Dataflow Analysis: Non-distributive Analysis
Outline of Today’s Class

- Dataflow frameworks, conclusion
  - Lattices (last time)
  - Transfer functions (last time)
  - Worklist algorithm
- MOP solution vs. MFP solution
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis
Monotone Dataflow Framework

- A problem fits into the dataflow framework if
  - its property space is a lattice $\mathbf{L}$, $\leq$ that satisfies the Ascending Chain Condition
  - its merge operator $V$ is the join of $\mathbf{L}$
  - and
    - its transfer function space $F : \mathbf{L} \rightarrow \mathbf{L}$ is monotone

- Thus, we can make use of a generic solution procedure, known as the worklist algorithm
  - Computes the so-called MFP solution
Worklist Algorithm for Forward Dataflow Problems

/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue; out(1) = f_1(in(1))
for m = 2 to n do in(m) = 0; out(m) = f_m(0)
W = \{2,\ldots,n\} /* put every node but 1 on the worklist */

while W ≠ Ø do {
    remove j from W
    in(j) = \bigvee \{ out(i) \mid i \text{ is predecessor of } j \}
    out(j) = f_j(in(j))
    if out(j) changed then
        W = W U \{ k \mid k \text{ is successor of } j \}
}

\[ i_u^t(j) \leq i_u^{tn}(j) \]
\[ o_u^t(j) \leq o_u^{tn}(j) \]
Worklist Algorithm on Reach

D = all definitions: {(x,1), (x,4), (a,3)}

Poset is $2^D$, $\leq$ is the subset relation $\subseteq$

1. $x = a * b$
2. if $y \leq a * b$
3. $a = a + 1$
4. $x = a * b$
5. goto 2

$W = \{2, 3, 4, 5\}$

Iter 1
remove 2
$\text{in}(2) = \{x, 4\}$
$\text{out}(2) = \{\}$

Iter 2
remove 3
$\text{in}(3) = \{x\}$
$\text{out}(3) = \{a, 3\}$

Iter 3
remove 4
$\text{in}(4) = \{x, 4\}$
$\text{out}(4) = \{\}$

Iter 4
remove 5
$\text{in}(5) = \{(x, 4), (a, 3)\}$
$\text{out}(5) = \{(a, 3)\}$
Termination Argument

Theorem: the algorithm terminates. Why?

Sketch of argument:

A node \( k \) is placed on worklist only if the \( \text{out}(j) \) of a predecessor \( j \) changes. Monotonicity of \( f \) guarantees \( \text{in}^t(j) \leq \text{in}^{t+1}(j) \) and \( \text{out}^t(j) \leq \text{out}^{t+1}(j) \). (Here \( \text{in}^t(j), \text{out}^t(j) \) are the sets at iteration \( t \).)

\( \text{in} \) and \( \text{out} \) sets are elements of \( L \) and \( L \) satisfies the Ascending Chain Condition; thus, there is only a finite number of times each \( \text{out}(j) \) changes.
Correctness Argument

Theorem: Worklist algorithm computes a solution that satisfies the dataflow equations. Why?

Sketch of argument:
Suppose either (1) \( V_{out}(i) \neq in(j) \) or (2) \( out(j) \neq f_j(in(j)) \). For (1) to hold we must have “grown” \( out(i) \) in some iteration and not added successor \( j \) to worklist; this is impossible.
Theorem: Worklist algorithm computes the least solution of the dataflow equations.

Historically, solution computed by worklist algorithm is called the maximal fixpoint solution (MFP solution).

For every node $j$, worklist algorithm computes a solution $MFP(j) = (in(j), out(j))$, such that for every solution $(in'(j), out'(j))$ of the dataflow equations we have $in(j) \leq in'(j)$ and $out(j) \leq out'(j)$.
**Example (Avail)**

1. \( z := x + y \)

   \[
   \text{out}(1) = (\text{in}(1) - E_z) \cup \{x+y\}
   \]

2. if \( z > 500 \)

   \[
   \text{out}(2) = \text{in}(2)
   \]

3. goto 2

\[
\text{in}(3) = \text{out}(2) \\
\text{out}(3) = \text{in}(3)
\]

Equivalent to: \( \text{in}(2) = \{x+y\} \cup \text{in}(2) \)

and recall that \( \cup \) is \( \cap \) (i.e., set intersection).
Outline of Today’s Class

- Dataflow frameworks, conclusion
  - Lattices (last time)
  - Transfer functions (last time)
  - Worklist algorithm
- MOP solution vs. MFP solution

- Non-distributive analyses
  - Constant propagation
  - Points-to analysis
Meet Over All Paths (MOP)

Desired dataflow information at \( n \) is obtained by traversing ALL PATHS from 1 (entry node) to \( n \).

For every path \( p = (1, n_2, n_3, ..., n_k) \) we compute
\[
f_{n_k}(\ldots f_{n_2}(f_1(\text{InitialValue})))
\]

The MOP at entry of \( n \) is \( V f_{n_k}(\ldots f_{n_2}(f_1(\text{InitialValue}))) \) over all paths \( p \) from 1 to \( n \).
MOP vs. MFP

- MOP is an abstraction of the best solution computable with dataflow analysis
  - It is a common assumption in dataflow analysis that all program paths are executable

- MFP is the solution computed by the worklist algorithm

CSCI 4450/6450, A Milanova
MOP vs. MFP

- For *distributive* problems \( MFP = MOP \! \)

- Unfortunately, for *monotone* problems this is not true. But we still have a safe solution: it is a theorem that for monotone problems, \( MFP \geq MOP \)
Safety of a Dataflow Solution

- A safe (also, correct or sound) solution $X$ overestimates the “best” possible dataflow solution, i.e., $X \geq \text{MOP}$

- Historically, an acceptable solution $X$ is one that is better than what we can do with the MFP, i.e., $X \leq \text{MFP}$
Safe Solutions: Reach

U = all definitions: {(x,1),(x,4),(a,3)}  {(x,1),(x,4),(a,3)}

Poset is 2^U, ≤ is the subset relation ⊆

1. x = a * b
2. if y ≤ a * b
3. a = a + 1
4. x = a * b
5. goto 2

CSCI 4450/6450, A Milanova
Safe Solutions: Avail

U = all expressions: \{a*b, a+1, y*z\}
Poset is $2^U$, $\leq$ is the superset relation $\supseteq$

1. $x := a*b$
2. if $y*z \leq a*b$
3. $a := a+1$
4. $x := a*b$
5. goto 2
Outline of Today’s Class

- Dataflow frameworks, conclusion
  - Lattices (last time)
  - Transfer functions (last time)
  - Worklist algorithm
- MOP solution vs. MFP solution

- Non-distributive analyses
  - Constant propagation
  - Points-to analysis
**Constant Propagation (Simple)**

- **Problem statement:** Can variable $x$ hold a constant value at a given program point?

- **Example:**

1. $x = 1$
   
   If $b > 0$
   
   **in:** $x$ is not constant
   
   **out:** $x$ is 1

2. $y = z + w$
   
   $x = 2$
   
   **in:** $x$ is 1
   
   **out:** $x$ is 2

3. $y = 0$
   
   **in:** $x$ is 1
   
   **out:** $x$ is 1

4. $z = 10 \times x$
   
   **in:** $x$ is NOT a constant!
Let’s Fit Analysis into Monotone Dataflow Framework

- If property space has desired properties
  - it is a lattice \( L, \leq \) that satisfies the *Ascending Chain Condition*
  - its merge operator is the join of \( L \)
- 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: 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 ⊥)</td>
<td>$x$ is unknown</td>
</tr>
</tbody>
</table>
Constant Propagation: Lattice

- Lattice $L_x, \leq$
  
  $\bot \leq \top$, $0 \leq \top$

  $l_1 = \langle x \mapsto \bot, y \mapsto \top, z \mapsto \bot \rangle$
  
  $l_2 = \langle x \mapsto \bot, y \mapsto 5, z \mapsto 0 \rangle$

- 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 the product lattice satisfy the ACC?
Product Lattice

- E.g., 
  \(<x \rightarrow \bot, y \rightarrow 1, z \rightarrow T>, <x \rightarrow 1, y \rightarrow 2, z \rightarrow 3>, \text{ etc.}\)
  
  are lattice elements

- E.g., 
  \(<x \rightarrow 1, y \rightarrow 2, z \rightarrow T> \leq <x \rightarrow T, y \rightarrow 2, z \rightarrow T> \quad \text{where } \begin{align*}
  l_{1x} &= 1 \\ l_{1y} &= 2 \\ l_{1z} &= T \\
  l_{2x} &= T \\ l_{2y} &= T \\ l_{2z} &= T
  \end{align*}

- E.g., 
  \(<x \rightarrow 1, y \rightarrow 3, z \rightarrow T> \lor <x \rightarrow T, y \rightarrow 2, z \rightarrow T> = <T, T, T> \)
Product Lattice

Does Product lattice satisfy the ACC?

\(<x \rightarrow T, y \rightarrow T>\)

\(<x \rightarrow T, y \rightarrow 1>\)

\(<x \rightarrow 1, y \rightarrow 1>\)

\(<x \rightarrow 1, y \rightarrow \bot>\)

Length of maximal ascending chain is

\(2 \times N\) where \(N\) is

the number of variables in the tuple.
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
    - \( \bot \): bottom (unknown)
    - \( C \): constant
    - \( T \): top (not a constant)
Constant Propagation: Transfer Functions

\[ j: \ x = y \ Op \ z \]

\[ f_j: \ \text{kill: } x \rightarrow \text{val} \]

\[ \text{gen:} \]

If \( y \rightarrow c_1 \) and \( z \rightarrow c_2 \) in \( \text{in}(j) \), then \( x \rightarrow c_1 \ Op \ c_2 \)

else if \( y \rightarrow T \) or \( z \rightarrow T \) in \( \text{in}(j) \), then \( x \rightarrow T \)

else \( x \rightarrow \bot \)

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

1. if \( b > 0 \)

2. \( x = 1 \)
   \( y = 2 \)
   out(2): \( <x\rightarrow 1, y\rightarrow 2, z\rightarrow T> \)

3. \( x = 2 \)
   \( y = 1 \)
   out(3): \( <x\rightarrow 2, y\rightarrow 1, z\rightarrow T> \)

4. \( z = x + y \)
   out(4): \( <x\rightarrow T, y\rightarrow T, z\rightarrow T> \)

5. \( w = 10 \times z \)
   in(4): \( <x\rightarrow T, y\rightarrow T, z\rightarrow T> \)

in(1) is \( T = <x\rightarrow T, y\rightarrow T, z\rightarrow T> \)
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

$f_4(f_2(f_1(T))) \lor f_4(f_3(f_1(T)))$ computes $z \rightarrow 3$

MFP at 5 computes $z \rightarrow T$

(i.e., $z$ is NOT a const)
More Product Lattices

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

- $L_x$:

  $$
  \begin{array}{c c c}
  T & \text{odd} & \text{even} \\
  \downarrow & & \\
  y=0 & x \rightarrow T, y \rightarrow \text{even} & \\
  \text{if } (x \geq 10) & x=x+1, y=y+2 & \text{even} \\
  \text{...} & \\
  \end{array}
  $$
Problem statement: What sign does a variable hold at a given program point, i.e., is it positive, negative, or 0

\[ L_x: \]

\[ \perp \]

\[ \begin{array}{ccc}
    + & 0 & - \\
    \downarrow & & \downarrow \\
    \top & & \bot \\
\end{array} \]

E.g., \( < x \rightarrow +, y \rightarrow T, z \rightarrow 0 > \)
Outline of Today’s Class

- Dataflow frameworks, conclusion
  - Lattices (last time)
  - Transfer functions (last time)
  - Worklist algorithm
- MOP solution vs. MFP solution

- Non-distributive analyses
  - Constant propagation
  - Points-to analysis
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 and taint analysis tools
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
Points-to Analysis: Property Space

- **Lattice \( L, \leq \)**
  - Lattice of the subsets over edges \( p \rightarrow q \) where \( p \) and \( q \) are (names of) memory locations
  - ... 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 Graphs Pt

- Nodes are names of memory locations
  - Program variables, a, p:
    - \( p = \& a \)
  - But also heap locations:
    - \( p = \text{malloc}(\text{sizeof}(\text{int})) \) // h1

- Edges represent points-to relations
  - E.g., \( p \rightarrow a \), read: “p points to a”
  - E.g., \( p \rightarrow h1 \), read: “p points to heap location h1”
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 \(f\)

\[
\begin{align*}
(1) & \quad Pt_1 \leq Pt_2 \text{ then } f_{p=q} (Pt_1) \leq f_{p=q} (Pt_2) \\
(2) & \quad Pt_1 \leq Pt_2 \text{ then } f_{p=q} (Pt_1) \leq f_{p=q} (Pt_2) \\
(3) & \quad Pt_1 \leq Pt_2 \text{ then } f_{p=*q} (Pt_1) \leq f_{p=*q} (Pt_2) \\
(4) & \quad Pt_1 \leq Pt_2 \text{ then } f_{*p=q} (Pt_1) \leq f_{*p=q} (Pt_2)
\end{align*}
\]
... but it is not distributive!

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

\[ p = \&x; \]
\[ q = \&y; \]

\[ p = \&z; \]
\[ q = \&w; \]

\[ *p = q \]

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

in\textsubscript{PT}(4) = out\textsubscript{PT}(2) \lor out\textsubscript{PT}(3)

5. ...

out\textsubscript{PT}(4) = f\ast p=q (in\textsubscript{PT}(4))

in\textsubscript{PT}(5) = out\textsubscript{PT}(4)

CSCI 4450/6450, A Milanova
Next Time

- Putting this into practice

- Program analysis frameworks
  - Soot
  - Ghidra