Dataflow Analysis: Non-distributive Analysis
Announcements

- Homework 1?
  - Will extend to Tuesday, February 16th

- Homework 2 next week
Outline of Today’s Class

- Dataflow frameworks, conclusion
- MOP solution vs. MFP solution
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis
Dataflow Framework

Equations:

\[ \text{in}(j) = \mathbf{V} \text{out}(i) \quad \text{out}(j) = f_j(\text{in}(j)) \]

\[ \text{i in pred}(j) \]

where:

- \( \text{in}(j), \text{out}(j) \) are elements of a property space
- \( f_j \) is the transfer function associated with node \( j \)
- \( \mathbf{V} \) is the merge operator
Monotone Dataflow Framework

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

- Thus, we can make use of a generic solution procedure, known as the worklist algorithm or the maximal fixpoint algorithm or the fixpoint iteration algorithm
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,...,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) = f_j(in(j))
    if out(j) changed then
        W = W U \{ k | k is successor of j \}
}
Termination Argument

Why does the algorithm terminate?

Sketch of argument:

- in(j), out(j) do not “shrink”: \( in^n(j) \leq in^{n+1}(j) \)
- A node k is added to W only if some out(j) “changes up”: \( out^n(j) < out^{n+1}(j) \)
- Since out(j) in \( \mathbf{L} \), and \( \mathbf{L} \) satisfies the Ascending Chain Condition, out(j) changes at most \( h \) times where \( h \) is the height of the lattice \( \mathbf{L} \)
Correctness Argument

- Theorem: The worklist algorithm computes a solution that satisfies the dataflow equations
Theorem: The algorithm computes the least solution of the dataflow equations. This solution is called the maximal fixpoint solution (MFP) (for historical reasons).

In general, we may have more than one solution to the dataflow equations.

Let $S$ be another solution. We have $\text{MFP} \leq S$ meaning that for every node $j$, if $\text{MFP}(j) = \{\text{in}(j), \text{out}(j)\}$ and $S(j) = \{\text{in}'(j), \text{out}'(j)\}$, then $\text{in}(j) \leq \text{in}'(j)$, $\text{out}(j) \leq \text{out}'(j)$.
Example with \textit{Avail}

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

\begin{align*}
\text{in}(1) &= \emptyset \\
\text{out}(1) &= (\text{in}(1) - E_z) \cup \{x+y\}
\end{align*}

2. \text{if } (z > 500)

\begin{align*}
\text{in}(2) &= \text{out}(1) \vee \text{out}(3) \\
\text{out}(2) &= \text{in}(2)
\end{align*}

3. \text{skip}

\begin{align*}
\text{in}(3) &= \text{out}(2) \\
\text{out}(3) &= \text{in}(3)
\end{align*}

\text{Boils down to: } \text{in}(2) = \{x+y\} \vee \text{in}(2)

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

- Dataflow frameworks, conclusion
- MOP solution vs. MFP solution
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis
Meet Over All Paths (MOP)

Desired dataflow information at node $n$ 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 $f_{n_k}(\ldots f_{n_2}(f_1(\text{init}(1))))$
- The MOP at entry of node $n$ is $V f_{n_k}(\ldots f_{n_2}(f_1(\text{init}(1))))$ 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*
  - Other analysis, such as symbolic execution and axiomatic semantics are more precise and rule out some infeasible paths
- Recall that the **MFP** is the solution computed by the worklist algorithm
MOP vs. MFP

- For *distributive* problems MFP = MOP!

- Unfortunately, for *monotone* problems this is not the case

- But we still have a *safe* solution: it is a *theorem that for monotone problems* MFP ≥ MOP
A safe (also, correct or sound) solution $X$ overestimates the “best” possible dataflow solution, i.e., $X \geq MOP$

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

- In *may problems*, 1 is the universal set of facts, the merge operator is the set union
  - It is **safe** to err by saying that a fact reaches a node when in fact it doesn’t
  - E.g., in *Reach* it is **safe** to err by adding spurious definitions; it is **unsafe** to err by omitting definitions that reach the node
  - In *Reach* a safe solution is a larger set than the MOP solution
Safe Solutions: Reach

\[ U = \text{all definitions:}\{(x,1),(x,4),(a,3)\} \quad \{(x,1),(x,4),(a,3)\} \]

Poset is \(2^U\), \(\leq\) is the subset relation \(\subseteq\)

1. \(x = a \cdot b\)

2. if \(y \leq a \cdot b\)

3. \(a = a + 1\)

4. \(x = a \cdot b\)

5. goto 2

\[
\begin{align*}
\{(x,1),(x,4)\} & \quad \{(x,4),(a,3)\} & \quad \{(x,1),(a,3)\} \\
\{(x,1)\} & \quad \{(x,4)\} & \quad \{(a,3)\} \\
\{} & \quad \{} & \quad \{} \\
\end{align*}
\]
Safe Solutions

- In *must problems* the 1 is the empty set, and the merge operator is set intersection.
  - It is *safe* to err by saying that a fact does not reach a node when in fact it does.
  - E.g., it is *safe* to err by *omitting* an expression that is available; it is *unsafe* to err by *adding* an expression that is unavailable along some path.
- In *Avail*, a safe solution is a smaller set than the MOP solution.
Safe Solutions: \textit{Avail}

\( U = \text{all expressions: } \{a*b,a+1,y*z\} \)

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
Precision of a Dataflow Solution

- Precise solution is one that is “close” to MOP
  - A precise solution contains few spurious dataflow facts (spurious facts is what we call noise)
  - Unfortunately, for most problems even the MOP (an approximation itself!) is undecidable

- $\text{MOP} \leq X \leq Y$: X is more precise than Y
  - Usually, we can compare two solutions X and Y
  - But, for most problems, we have no way of knowing the “ground truth”
Outline of Today’s Class

- Dataflow frameworks, conclusion
- MOP solution vs. MFP solution
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis
Constant Propagation (Simple)

- Problem statement: Does variable \( x \) hold a constant value at a given program point

- Example:

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

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

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

4. \( z=10*x \)
   - in(4): \( x \) is NOT a const!
```
Fit Analysis in Dataflow Framework

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

- 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 $\perp$)</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 $L$, $l_{1} \leq l_{2}$ iff $l_{1x} \leq l_{2x}$ for every variable $x$
  - $l_{1} \lor l_{2}$ amounts to $l_{1x} \lor l_{2x}$ for every variable $x$
  - Merge operator is join of $L$

- Does the product lattice satisfy the ACC?
Product Lattice

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

- E.g.,
  \(<x=1, y=2, z=T> \leq <x=T, y=2, z=T>\)

- E.g.,
  \(<x=1, y=3, z=T> \lor <x=T, y=2, z=T> = <T, T, T>\)
Product Lattice
Constant Propagation: Transfer Functions

- \( j: x = C \)
  \( f_j: \text{kill } x \rightarrow \text{val}, \text{generate } x \rightarrow C \)

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

- $j: x = V_1 \text{ Op } V_2$
  
- $f_j: \text{kill: } x \rightarrow \text{val}$
  
- gen:
  
  If $V_1 \rightarrow c_1$ and $V_2 \rightarrow c_2$ in $\text{in}(j)$, then $x \rightarrow c_1 \text{ Op } c_2$
  
  else if $V_1 \rightarrow T$ or $V_2 \rightarrow T$ in $\text{in}(j)$, then $x \rightarrow T$
  
  else $x \rightarrow \perp$

- 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→1, y→2, z→T>
   3. x=2
      y=1
      out(3): <x→2, y→1, z→T>
   4. z=x+y
      out(4): <x→T, y→T, z→T>
   5. w=10*z
      in(5): <x→T, y→T, z→T>

in(1) is T = <x→T, y→T, z→T>

out(2): <x→1, y→2, z→T>
out(3): <x→2, y→1, z→T>
out(4): <x→T, y→T, z→T>
Not Distributive! A Counter Example

- \( f_4(f_2(f_1(T))) \) 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)
Problem statement: Is integer variable $x$ odd or even at program point $n$? $x \Rightarrow T$, $y \Rightarrow T$

$L_x$:

More Product Lattices

![Diagram showing the flow of the program with decision points and updates]

- $y = 0$
- $x \Rightarrow T$, $y \Rightarrow$ even
- if $(x \geq 10)$
  - $x = x + 1$
  - $y = y + 2$
- $x \Rightarrow T$, $y \Rightarrow$ even
- $\ldots$

Spring 21 CSCI 4450/6450, A Milanova (Example program from MIT OCW Program Analysis)
More Product Lattices

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

- $L_x:$

  $\top$  

  $+$  $0$  $-$  

  $\bot$

E.g., $< x=+, y=T, z=0 >$
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 Analysis: Example

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;
```
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 transform any C-like 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., $V$ 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 \( \ast p = q \) 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

\( \text{in}_{PT}(4) = \text{out}_{PT}(2) \lor \text{out}_{PT}(3) \)
\( \text{out}_{PT}(4) = f_{*p=q}(\text{in}_{PT}(4)) \)
\( \text{in}_{PT}(5) = \text{out}_{PT}(4) \)

MFP
\( \emptyset \)

MOP?
\( \emptyset \)

Spring 21 CSCI 4450/6450, A Milanova
So far and moving on

- **Intra**procedural dataflow analysis
- CFG
- Lattices and transfer functions
- Classical analyses

- **Inter**procedural analysis
- Program analysis frameworks
- Analysis of object-oriented programs