



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


1



Announcements

- HW1 due today
- HW2 posted
 - Your task is to set the infrastructure locally
 - Start as soon as you can
 - Ask questions on Submitty, in class, in office hours
 - Run Soot on toy programs and study Jimple IR

2




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

CSCI 4450/6450, A Milanova 3

3



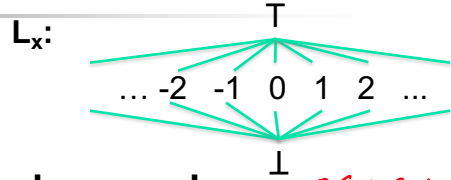
Outline of Today's Class

- Constant propagation (recap)
- Program analysis in practice
 - Program analysis frameworks
 - Soot program analysis framework
 - Ghidra framework
 - Analysis scope and approximation
- Class analysis *for Java*

CSCI 4450/6450, A Milanova 4

4

Constant Propagation fits into Monotone Dataflow Framework



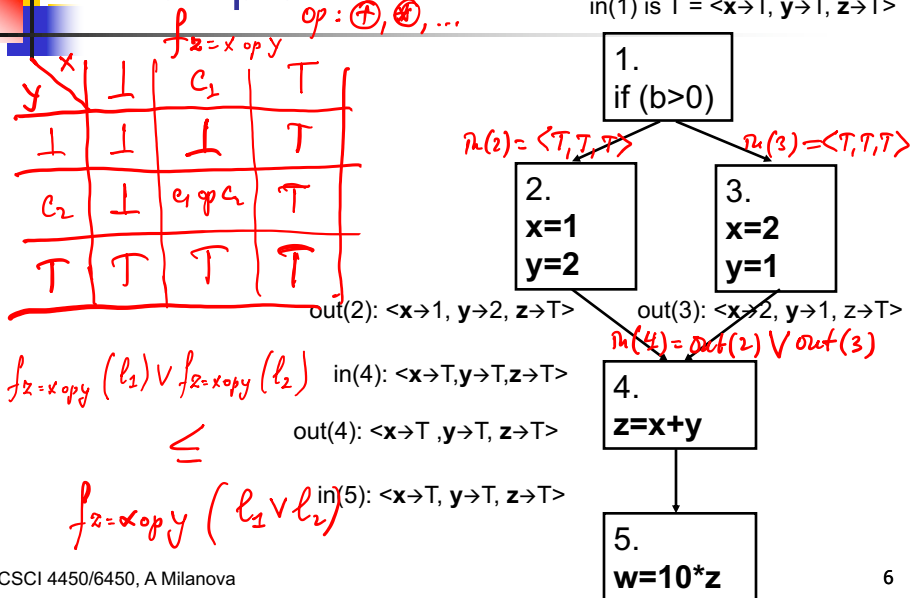
- Property space
 - Product lattice $L = L_x \times L_y \times \dots \times L_z$ 0 ≤ 1 ? NO
 - **L satisfies the ACC** 1 ≤ T ? YES
- and
- Function space $F: L \rightarrow L$ is monotone
- Thus, analysis fits into the monotone dataflow framework and can be solved using the worklist algorithm

CSCI 4450/6450, A Milanova

5

5

Example



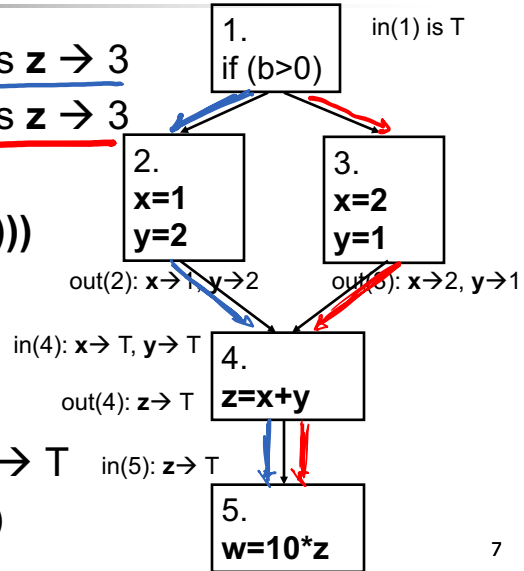
CSCI 4450/6450, A Milanova

6

6

Constant Propagation is Monotone but Not Distributive!

- $f_4(f_2(f_1(T)))$ computes $z \rightarrow 3$
- $f_4(f_3(f_1(T)))$ computes $z \rightarrow 3$
- Thus, MOP at 5
 $f_4(f_2(f_1(T))) \vee f_4(f_3(f_1(T)))$
 computes $z \rightarrow 3$



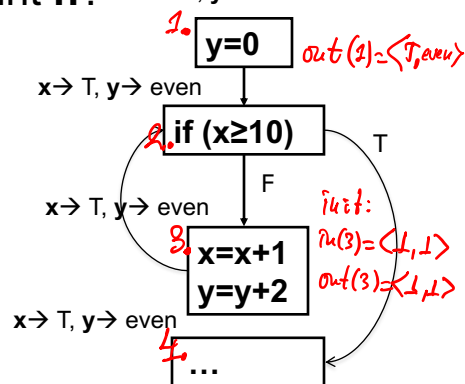
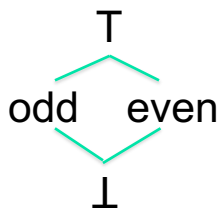
MFP at 5 computes $z \rightarrow T$
 (i.e., z is NOT a const)

7

More Product Lattices

- Problem statement: Is integer variable x odd or even at program point n ? $x \rightarrow T, y \rightarrow T$

■ L_x :



CSCI 4450/6450, A Milanova (Example program from MIT OCW Program Analysis)

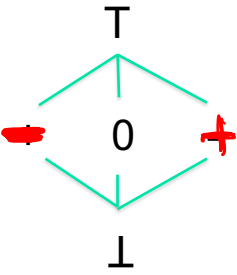
8

8

More Product Lattices

- Problem statement: What **sign** does a variable hold at a given program point, i.e., is it positive, negative, or 0

- L_x :



E.g., $\langle x \rightarrow +, y \rightarrow T, z \rightarrow 0 \rangle$

CSCI 4450/6450, A Milanova 9

9


So far and moving on

- Intraprocedural dataflow analysis
 - CFGs, lattices, transfer functions, worklist algorithm, etc.
 - Classical analyses

- Interprocedural analysis
- Analysis scope and approximation

CSCI 4450/6450, A Milanova 10

10




Program Analysis in Practice

- Program analysis frameworks
 - LLVM
 - Ghidra
 - Soot *Jimple*
API
 - WALA, other

CSCI 4450/6450, A Milanova 11

11




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

12

12




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>

CSCI 4450/6450, A Milanova 13

13



Overview of Soot

Class files/APK

↓

JIMPLIFY

↓

ANALYSIS/
OPTIMIZATION

↓

Optimized jimple

↓

Some IR

↓

Class files/APK

CSCI 4450/6450, A Milanova 14

14

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**
 - Intraprocedural dataflow analysis framework
 - Points-to analysis for Java
 - IR from Dalvik and taint analysis
 - Other analyses and optimizations

15

15

Jimple

```

ojava --> Jimple
        |
        v
        Soot
.class --> Soot
    
```

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

Java:

```

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

Jimple:

```

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

(continues on next slide...)

CSCI 4450/6450, A Milanova 16

16

Jimple

<p>Java:</p> <pre> public class A { main(String[] args) { A a = new A(); a.m(); } public void m() { } } </pre>	<p>Jimple:</p> <pre> ... public void m() { A r0; r0 := @this: A; return; } ... </pre> <p><i>instance method</i> <i>typed reference variable</i></p>
---	---

CSCI 4450/6450, A Milanova17


17

Jimple

<p>Java:</p> <pre> public class A { main(String[] args) { A a = new A(); a.m(); } public void m() { } } </pre>	<p>Jimple:</p> <pre> ... main(java.lang.String[]) { java.lang.String[] r0; A \$r1, r2; r0 := @parameter0: java.lang.String[]; \$r1 = new A; ALLOCATION STATE specialinvoke \$r1.<A: void <init>(>)(); CALL STATE r2 = \$r1; virtualinvoke r2.<A: void m(>)(); return; } </pre> <p><i>static target</i></p>
---	---

CSCI 4450/6450, A Milanova18

18




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();**

CSCI 4450/6450, A Milanova 19

19



Resources

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

<https://javadoc.io/doc/ca.mcgill.sable/soot/latest/index.html>

CSCI 4450/6450, A Milanova 20

20

Analysis Scope

- **Intraprocedural 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
- **Interprocedural analysis**
 - Scope of analysis is the **ICFG (Interprocedural CFG)**, which models flow of control between routines

CSCI 4450/6450, A Milanova 23

23


Analysis Scope

E.g.

- **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

CSCI 4450/6450, A Milanova 24

24




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

CSCI 4450/6450, A Milanova 25

25



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**

CSCI 4450/6450, A Milanova 26

26

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

27

Context Insensitivity

```

int id(int p) {
  return p;
}

a = 5;
2: b = id(a);
x = b*b;
c = 6;
5: d = id(c);

```

Handwritten notes:
 $\lambda x. x$
 b here?
 "Ground truth": b is 5 // b is T
 Context-insensitive: b is T
 d is 6


main;

```

graph TD
    1[1. a = 5] --> 2[2. p = a  
call id]
    2 --> 3[3. return id  
b = ret]
    3 --> 4[4. x = b*b  
c = 6]
    4 --> 5[5. p = c  
call id]
    5 --> 6[6. return id  
d = ret]
    7[7. entry id] --> 8[8. ret = p]
    8 --> 9[9. exit id]
    
    2 --> 7
    3 --> 2
    5 --> 7
    6 --> 9
  
```

Handwritten annotations:
 path 1: 2, 3, 7, 8, 9, 3, 4
 path 2: 2, 3, 7, 8, 9, 3, 4, 5, 7, 8, 9, 3, 4
 id:
 b is 5
 d is 6

28




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

CSCI 4450/6450, A Milanova 29

29



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 $S = f_j(S) \forall S$
 - Points-to analysis is an example where such a solution makes sense!

CSCI 4450/6450, A Milanova 30

30



Flow Insensitivity

- An “operational” definition. A worklist algorithm:

```
S = 0, W = { 1, 2, ... n } /* all nodes */
while W ≠ ∅ do {
  remove j from W
  S = fj(S) ∨ S
  if S changed then
    W = W ∪ { k | k is “successor” of j }
}
```

- “**successor**” is not CFG successor nodes, but more generally, nodes **k** whose transfer function **f_k** may be affected as a result of the change in **S** by **j**

31

31




Your Homework

- A bunch of flow-insensitive, context-insensitive analyses for Java
 - **RTA**, 0-CFA, PTA, 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

CSCI 4450/6450, A Milanova

32

32




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

CSCI 4450/6450, A Milanova 33

33



Homework

- Overview of class analysis framework

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

CSCI 4450/6450, A Milanova 34

34