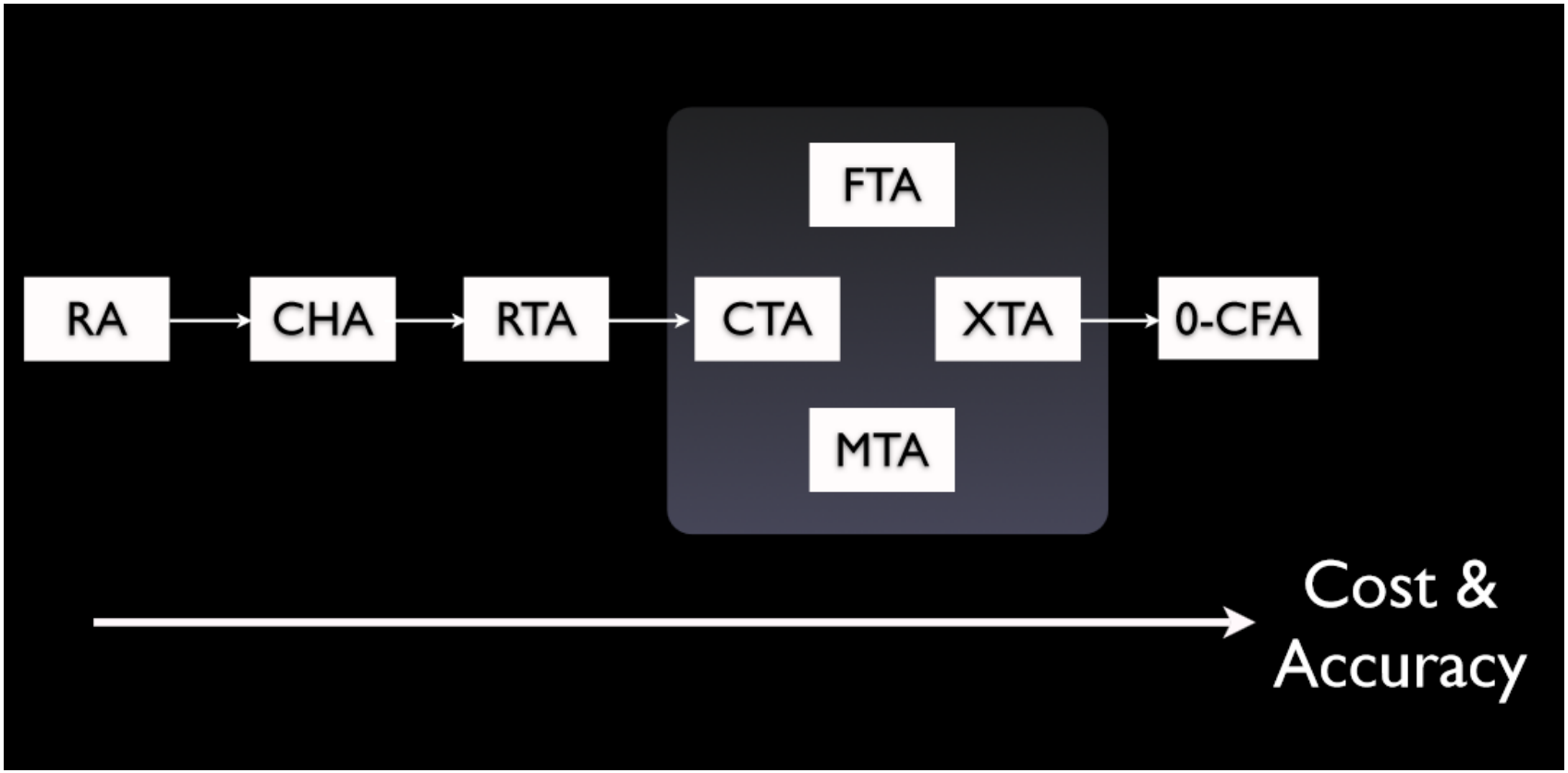
```c
#include <math.h>
#include <stdio.h>

// 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 for Function Pointers

- Determining dataflow or values of variables
- Call-target resolution depend on the flow of values
- data-flow depends on control-flow, yet control-flow depends on data-flow

Call Graph Construction for Object-Oriented Programs: Algorithms

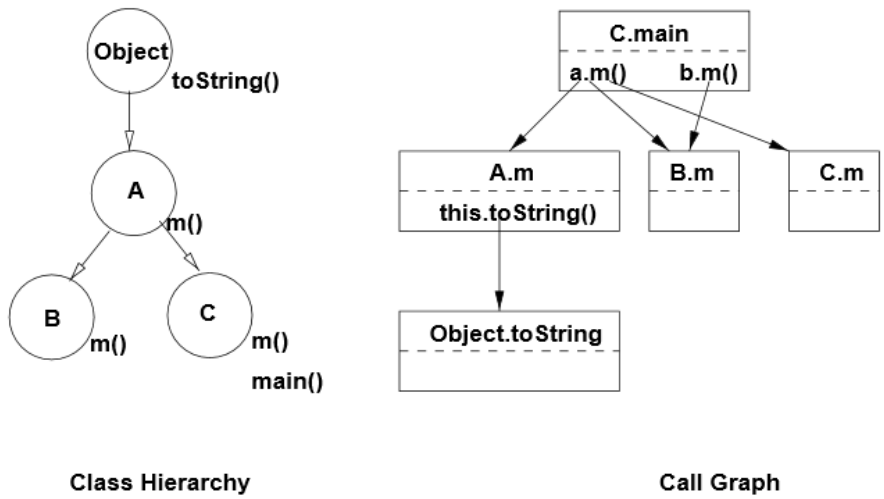
Scalable Propagation-Based Call Graph Construction Algorithms by Frank Tip and Jens Palsberg



Class Hierarchy Analysis: CHA

```
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

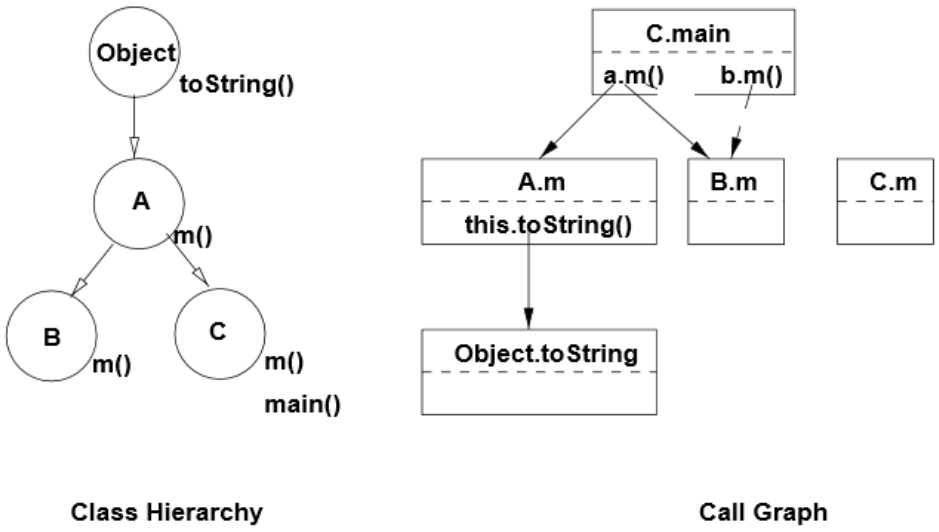


(b) Class Hierarchy and Call Graph

Rapid Type Analysis: RTA

```
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

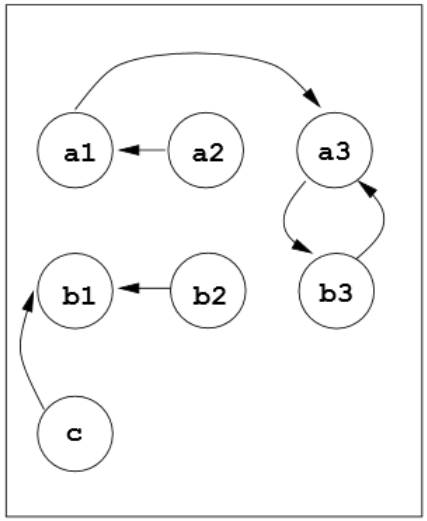


(b) Class Hierarchy and Call Graph

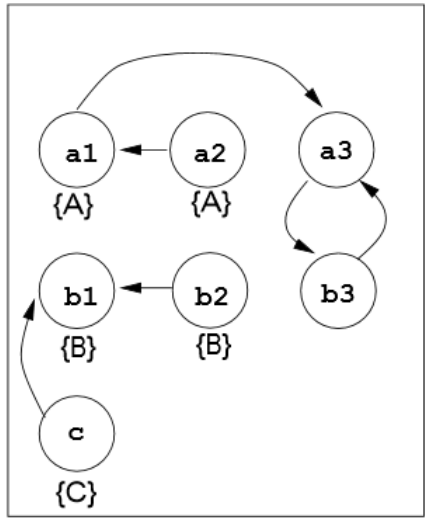
Variable Type Analysis: VTA

```
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;
```

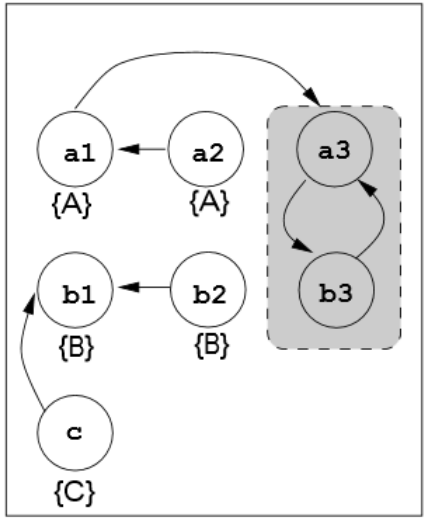
(a) Program



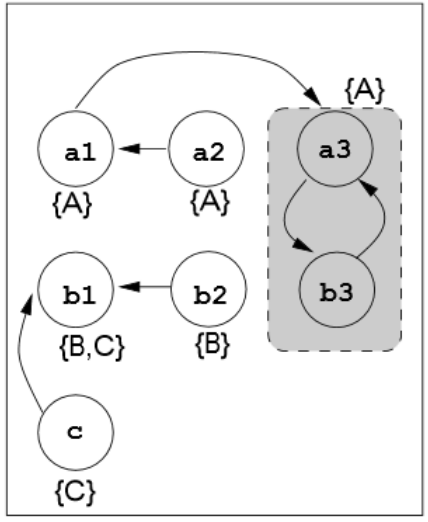
(b) Nodes and Edges



(c) Initial Types



(d) Strongly-connected components



(e) final solution

Call Graph Construction for Object-Oriented Programs

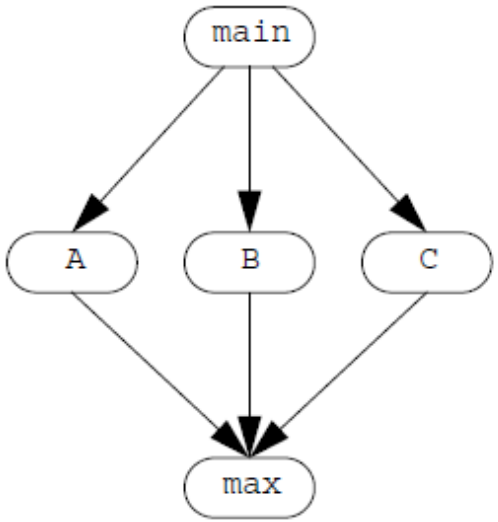
Between 1990-2000:

- **Class hierarchy analysis** (newly defined types) and **rapid type analysis** (RTA) (analyzing instantiation of the object) – resolve 71% virtual function calls [1996:Bacon]
- Theoretical framework for call graph constructions for object-oriented programs [1997:Grove]
- Pointer target tracking [1991:Loeliger]
- Callgraph analysis [1992:Hall]
- Variable type and declared type analysis [2000:Sundaresan]
- Scaling Java Points-To Analysis using SPARK [2003:Lhotak]

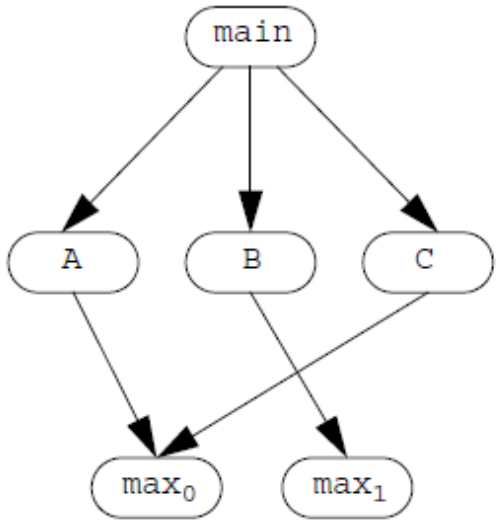
Context-Sensitivity for Building Call Graphs

```
procedure main() {
  return A() + B() + C();
}

procedure A() {
  return max(4, 7);
}
procedure B() {
  return max(4.5, 2.5);
}
procedure C() {
  return max(3, 1);
}
```

(a) Example Program



(b) Context-Insensitive



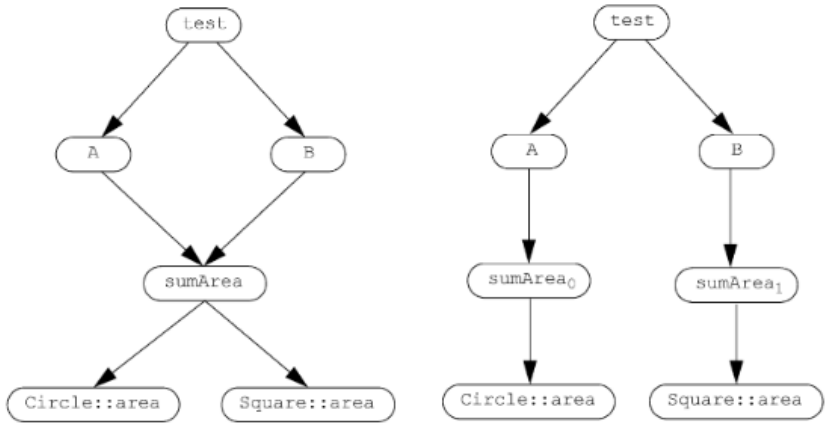
(c) Context-Sensitive

Context-Sensitivity for Building Call Graphs

```
class Shape {
    abstract float area();
}

class Square extends Shape {
    float size;
    Square(float s) {
        size = s;
    }
    float area() {
        return size * size;
    }
}

class Circle extends Shape {
    float radius;
    Circle(float r) {
        radius = r;
    }
    float area() {
        return PI*radius*radius;
    }
}

class SPair {
    Shape first;
    Shape second;
    SPair(Shape s1, Shape s2) {
        first = s1; second = s2;
    }
}
```

```
class Example {
    float test(float v1, float v2) {
        return A(v1, v2) + B(v1, v2);
    }

    float A(float v1, float v2) {
        Circle c1 = new Circle(v1);
        Circle c2 = new Circle(v2);
        return sumArea(new SPair(c1, c2));
    }

    float B(float v1, float v2) {
        Square s1 = new Square(v1);
        Square s2 = new Square(v2);
        return sumArea(new SPair(s1, s2));
    }

    float sumArea(SPair p) {
        return p.first.area() + p.second.area();
    }
}
```

(a) Example program fragment



(b) Context-Insensitive (c) Context-Sensitive

Fig. 1. Context-insensitive vs. context-sensitive call graph.

k-CFA [1988:Shivers]

- 0-CFA: context-insensitive
- **k-CFA**: k number of calls are considered
- k-CFA for functional language: EXPTIME-complete (non-polynomial)
- k-CFA for OO programs: polynomial

Type Inference, Alias Analysis, Call Graph Construction

- Call graph construction needs to know the type of the object receivers for the virtual functions
- Object receivers may alias to a set of reference variables so we need to perform alias analysis
- Determine types of the set of relevant variables: type inferences – infer types of program variables

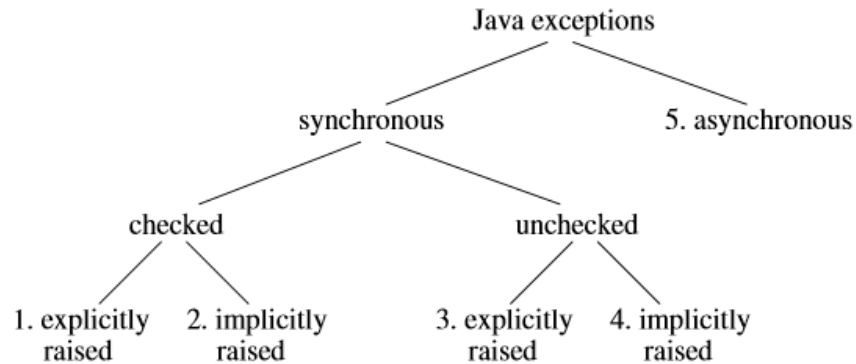
Exception Handling

Exception Handling: C++

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

Exception 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
   . . .
}
```



Exception Handling: Java

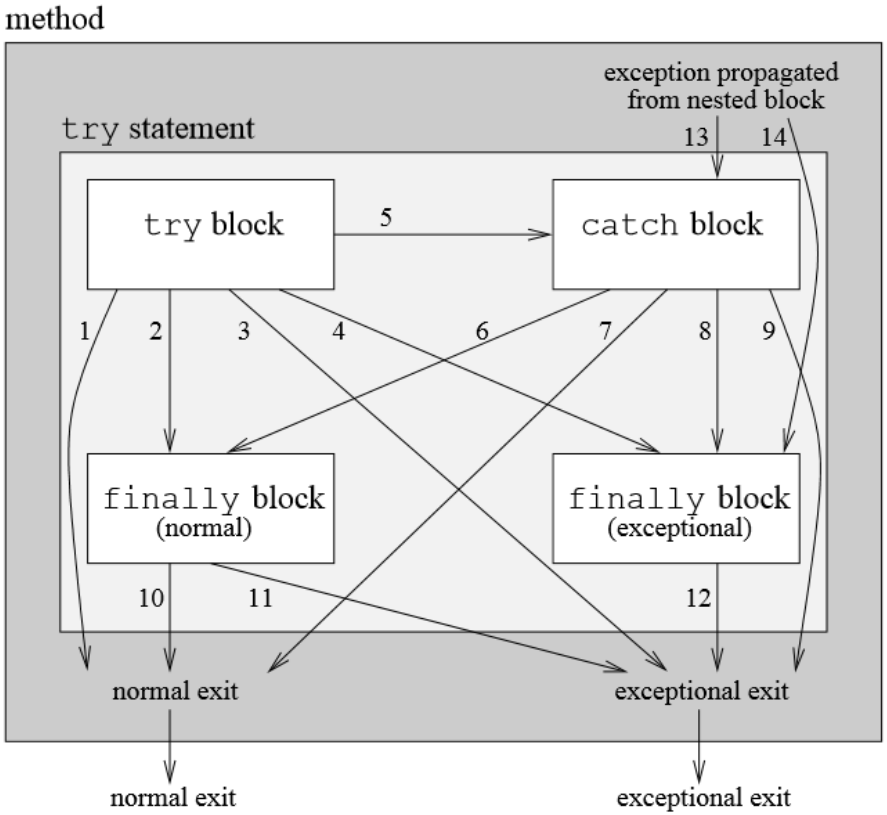
```
public void openFile(){
    try {
        // constructor may throw FileNotFoundException
        FileReader reader = new FileReader("someFile");
        int i=0;
        while(i != -1){
            //reader.read() may throw IOException
            i = reader.read();
            System.out.println((char) 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 [Sinha:2000]

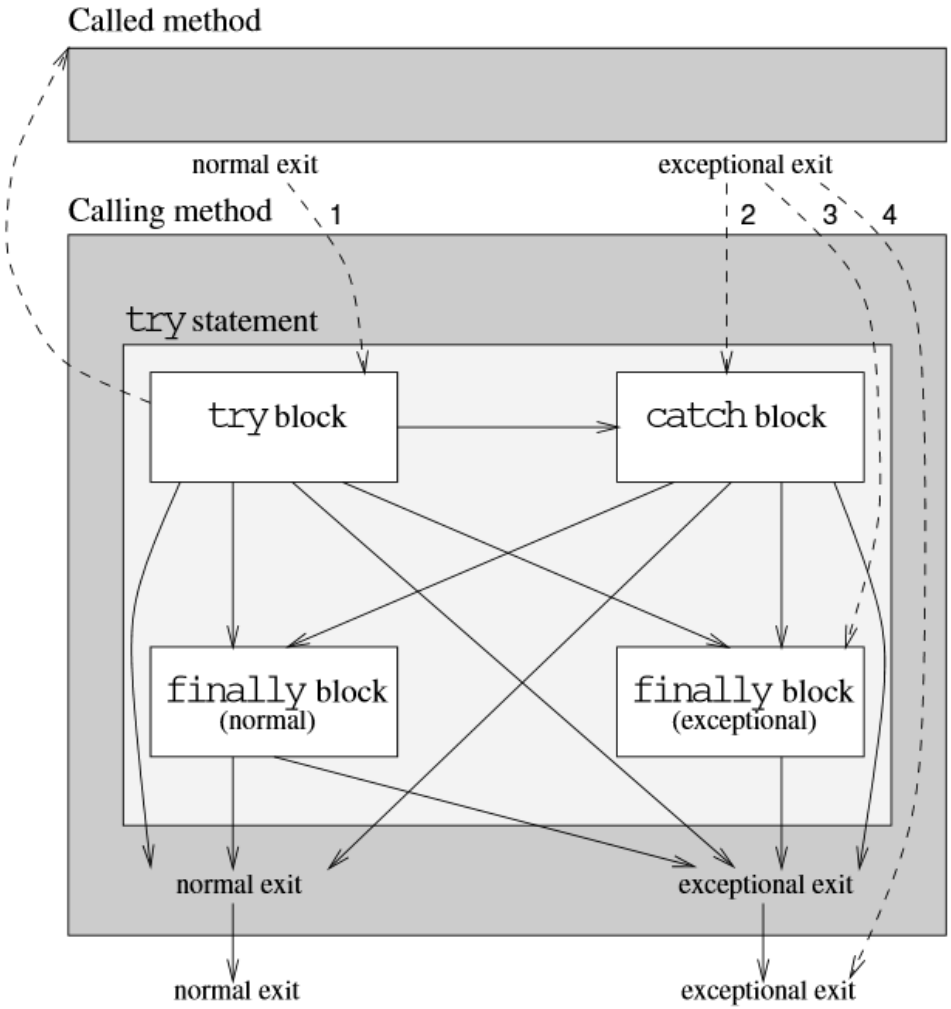
Subject		Number of classes	Number of methods	Methods with EH constructs
Name	Description			
antlr	Framework for compiler construction	175	1663	175 (10.5%)
debug	Sun's Java debugger	45	416	80 (19.2%)
jaba	Architecture for analysis of Java bytecode	312	1615	200 (12.4%)
jar	Sun's Java archive tool	8	89	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%)

Modeling Exception Handling Constructs in ICFGs [Sinha:2000]



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[Sinha:2000]



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" );
      }
    }
```

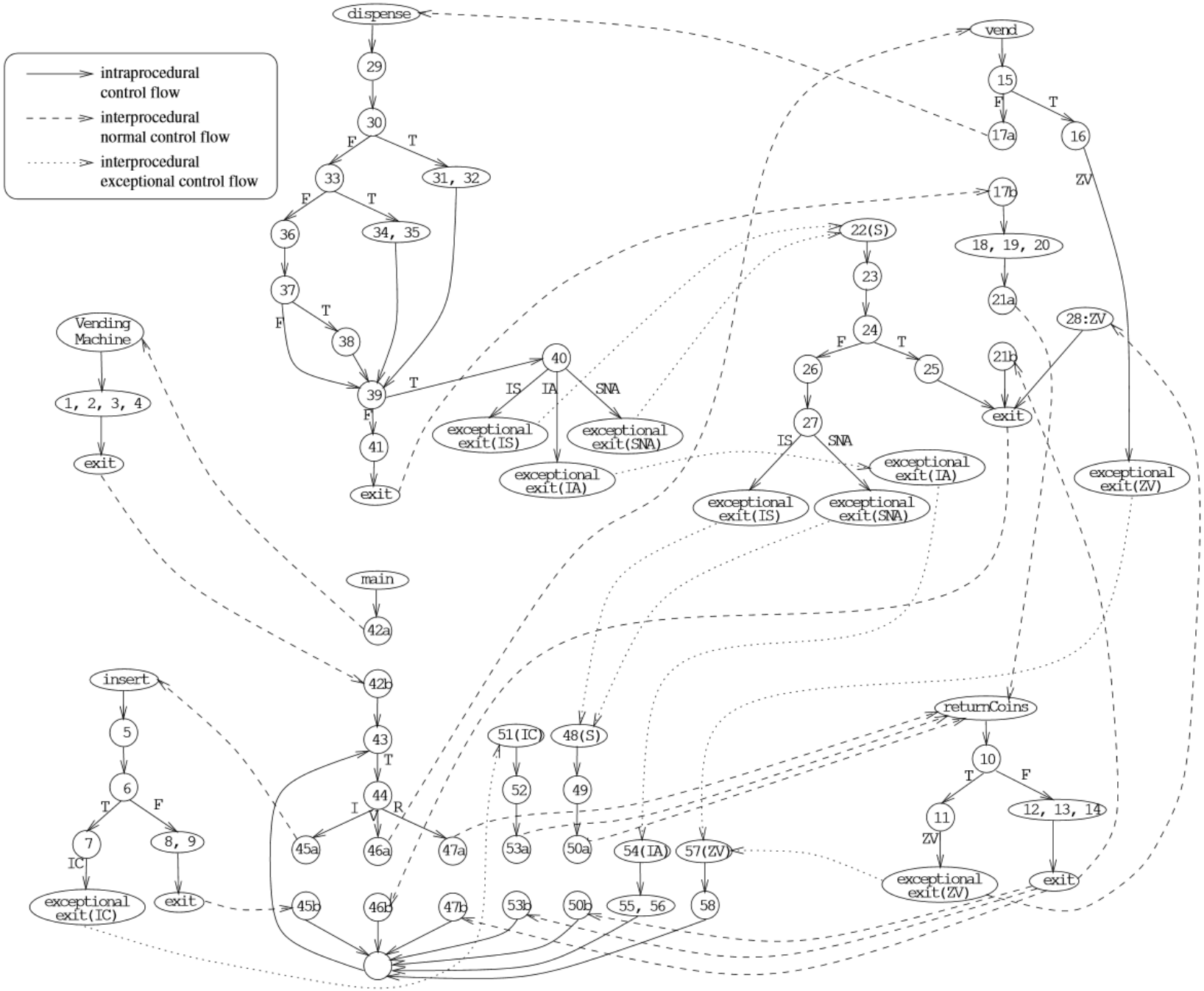


Figure 10: ICFG for the vending-machine program.

Other Challenges

- Event driven code
- Libraries and callbacks

Analyzing Android app's Control Flow: Example