Dataflow Analysis in Practice: Program Analysis Frameworks, Analysis Scope
Announcements

- Homework 1 due on Thursday
  - Questions?

- Will have Quiz 1 in Submitty soon
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 (last time)
    - Points-to analysis

- Program analysis in practice
Constant Propagation: Lattice

- **Lattice** $L_{x, \leq}$

- Dataflow lattice $L$ is the **product lattice** of $L_x$
  - $l_1, l_2$ in $L$, $l_1 \leq l_2$ iff $l_1_x \leq l_2_x$ for every variable $x$
  - $l_1 \lor l_2$ amounts to $l_1_x \lor l_2_x$ for every variable $x$
  - Merge operator is join of $L$

- Does product lattice satisfy the ACC?
More Product Lattices

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

- $L_x:

  \[
  \begin{array}{c}
  \top \\
  \downarrow \\
  \text{odd} \\
  \downarrow \\
  \text{even} \\
  \top
  \end{array}
  \]
More Product Lattices

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

$L_x$:

- $T$
- Odd
- Even
- $\bot$

Example program from MIT OCW Program Analysis:

```plaintext
x = x + 1
y = y + 2
... if (x ≥ 10)
    T
    y = 0
    if (x ≥ 10)
        T
        x = x + 1
        y = y + 2
        ...
```
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., < x=+, y=T, z=0 >
Outline of Today’s Class

- Non-distributive analysis
  - Constant propagation (last time)
  - Points-to analysis

- Program analysis in practice
  - Program analysis frameworks
    - Soot program analysis framework
    - Ghidra framework
  - Analysis scope and approximation
Points-to Analysis

- Problem statement: What memory locations may a pointer variable point to?

- Many applications!
  - Enables compiler optimizations
    1. a = 1;
    2. *p = b;
    3. s = a*a;
    1. a = x*y*z+x;
    2. *p = b;
    3. s = x*y*z+x;
  - Static debugging tools, static taint analysis tools
Points-to Graph: Example

Example 1:

```c
int a, b;
int *p1, *p2;
p1 = &a;
p2 = p1;
*p2 = 1;
```
Points-to Graph: Example

Example 2:

```c
int a, b = 15;
int *p1, *p2;
int **p3;
p3 = &p1;
p1 = &a;
p2 = *p3;
*p2 = b;
```
Points-to Analysis (for a C-like language)

- Assume the following 4 simple statements
  (1) address taken \( p = \&q \)
  (2) propagation \( p = q \)
  (3) indirect read \( p = *q \)
  (4) indirect write (update) \( *p = q \)

- We can preprocess any C program into a sequence of statements of these kinds
Real-world Points-to, Preprocessing

```c
struct Account {
    ... void * activeOrders[16];
} g_Account;

void create_order() {
    Order * o = malloc(sizeof(Order));
    ... g_Account.activeOrders[i] = o; ... }

void create_sl_order() {
    Sl_Order * slo = malloc(sizeof(Sl_Order));
    ... g_Account.activeOrders[i] = slo; ... }

void print_menu() {
    if (...) {
        ((Order *)g_Account.activeOrders[i])->infoFunc(...)
    } else {
        ((Sl_Order *)g_Account.activeOrders[i])->infoFunc(...)
    }
}
```
Real-world Points-to, Preprocessing

```c
struct Account {
    ... void * activeOrders[16];
} g_Account;

void create_order() {
    Order * o = malloc(sizeof(Order));
    ... g_Account.activeOrders[i] = o; ... }

void create_sl_order() {
    Sl_Order * slo = malloc(sizeof(Sl_Order));
    ... g_Account.activeOrders[i] = slo; ... }

void print_menu() {
    if (...) {
        ((Order *)g_Account.activeOrders[i])->infoFunc(...)
    } else {
        ((Sl_Order *)g_Account.activeOrders[i])->infoFunc(...)
    }
}
```
g_Account = &orders_array (1)

o = &order_heap; (1)
*g_Account = o; (4)
(or *g_Account.activeOrders = o)

slo = &sl_order_heap;
*g_Account = slo;

t1 = *g_Account; (3)
(or t1 = *g_Account.activeOrders)
func1 = *t1; (or func1 = *t1.infoFunc)

t2 = *g_Account;
func2 = *t2;
Points-to Analysis: Property Space

- Lattice $\mathbf{L}, \leq$
  - Lattice of the subsets over all edges $p \rightarrow q$ where $p$ and $q$ are program variables
  - ... or in simpler terms, lattice elements are points-to graphs, e.g., $V$ is points-to graph union, $0$ of $L$ is empty graph, $1$ of $L$ is complete graph

V is points-to graph union
0 of L is empty graph
1 of L is complete graph
Points-to Analysis: Transfer Functions

(1) $f_{p=\& q}$: “kill” all points-to edges from $p$, and “generate” a new points-to edge from $p$ to $q$

(2) $f_{p=q}$: “kill” all points-to edges from $p$; “generate” new points-to edges from $p$ to every $x$, such that $q$ points to $x$ in incoming points-to graph $in(j)$
Points-to Analysis: Transfer Functions

(3) $f_{p=q}$: “kill” all points-to edges from $p$; “generate” new points-to edges from $p$ to every $x$, s.t. there is $y$ where $q$ points to $y$, and $y$ points to $x$ in $\text{in}(j)$

(4) $f_{p=q}$: Do not kill! Can you think of a reason why? “Generate” new points-to edges from every $y$ to every $x$, such that $p$ points to $y$ and $q$ points to $x$
Points-to Analysis is Monotone

To argue monotonicity we must show that if $Pt_1$ is $\leq$ (subset of) $Pt_2$, then $f(Pt_1) \leq f(Pt_2)$ for each transfer function $f$

1. $Pt_1 \leq Pt_2$ then $f_{p=q}(Pt_1) \leq f_{p=q}(Pt_2)$
2. $Pt_1 \leq Pt_2$ then $f_{p=q}(Pt_1) \leq f_{p=q}(Pt_2)$
3. $Pt_1 \leq Pt_2$ then $f_{p=q}(Pt_1) \leq f_{p=q}(Pt_2)$
4. $Pt_1 \leq Pt_2$ then $f_{p=q}(Pt_1) \leq f_{p=q}(Pt_2)$
… but it is not distributive!

- Because of updates!
Points-to Analysis is Not Distributive

What \( f \) for \(*p = q\) does: Adds edges from each variable that \( p \) points to (\( x \) and \( z \)), to each variable that \( q \) points to (\( y \) and \( w \)). Result is 4 new edges: from \( x \) to \( y \) and to \( w \) and from \( z \) to \( y \) and to \( w \).
MFP vs. MOP for Points-to

1. if (n>0)

2. p=&x;
   q=&y;

3. p=&z;
   q=&w;

4. *p=q

\[ \text{in}_{PT}(4) = \text{out}_{PT}(2) \lor \text{out}_{PT}(3) \]

\[ \text{out}_{PT}(4) = f_{*p=q}(\text{in}_{PT}(4)) \]

\[ \text{in}_{PT}(5) = \text{out}_{PT}(4) \]

\[ \text{in}_{PT}(5) = \text{out}_{PT}(4) \]

CSCI 4450/6450, A Milanova
So far and moving on

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

- Program analysis frameworks
- **Inter**procedural analysis
- Analysis scope and approximation
Soot: a framework for analysis and optimization of Java/Dalvik bytecode

- [https://soot-oss.github.io/soot/](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/](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](https://github.com/soot-oss/soot)
Overview of Soot

Class files/APK → JIMPLIFY → ANALYSIS/OPTIMIZATION → Optimized jimple → Some IR → Class files/APK
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
Run soot: `java soot.Main --jimple A`
(need paths)

```java
public class A extends java.lang.Object {

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

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

(continues on next slide…)
public class A {
    main(String[] args) {
        A a = new A();
        a.m();
    }
}

public class A {
    ...
Java:

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

Jimple:

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

CSCI 4450/6450, A Milanova
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

4 Kinds of Calls

- **Constructor/Super Call:**
  
  ```
  A a = new A();
  $r1 = new A;
  specialinvoke $r1.<A: void <init>()>();
  a.m();
  virtualinvoke r2.<A: void m>()();
  sm();
  staticinvoke <A: void sm>()();
  interfaceinvoke r0.<pack2.X: void m>()();
  ```

- **Virtual Call:**

- **Static Call:**

- **Interface Call:**

1. We should not need to worry about dynamicInvoke. (Soot does support it.)
Outline of Today’s Class

- Non-distributive analysis
  - Constant propagation (last time)
  - Points-to analysis

- Program analysis in practice
  - Program analysis frameworks
    - Soot – reverses bytecode/Dalvik into Jimple IR
    - Ghidra – reverses x86 into Pcode IR
  - Analysis scope and approximation
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
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., \( \text{in}(j) \) and \( \text{out}(j) \) sets) becomes too expensive

- Approximations
  - Transfer function space
  - Lattice
  - Context sensitivity
  - Flow 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., \( x = \text{id}(y) \) where \( \text{id} : \text{int id(int p)} \{ \text{return p;} \} \)
  - Adds \( p = y \quad // \text{flow of values from arg to param} \)
  - And \( x = \text{ret} \quad // \text{flow of return to left_hand_side} \)

- Can be flow-sensitive or flow-insensitive.
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 $S = f_j(S) \lor S$

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

An “operational” definition. A worklist-like algorithm:

\[ S = 0, W = \{ 1, 2, \ldots, n \} /* all nodes */ \]

while \( W \neq \emptyset \) do {
    remove \( j \) from \( W \)
    \[ S = f_j(S) \lor S \]
    if \( S \) changed then
        \[ W = W \cup \{ k \mid k \text{ is "successor" of } j \} \]
}

“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 \( j \)
“Classical” Points-to Analysis

- Flow-insensitive, context-insensitive analysis
  - Makes sense for points-to analysis

S = 0 /* initialize solution to empty graph */

F = \{ \textit{f}_1, \textit{f}_2, \ldots \textit{f}_n \} /* all transfer functions, without “kills“, including ones for implicit assignments */

W = \{ \textit{f}_1, \textit{f}_2, \ldots \textit{f}_n \}

while W ≠ Ø do {
  remove \textit{f}_j from W
  S = \textit{f}_j(S) \lor S
  if S changed then
    W = W \cup F /* Safe to add all transfer functions! */
}
“Classical” Points-to Analysis

- Known as Andersen’s Points-to Analysis

\(\text{pts}(p)\) denotes the points-to set of \(p\)

1. \(p = \&a \{ a \} \subseteq \text{pts}(p)\)
2. \(p = q \quad \text{pts}(q) \subseteq \text{pts}(p)\)
3. \(p = *q \quad \text{for each } x \text{ in } \text{pts}(q). \quad \text{pts}(x) \subseteq \text{pts}(p)\)
4. \(*p = q \quad \text{for each } x \text{ in } \text{pts}(p). \quad \text{pts}(q) \subseteq \text{pts}(x)\)

Use \textit{worklist-like algorithm} to compute least solution of these constraints
g_Account = &orders_array;

o = &order_heap;
*g_Account = o;

slo = &sl_order_heap;
*g_Account = slo;

t1 = *g_Account;
func1 = *t1;

t2 = *g_Account;
func2 = *t2;
Your Homework

- A bunch of flow-insensitive, context-insensitive analyses for Java
  - RTA, XTA, and optionally 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