Announcements

- HW1 due
- HW2 is out
  - Get started with your Submitty git repos, Soot and Jimple, etc.
  - We’ll cover class analysis (RTA, XTA, etc.) on Thursday

Outline of Today’s Class

- One more note on Worklist algorithm
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis
- Analysis scope and approximations

Worklist Algorithm for Forward Dataflow Problems

```java
/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue;
out(1) = f1(in(1));
for m := 2 to n do
  in(m) = 0;
  out(m) = fm(0);
W = {2,…,n} /* put every node but 1 on the worklist */
while W ≠ Ø do {
  remove j from W
  in(j) = V { out(i) | i is predecessor of j }
  out(j) = fj(in(j))
  if out(j) changed then
    W = W U { k | k is successor of j }
}
```

Example. Reach with Bitvectors

**Initialization**

<table>
<thead>
<tr>
<th>(i,1), (k,1)</th>
<th>B1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
<td>pres: 00000</td>
</tr>
<tr>
<td>k=0</td>
<td>gen: 10000</td>
</tr>
</tbody>
</table>

```
000000  111111  111111  10001  10001  01110
```

```
000000  000000  000000  00100  00010  00001
```

**Bitvector:** 0 0 0 0 0 0

```
000000  000000  000000  00100  00010  00001
```

<table>
<thead>
<tr>
<th>(i,4), (k,5)</th>
<th>B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>k=k-1</td>
<td>out(B2) = 00000</td>
</tr>
<tr>
<td>k=k+1</td>
<td></td>
</tr>
<tr>
<td>i=i+1</td>
<td></td>
</tr>
</tbody>
</table>

```
000000  000000  000000  00000  00000  00001
```

```
000000  000000  000000  00000  00000  00001
```

```
000000  000000  000000  00000  00000  00001
```

**Exit**
Constant Propagation (Simple)

- Problem statement: What variables always hold constant values at a given program point

- Example:

1. \( x = 1 \) if \( b > 0 \)
   - in(1): \( x \) is not a const
   - out(1): \( x = 1 \)

2. \( y = z + w \)
   - in(2): \( x = 1 \)
   - out(2): \( x = 2 \)

3. \( y = 0 \)
   - in(3): \( x = 1 \)
   - out(3): \( x = 1 \)

4. \( z = 10^x \)
   - in(4): \( x \) is not a const

Aside: Defining an Analysis

- Define program syntax
  - In practice, we deal with a lot more than the simple abstraction

- Define property space
  - The abstract program state that approximates the concrete program state

- Define transfer function space over syntax
  - Symbolically execute program over abstract state

Aside: Defining an Analysis

- If property space has desired properties
  - is a lattice \( L \), \( \leq \) that satisfies the Ascending Chain Condition
  - merge operator \( V \) is the join of \( L \) and

- Function space \( F: L \rightarrow L \) is monotone
  - then analysis fits the monotone dataflow framework and can be solved using the worklist algorithm

Constant Propagation: Syntax

\[
S ::= S ; | \text{while (} b \text{)} \{ S \} \} | \text{if (} b \text{)} \{ S \} \text{else } \{ S \} \}
\]
\[
| x = V \mid x = V \text{ Op } V
\]
\[
\text{Op ::= + | - | * | /}
\]
\[
V ::= x \mid y \mid z \mid C
\]

- \( x, y, z \) are program variables
- \( C \) is constant
- We have to define transfer functions for \( x = V \) and \( x = V \text{ Op } V \)
Constant Propagation: Property Space

- Associate one of the following values with variable $x$ at each program point

<table>
<thead>
<tr>
<th>value</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (or T)</td>
<td>$x$ is NOT a constant</td>
</tr>
<tr>
<td>C</td>
<td>$x$ has constant value C</td>
</tr>
<tr>
<td>0 (or T)</td>
<td>$x$ is unknown</td>
</tr>
</tbody>
</table>

Constant Propagation: Lattice

- Lattice $L_x \leq$

- Dataflow lattice $L$ is the product lattice of $L_x$
  - $l_1, l_2 \in L$, $l_1 \leq l_2$ if $l_1 x \leq l_2 x$ 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 the product lattice satisfy the ACC?

Product Lattice

- E.g., $<x=1, y=1, z=T>$, $<x=1, y=2, z=3>$, etc. are lattice elements
- E.g., $<x=1, y=2, z=T> \leq <x=T, y=2, z=T>$
- E.g., $<x=1, y=3, z=T> \lor <x=T, y=2, z=T> = <T, T, T>$

Constant Propagation: Transfer Functions

- $j$: $x = C$
  - $f_j$: kill $x \rightarrow \text{val}$, generate $x \rightarrow C$
- $j$: $x = y$
  - $f_j$: kill $x \rightarrow \text{val}$, add $x \rightarrow \text{val}'$, s.t. $y \rightarrow \text{val}'$ in $\text{in(j)}$. $\text{val}$ and $\text{val}'$ are one of
    - $1$: bottom (unknown)
    - $C$: constant
    - $T$: top (not a constant)

Next, we'll argue monotonicity which would give us that Constant Propagation is solvable by the Worklist algorithm

Example

$$\text{in(1)} = T = <x=T, y=T, z=T>$$

1. If $(b>0)$
2. $x=1$
3. $x=2$
4. $z=x+y$
5. $w=10*z$
Not Distributive! A Counter Example

\[ f_d(f_2(f_1(T))) \] computes \( z \rightarrow 3 \)

\[ f_d(f_2(f_1(T))) \] computes \( z \rightarrow 3 \)

Thus, MOP at 5
\[ f_d(f_2(f_1(T))) \lor f_d(f_2(f_1(T))) \] computes \( z \rightarrow 3 \)

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

More Product Lattices

Problem statement: Is integer variable \( x \) odd or even at program point \( n \)?

\[ \begin{array}{c}
\text{if (b>0)} \\
1.
\end{array} \]

\[ \begin{array}{c}
x=1 \\
y=2
\end{array} \]

\[ \begin{array}{c}
x=2 \\
y=1
\end{array} \]

\[ \begin{array}{c}
x=x+1 \\
y=y+2
\end{array} \]

Points-to Analysis: Syntax

Assume the following 4 simple statements

1. \( a = 1 \)
2. \( \star p = b \)
3. \( s = a*a \)
4. \( \star p = q \)

One can transform any program into a sequence of statements of these kinds
Points-to Analysis: Property Space

- Lattice $L$, $\leq$
  - 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., $p_3 \rightarrow p_1 \rightarrow p_2$
  - $V$ 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. $f_{p=q}$: “generate” new points-to edges from $p$ to every $x$, such that $q$ points to $x$ in incoming points-to graph in($j$)

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 a $y$ where $q$ points to $y$, and $y$ points to $x$ in 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 $x$

The Problem with Updates

- Updates (4) $f_{p=q}$ (also known as destructive updates) … are a pain
  - If we drop (4) from our language we get
    1. $p = &a$
    2. $p = q$
    3. $p = *q$
  - Research problems!

Points-to Analysis is Monotone

To argue monotonicity we must show that if $P_1 \leq P_2$ (subset of) $P_{t_2}$, then $f(P_1) \leq f(P_{t_2})$ for each transfer function $f$

1. $P_1 \leq P_2$ then $f_{p=q}(P_1) \leq f_{p=q}(P_{t_2})$
2. $P_1 \leq P_2$ then $f_{p=q}(P_1) \leq f_{p=q}(P_{t_2})$
3. $P_1 \leq P_2$ then $f_{p=q}(P_1) \leq f_{p=q}(P_{t_2})$
4. $P_1 \leq P_2$ then $f_{p=q}(P_1) \leq f_{p=q}(P_{t_2})$

Points-to Analysis is Not Distributive

- ... but it is not distributive!
  - Because of updates!
MFP vs. MOP for Points-to

Andersen’s Points-to Analysis
- Commonly attributed to Lars Andersen [1994]
- More approximation than our earlier formulation: don’t ever “kill”, thus, can maintain a single points-to graph
  - Straight-forward analysis formulation
    - Formulated in terms of subset constraints
    - Solvable by a version of the fixpoint iteration algorithm
  - Commonly referred to as a flow-insensitive, context-insensitive analysis

Andersen’s Points-to Analysis: Examples
Example 1:
- \( p1 = \&a \{ a \} \subseteq \text{pts}(p) \)
- \( p2 = q \subseteq \text{pts}(q) \)
- \( *p2 = q \) for each \( x \) in \( \text{pts}(q) \), \( \text{pts}(x) \subseteq \text{pts}(p) \)
- \( *p1 = q \) for each \( x \) in \( \text{pts}(p) \), \( \text{pts}(x) \subseteq \text{pts}(x) \)

Example 2:
- \( p3 = \&p1 \)
- \( p1 = \&a \)
- \( p2 = p1 \)
- \( *p2 = 1 \)
- \( q = p3 \)
- \( r = *q \)
- \( p1 = \&b \)

Outline of Today’s Class
- One brief note on the Worklist algorithm
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis
- Analysis scope and approximations

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
  - A lot more on this to come!
**Analysis Scope**

- **Whole-program analysis**
  - Assumes entry point “main”
  - Application code + standard 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**

- Dimensions of approximation
  - Transfer function space
  - Lattice

- Flow sensitivity
- Context sensitivity
  (Somewhat poorly defined notions)

**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 essentially discards CFG edges, computes a single solution $S$
  - E.g., Andersen’s points-to analysis is flow-insensitive
  - A “declarative” worklist-like algorithm:
    
    ```
    S = 0
    do {
      for each node $j$ do
        $S = f_j(S) \lor S$
    } while $S$ changes
    ```

- A more “operational” worklist-like algorithm:
  - $S = 0$, $W = \{1, 2, \ldots, n\}^\ast$ all nodes */
    while $W \neq \phi$ do {
      remove $j$ from $W$
      $S = f_j(S) \lor S$
      if $S$ changed then
        $W = W \cup \{k \mid k$ is "successor" of $j}\}$
    }

- Note that "successors" here does not refer to successor nodes in the CFG, but nodes $k$ whose transfer function $f_k$ may contribute to $S$ as a result of the change by $j$

**Context Sensitivity**

- Context-sensitive vs. context-insensitive
- So far, we did intraprocedural analysis
- Once we consider interprocedural analysis
- the issue of context-sensitive vs. context-insensitive comes up
- **Context-insensitive** analysis treats calls and returns as assignments
  - Can be flow-sensitive or flow-insensitive
- A lot more on context-sensitive analysis…
Context Insensitivity

```c
int id(int p) {
    return p;
}

a = 5;
c1: b = id(a);
    x = b*b;
    c = 6;
c2: d = id(c);
```

Your Homework

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