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 $n$?

$L_x$:  

- $x = \text{const}$: $i_n(j) = \langle x \rightarrow \text{v}_x \rangle \Rightarrow$ replace $x \rightarrow \text{v}_x$ with $x \rightarrow \text{even}$ if const is even $x \rightarrow \text{odd}$ otherwise
- $x = y$: replace $x \rightarrow \text{v}_x$ with $x \rightarrow \text{v}_y$
- $x = y + 2$

\[
\begin{array}{cccc}
\text{odd} & \text{even} & \text{odd} & \text{even} \\
\downarrow & \downarrow & \downarrow & \downarrow \\
\bot & \top & \top & \top \\
\end{array}
\]
Problem statement: Is integer variable $x$ odd or even at program point $n$?

$L_x$:

\[
\begin{array}{c}
\text{odd} \\
\uparrow \\
\text{even} \\
\end{array}
\]
Problem statement: What sign does a variable hold at a given program point, i.e., is it positive, negative, or 0?

\[ L_x: \]

\[ < x=+, y=T, z=0 > \]
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. \(s = a*a;\)
  - Static debugging tools, static taint analysis tools
    1. \(a = x*y*z+x;\)
    2. \(*p = b;\)
    3. \(s = x*y*z+x;\)
Example 1:

```c
int a, b;
int *p1, *p2;

p1 = &a;
p2 = p1;
*p2 = 1;
```
Example 2:

```c
int a, b = 15;
int *p1, *p2;
int **p3;
p3 = &p1;
p1 = &a;
p2 = *p3;
*p2 = b;
```

Find where `p3` points to (p1), then copy address stored in p1 (address of a) into p2, thus making p2 point to a.
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
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(…)
    }
}
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

\[ g_{\text{Account}} = \&\text{orders\_array}; \quad (1) \]
\[ o = \&\text{order\_heap}; \quad (1) \]
\[ *g_{\text{Account}} = o; \quad (4) \]
\[ \text{(or } *g_{\text{Account}}.\text{activeOrders} = o) \]

\[ s\text{lo} = \&s\text{l\_order\_heap}; \]
\[ *g_{\text{Account}} = s\text{lo}; \]

\[ t1 = *g_{\text{Account}}; \quad (3) \]
\[ \text{(or } t1 = *g_{\text{Account}}.\text{activeOrders}) \]
\[ \text{func1} = *t1; \quad \text{(or } \text{func1} = *t1.\text{infoFunc}) \]

\[ t2 = *g_{\text{Account}}; \]
\[ \text{func2} = *t2; \]
Points-to Analysis: Property Space

- **Lattice L, ≤**
  - 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
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 $q$

$$\text{In}(j): p \rightarrow q \quad \Rightarrow \quad \text{Out}(j): p \rightarrow q$$

(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 $\text{in}(j)$

$$\text{In}(j): p \rightarrow a_1 \rightarrow x_1 \rightarrow x_2 \rightarrow q \quad \Rightarrow \quad \text{Out}(j): p \rightarrow x_1 \rightarrow x_2 \rightarrow q$$
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 $\text{Pt}_1$ is $\leq$ (subset of) $\text{Pt}_2$, then $f(\text{Pt}_1) \leq f(\text{Pt}_2)$ for each transfer function $f$

\begin{align*}
(1) & \quad \text{Pt}_1 \leq \text{Pt}_2 \text{ then } f_{p=q}(\text{Pt}_1) \leq f_{p=q}(\text{Pt}_2) \\
(2) & \quad \text{Pt}_1 \leq \text{Pt}_2 \text{ then } f_{p=q}(\text{Pt}_1) \leq f_{p=q}(\text{Pt}_2) \\
(3) & \quad \text{Pt}_1 \leq \text{Pt}_2 \text{ then } f_{p=q}(\text{Pt}_1) \leq f_{p=q}(\text{Pt}_2) \\
(4) & \quad \text{Pt}_1 \leq \text{Pt}_2 \text{ then } f_{p=q}(\text{Pt}_1) \leq f_{p=q}(\text{Pt}_2)
\end{align*}
... but it is not distributive!

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

What \( f \) for \( \ast p = q \) 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

5. ...

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

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

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