Dataflow Analysis in Practice: Program Analysis Frameworks, Analysis Scope and Approximation

Announcements

- HW1 due today
- HW2 posted
  - Your task is to set this up locally as soon as possible

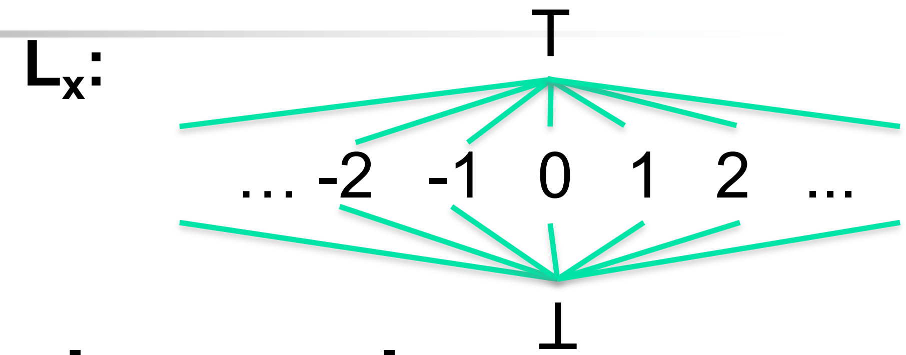
So Far and Moving On…

- Dataflow analysis
  - Four classical dataflow problems
  - Dataflow frameworks
    - CFGs, lattices, transfer functions and properties, worklist algorithm, MFP vs. MOP solutions
  - Non-distributive analysis
    - Constant propagation
    - Points-to analysis (will cover in catchup week!)
- Program analysis in practice

Outline of Today's Class

- Constant propagation (catchup)
- Program analysis in practice
  - Program analysis frameworks
    - Soot program analysis framework
    - Ghidra framework
  - Analysis scope and approximation
- Class analysis

Constant Propagation fits into Monotone Dataflow Framework

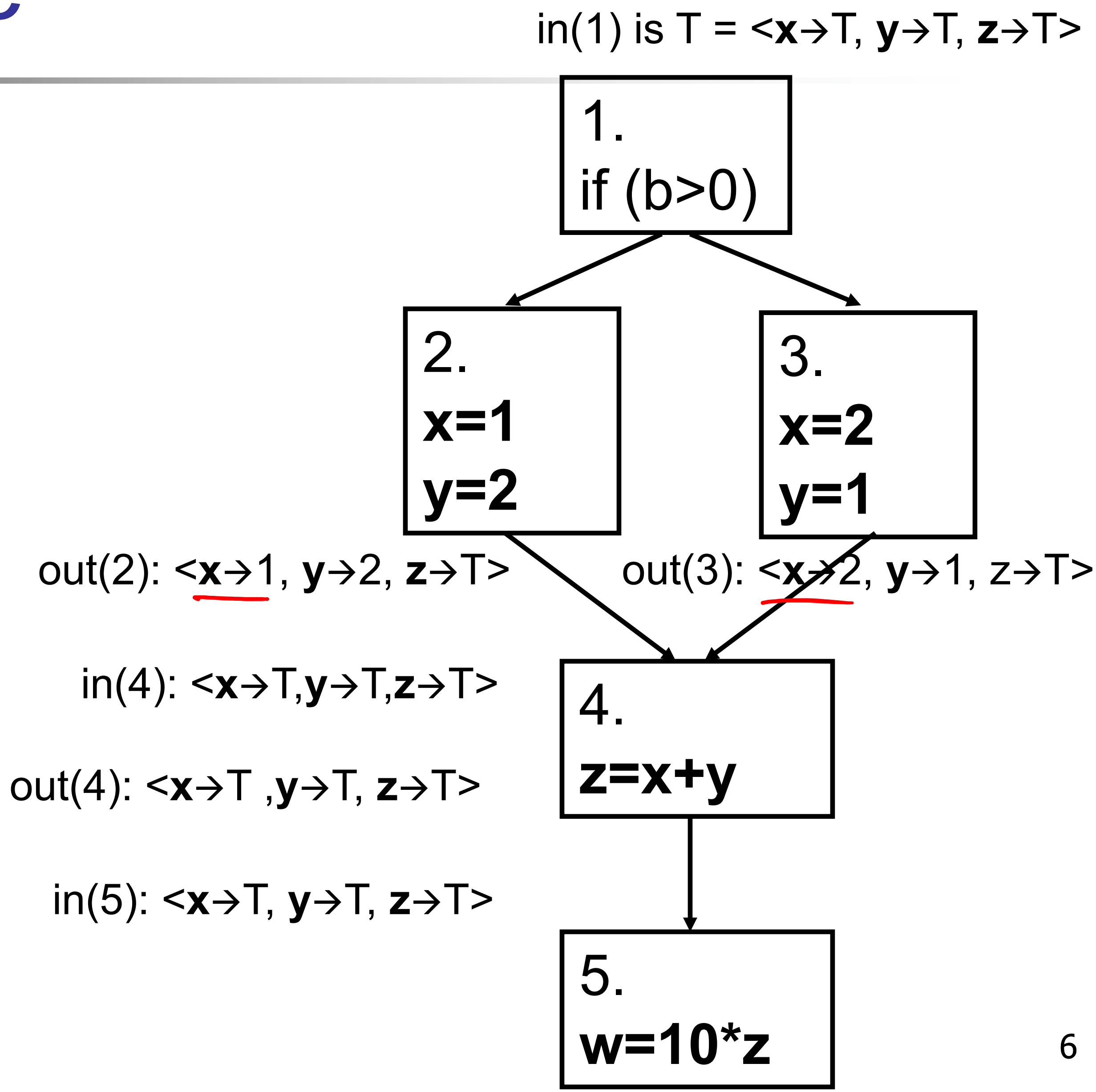


- Property space
  - Product lattice **$L = L_x \times L_y \times \ldots \times L_z$**
  - **L satisfies the ACC**

and

- Function space **F: L→ L** is monotone
- Thus, analysis fits into the monotone dataflow framework and can be solved using the worklist algorithm

Example


in(1) is T = <x→T, y→T, z→T>
1.
if (b>0)
2.
x=1
y=2
3.
x=2
y=1
out(2): <x→1, y→2, z→T>
out(3): <x→2, y→1, z→T>
in(4): <x→T,y→T,z→T>
out(4): <x→T ,y→T, z→T>
4.
z=x+y
in(5): <x→T, y→T, z→T>
5.
w=10*z

Constant Propagation is Monotone but Not Distributive!

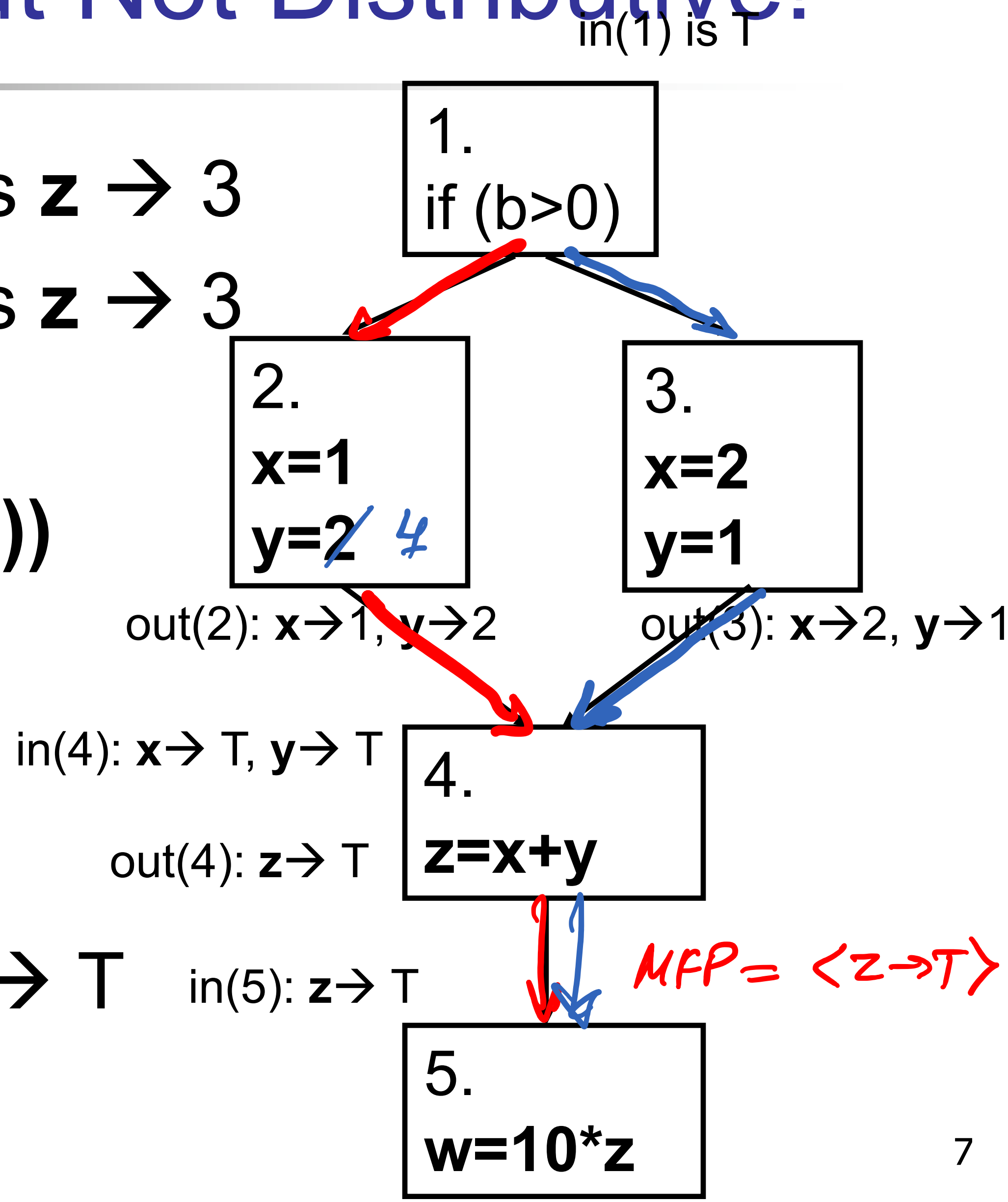
- $f_4(f_2(f_1(T)))$ computes **z** → 3
- $f_4(f_3(f_1(T)))$ computes **z** → 3
- Thus, MOP at 5

$f_4(f_2(f_1(T)))$ V $f_4(f_3(f_1(T)))$

computes **z** → 3

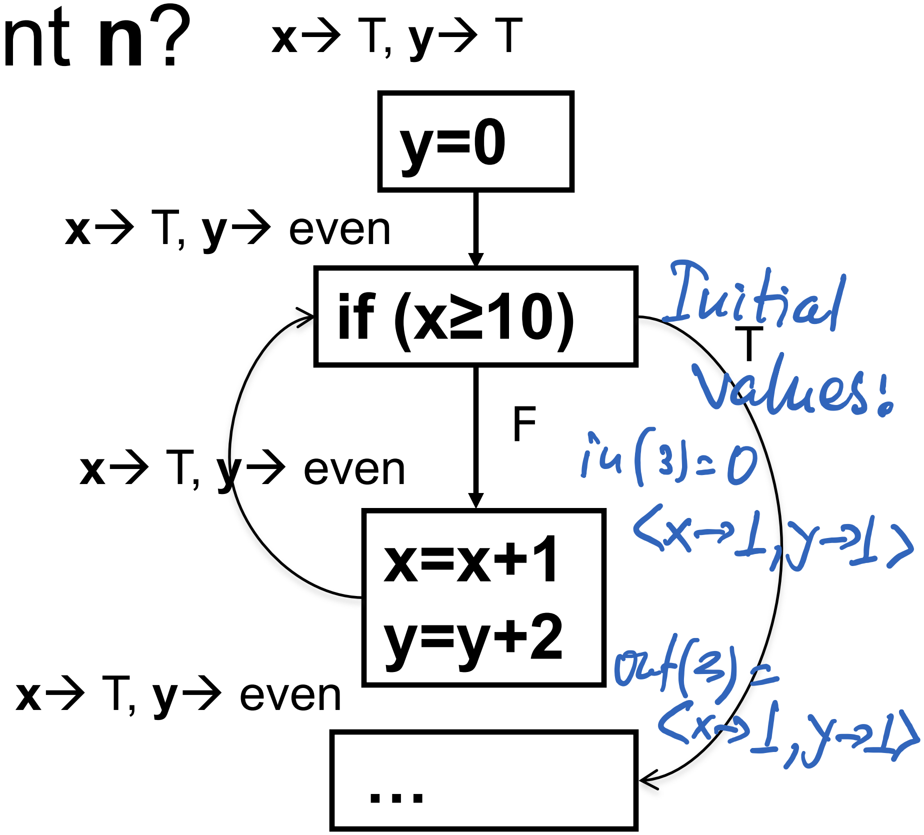
MFP at 5 computes **z** → T

(i.e., **z** is NOT a const)

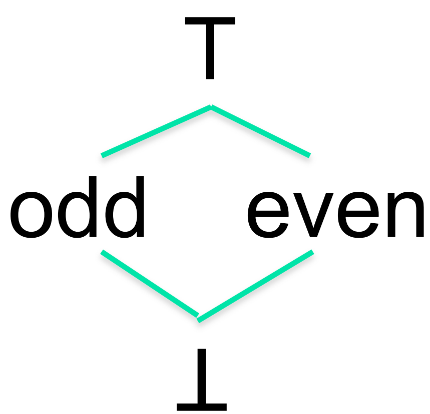


More Product Lattices

- Problem statement: Is integer variable **x** odd or even at program point **n**?

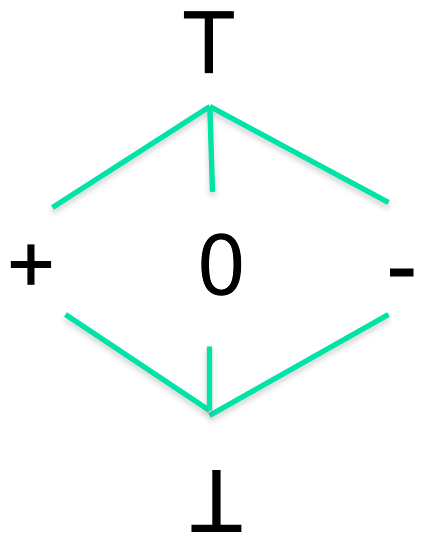


- L_x:



More Product Lattices

- Problem statement: What sign does a variable hold at a given program point, i.e., is it positive, negative, or 0
- $\mathbf{L_x}$:



E.g., **< x→+,y→**T**,z→0 >**

So far and moving on

- **Intra**procedural dataflow analysis
  - CFGs, lattices, transfer functions, worklist algorithm, etc.
  - Classical analyses

- **Inter**procedural analysis
- Analysis scope and approximation

Program Analysis in Practice

- Program analysis frameworks
  - LLVM — C, C++, ... → LLVM-IR
  - Ghidra — x86, ARM → PCode → C ?
  - Soot — Java, Java bytecode, DALVIK → Jimple
  - WALA, other

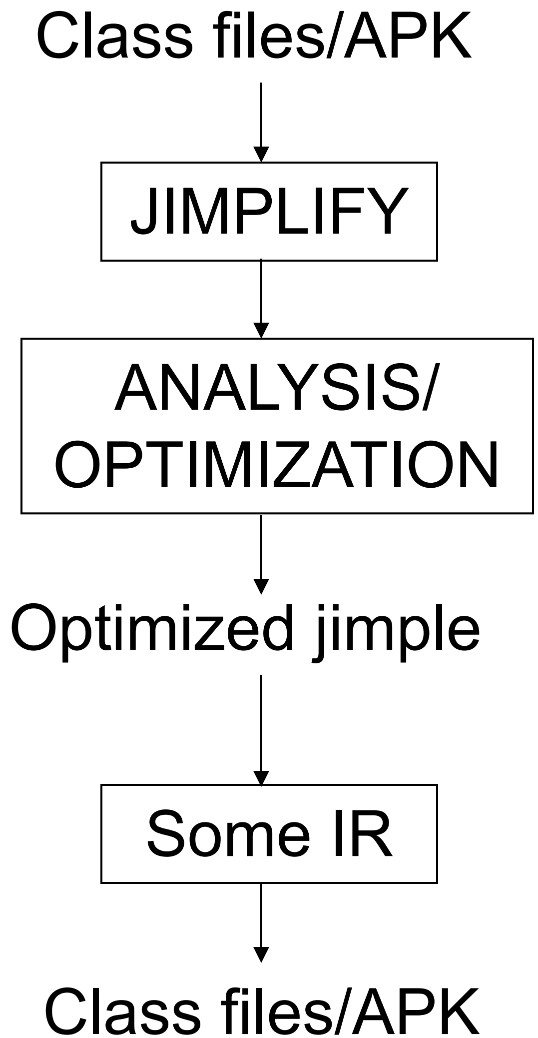
Soot: a framework for analysis and optimization of Java/Dalvik bytecode

- https://soot-oss.github.io/soot/
- History
- Overview of Soot
  - From Java bytecode/Dalvik bytecode to **typed** 3-address code (**Jimple**)
  - 3-address code analysis and optimization
  - From Jimple to Java/Dalvik
- Jimple
- Analysis

History

- https://soot-oss.github.io/soot/
- Started by Prof. Laurie Hendren at McGill
  - First paper on Soot came in 1999
  - Patrick Lam
  - Ondřej Lhoták
  - Eric Bodden
  - and other…
- Now developed by Eric Bodden and his group: https://github.com/soot-oss/soot

Overview of Soot



Advantages of Jimple and Soot

- Jimple
  - Typed local variables
  - 16 simple 3-address statements (1 operator per statement). Bridges gap from analysis abstraction to analysis implementation
- Soot provides
  - Itraprocedural dataflow analysis framework
  - Points-to analysis for Java
  - IR from Dalvik and taint analysis
  - Other analyses and optimizations

Jimple

- Run soot: java soot.Main –jimple A (need paths)

```
public class A {
    main(String[] args) {
        A a = new A();
        a.m();
    }
    public void m() {
    }
}
```

```
public class A extends java.lang.Object
  {
   public void <init>() {
     A r0;
     r0 := @this: A;
     specialinvoke r0.
          <java.lang.Object: void <init>()>();
     return;
   }
   …
```

(continues on next slide…)

Jimple

Java:

```
public class A {
    main(String[] args) {
        A a = new A();
        a.m();
    }
    public void m() {
    }
}
```

Jimple:

```
…
public void m() {
        A r0;
        r0 := @this: A;
        return;
    }
…
```

reference variables

Implicit parameter receiver

Jimple

Java:

```
public class A {
  main(String[] args) {
     A a = new A();
     a.m();
  }
  public void m() {
  }
}
```

Jimple:

```
...
main(java.lang.String[]) {
   java.lang.String[] r0;
   A $r1, r2;
   r0 := @parameter0: java.lang.String[];
   $r1 = new A;
   specialinvoke $r1.<A: void <init>()>();
   r2 = $r1;
   virtualinvoke r2.<A: void m()>();
   return;
}
}
```

receiver

signature of method

known at compile time

NOT KNOWN AT COMPILE TIME

Soot Abstractions. Look up API!

- Abstracts program constructs
- Some basic Soot classes and interfaces
  - **SootClass**
  - **SootMethod**
    - **SootMethod sm; sm.isMain(), sm.isStatic(),** etc.
  - **Local**
    - **Local l; … l.getType()**
  - **InstanceInvokeExpr**
    - Represents an instance (as opposed to static) invoke expression
    - **InstanceInvokeExpr iie; … receiver = iie.getBase();**

Resources

- Github project: https://github.com/soot-oss/soot
- Javadoc: https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/jdoc/

4 Kinds of Calls[1]

- Constructor/Super Call:

```
A a = new A();   →   $r1 = new A;
                     specialinvoke $r1.<A: void <init>()>();
```

- Virtual Call:

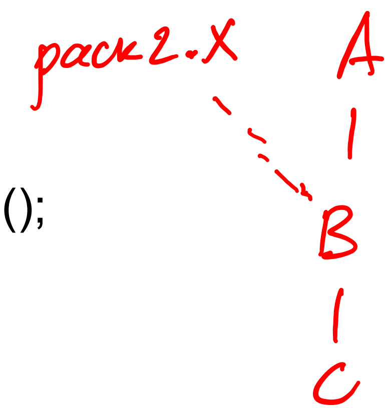
```
a.m();   →   virtualinvoke r2.<A: void m()>();
```

- Static Call:

```
sm();   →   staticinvoke <A: void sm()>();
```

- Interface Call:

```
x.m();   →   interfaceinvoke r0.<pack2.X: void m()>();
```



1. We should not need to worry about dynamicInvoke. (Soot does support it.)

Outline of Today's Class

- Program analysis in practice
  - Program analysis frameworks
    - Soot program analysis framework
    - Ghidra framework
  - Analysis scope and approximation
- Overview of class analysis framework (HW2)
- Class analysis

Analysis Scope

- **Intra**procedural analysis
  - Scope is the CFG of a single subroutine
  - Assumes no call and returns in routine, or models calls and returns
  - What we did so far
- **Inter**procedural analysis
  - Scope of analysis is the ICFG (**Inter**procedural CFG), which models flow of control between routines

Analysis Scope

- **Whole-program** analysis
  - Usually, assumes entry point “main”
  - Application code + libraries
    - Intricate interdependences, e.g., Android apps
- **Modular** analysis
  - Scope either a library without entry point
  - or application code with missing libraries
  - … or a library that depends on other missing libraries

Approximations

- Once we tackle the "whole program" maintaining a solution per program point (i.e., in(j) and out(j) sets) becomes too expensive

- Approximations
  - Transfer function space
  - Lattice
  - Context sensitivity
  - Flow sensitivity

Context Sensitivity

- So far, we studied **intraprocedural analysis**
- Once we extend to **interprocedural analysis** the issue of "context sensitivity" comes up
- Interprocedural analysis can be context-insensitive or context-sensitive
  - In our Java homework, we'll work with context-insensitive analyses
  - We'll talk more about context-sensitive analysis

Context Insensitivity

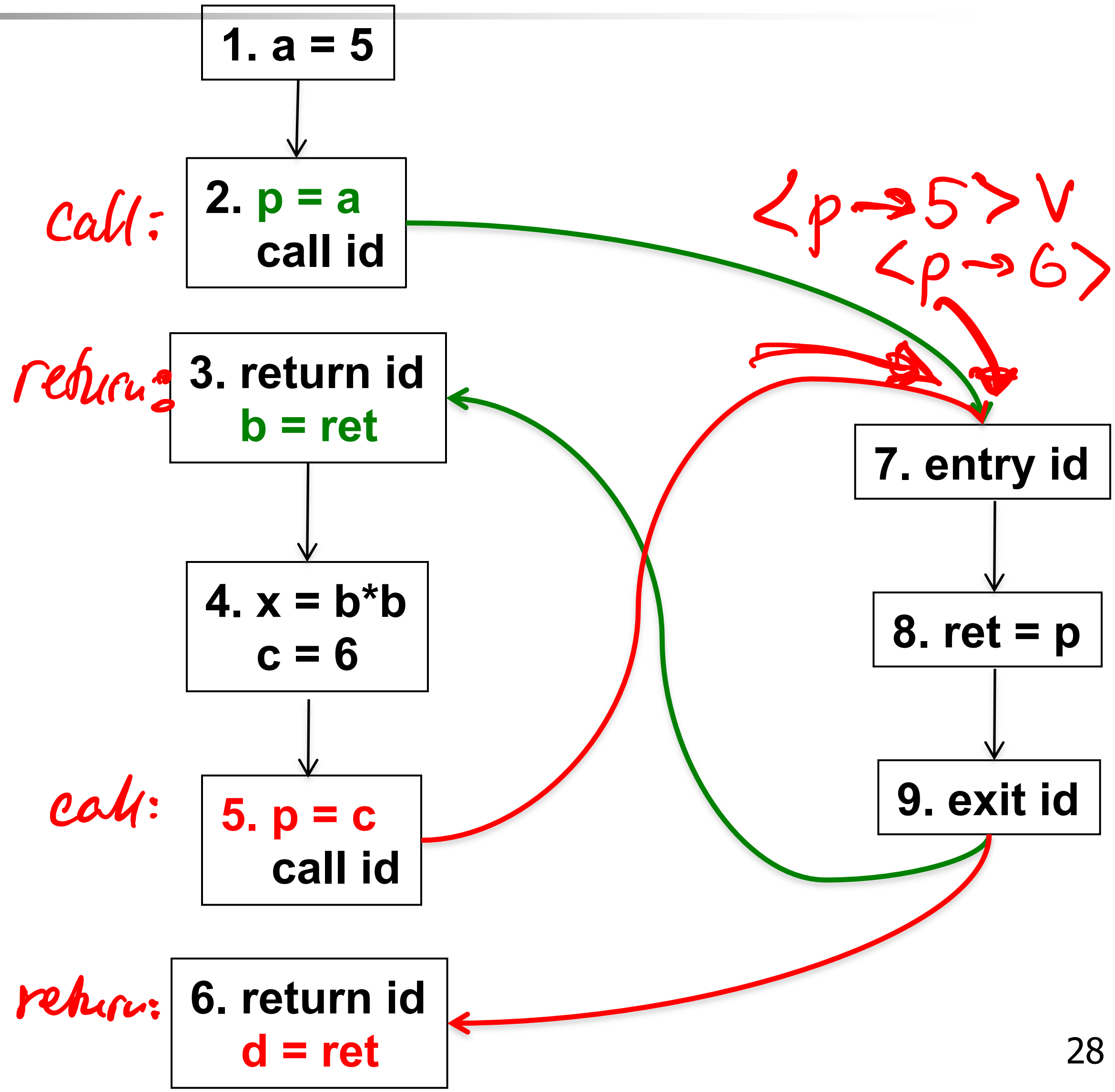
- Context-insensitive analysis makes one big CFG; reduces the problem to standard dataflow, which we know how to solve
- Treats implicit assignment of actual-to-parameter and return-to-left_hand_side as explicit assignment
  - E.g., **x = id(y)** where **id: int id(int p) { return p; }**

    adds **p = y** // flow of values from arg to param

    and **x = ret** // flow of return to left_hand_side
- Can be flow-sensitive or flow-insensitive

Context Insensitivity

```
int id(int p) {
  return p;
}
      a = 5;
  2: b = id(a);
      x = b*b;
      c = 6;
 5: d = id(c);
```



Flow Sensitivity

- Flow-sensitive vs. flow-insensitive analysis
- Flow-sensitive analysis maintains the CFG and computes a solution per each node in CFG (i.e. each program point)
  - Standard dataflow analysis is flow-sensitive
- For large programs, maintaining CFG and solution per program point does not scale

Flow Insensitivity

- Flow-insensitive analysis discards CFG edges and computes a single solution S

- A "declarative" definition, i.e., specification:
  - Least solution S of equations $S = f_j(S) \vee S$

  - Points-to analysis is an example where such a solution makes sense!

Flow Insensitivity

- An "operational" definition. A worklist algorithm:

```
S = 0, W = { 1, 2, … n } /* all nodes */
while W ≠ Ø do {
   remove j from W
   S = fj(S) V S
   if S changed then
      W = W U { k | k is "successor" of j }
}
```

- "successor" is not CFG successor nodes, but more generally, nodes **k** whose transfer function $\mathbf{f_k}$ may be affected as a result of the change in S by **j**

Your Homework

- A bunch of flow-insensitive, context-insensitive analyses for Java
  - RTA, XTA, other
  - Simple property space
  - Simple transfer functions
    - E.g., in fact, RTA gets rid of most CFG nodes, processes just 2 kinds of nodes!
  - Millions of lines of code in seconds

Homework

- Install and run starter code
  - Please let me as soon as possible if you have issues
  - Frameworks are very fragile. They anger a lot
- Look into your git_repo/sootOutput directory and study Jimple
- Study framework code and API
  - Soot API
  - Class analysis framework API

Homework

- Overview of class analysis framework

- We'll discuss more on Thursday
- Come prepared with questions

Outline of Today's Class

- Constant propagation (catchup)
- Program analysis in practice
  - Program analysis frameworks
    - Soot program analysis framework
    - Ghidra framework
  - Analysis scope and approximation
- Class analysis

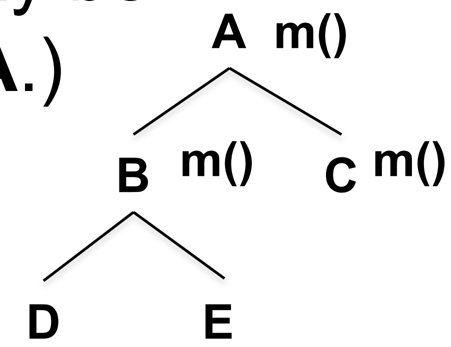
Class Analysis

- Problem statement: What are the **classes** of objects that a (Java) **reference** variable may refer to at runtime?

- Class Hierarchy Analysis (CHA)
- Rapid Type Analysis (RTA)
- XTA
- 0-CFA
- Points-to Analysis (PTA)

Applications of Class Analysis

- Call graph construction
  - At virtual call **r.m()**, what methods may be called? (Assuming **r** is of static type **A**.)



- Virtual call resolution
  - If analysis proves that a virtual call has a single target, it can replace it with a direct call
  - An OOPSLA'96 paper by Holzle and Driesen reports that C++ programs spend 5% of their time in dispatch code. For "all virtual", it is 14%

Boolean Expression Hierarchy

```
public abstract class BoolExp {
   public boolean evaluate(Context c);
}
```

```
public class Constant extends BoolExp {
   private boolean constant;
   public boolean evaluate(Context c) {
     return constant;
   }
}
```

```
public class VarExp extends BoolExp {
    private String name;
    public boolean evaluate(Context c) {
        return c.lookup(name);
    }
}
```

Boolean Expression Hierarchy

```
public class AndExp extends BoolExp {
   private BoolExp left;
   private BoolExp right;

   public AndExp(BoolExp left, BoolExp right) {
      this.left = left;
      this.right = right;
   }
   public boolean evaluate(Context c) {
       return left.evaluate(c) && right.evaluate(c);
    }
}
```

Boolean Expression Hierarchy

```
public class OrExp extends BoolExp {
   private BoolExp left;
   private BoolExp right;

   public OrExp(BoolExp left, BoolExp right) {
      this.left = left;
      this.right = right;
   }
   public boolean evaluate(Context c) {
      return left.evaluate(c) || right.evaluate(c);
   }
}
```

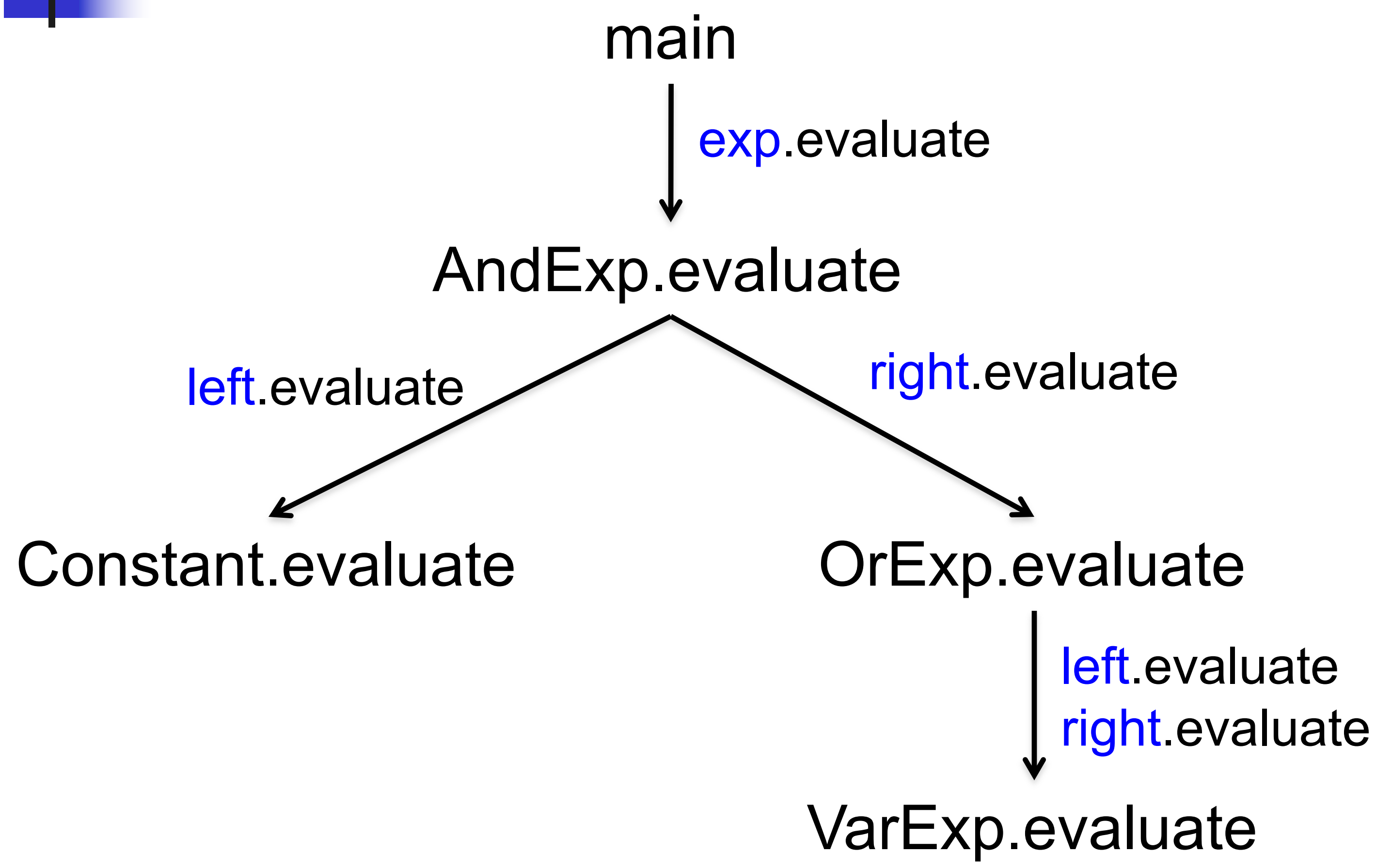
A Client of the Boolean Expression Hierarchy

```
main() {
  Context theContext = new …
  BoolExp x = new VarExp(“X”);
  BoolExp y = new VarExp(“Y”);
  BoolExp exp = new AndExp(
                       new Constant(true), new OrExp(x, y) );
  theContext.assign(x, true);
  theContext.assign(y, false);
  boolean result = exp.evaluate(theContext);
}
```

exp: {AndExp}

At runtime, exp can refer to an object of class AndExp,
but it cannot refer to objects of class OrExp, Constant or VarExp!

Call Graph Example (Partial)



Class Hierarchy Analysis (CHA)

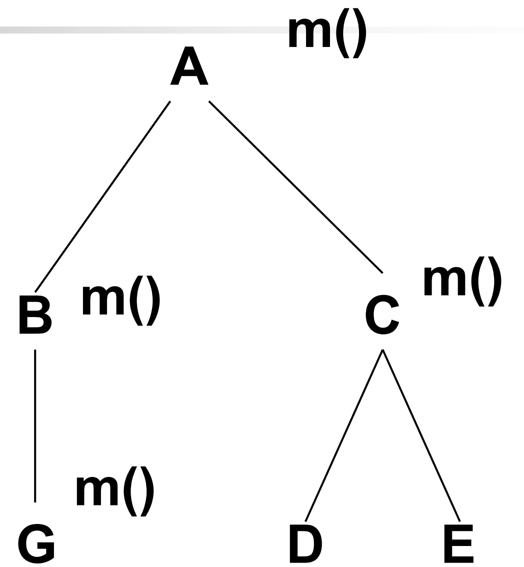
- Attributed to Dean, Grove and Chambers:
  - Jeff Dean, David Grove, and Craig Chambers, "Optimization of OO Programs Using Static Class Hierarchy Analysis", ECOOP' 95

- Simplest way of inferring information about reference variables --- just look at class hierarchy

Class Hierarchy Analysis (CHA)

- In Java, if a reference variable **r** has type **A**, **r** can refer only to objects that are concrete subclasses of **A**. Denoted by **SubTypes(A)**
  - Note: refers to Java subtype, not true subtype
  - Note: **SubTypes(A)** notation due to Tip and Palsberg (OOPSLA'00)
- At virtual call site **r.m(),** we can find what methods may be called based on the hierarchy information

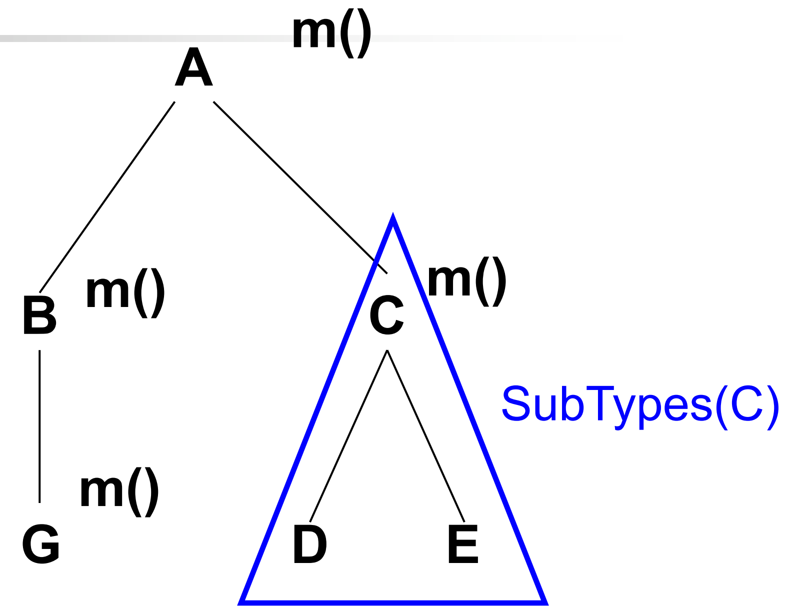
Example




```
public class A {
    public static void main() {
        A a;
        D d = new D();
        E e = new E();
        if (…) a = d; else a = e;
        a.m();
    }
}
public class B extends A {
    public void foo() {
        G g = new G();
    }
} … // no other creation sites or calls in the program
```

Example

```
public class A {
    public static void main() {
        A a;
        D d = new D();
        E e = new E();
        if (…) a = d; else a = e;
        a.m();
    }
}
public class B extends A {
    public void foo() {
        G g = new G();
    }
} …
```

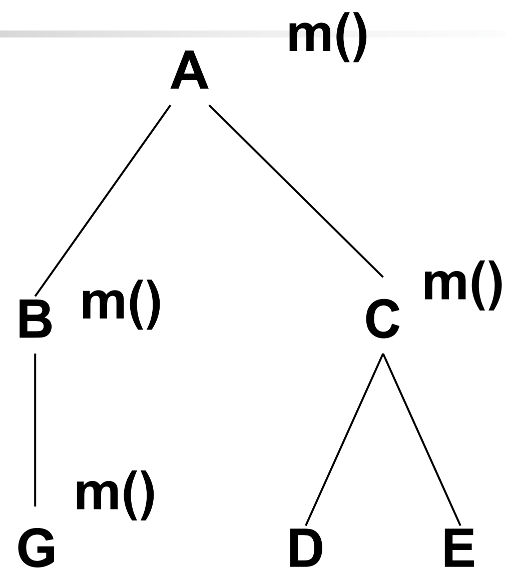


SubTypes(A) = **{ A, B, C, D, E, G }**

SubTypes(B) = **{ B, G }**

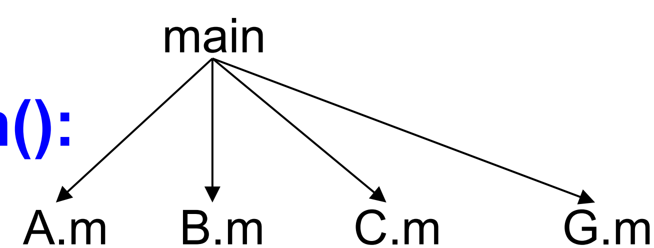
Example

```
public class A {
    public static void main() {
        A a;
        D d = new D();
        E e = new E();
        if (…) a = d; else a = e;
        a.m();
    }
}
public class B extends A {
    public void foo() {
        G g = new G();
    }
} …
```



a: **SubTypes(StaticType(a)) = SubTypes(A)**
= { A, B, C, D, E, G }

a.m():



CHA as Reachability Analysis

R denotes the set of reachable methods

1. **{** main **}** ⊑ **R** // Algo: initialize **R** with **main**
2. for each method **m** ∈ **R**,
each virtual call **y.n(z)** in **m**,
each class **C** in **SubTypes(StaticType(y))** and
n', where **n' = resolve(C,n)**

    { **n'** **}** ⊑ **R** // Algo: add **n'** to **R**

    (Practical concerns: must consider direct calls too!)