Dataflow Frameworks: Conclusion

Dataflow Analysis: Non-distributive Analysis
Announcements

- Go over Quiz 1
- Homework 1?
Outline of Today’s Class

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

Equations:

$$\text{in}(j) = V \text{out}(i) \quad \text{out}(j) = f_j(\text{in}(j))$$

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 $j$
- $V$ is the merge operator
Dataflow Frameworks (cont.)

- The property space must be:

  1. A lattice $L$, $\leq$
  2. $L$ satisfies the *Ascending Chain Condition*
     Requires that all ascending chains are finite
Dataflow Frameworks (cont.)

- The merge operator $V$ must be the join of $L$
- In dataflow, $L$ is often the lattice of the subsets over a finite set of dataflow facts $D$
  - Choose universal set $D$ (e.g., all definitions)
  - Figure out if we have a a *may* or *must* problem
    - Set ordering operation $\leq$
    - Since the merge operator must be the join of $L$, a *may* problem sets $\leq$ to *subset* and a *must* problem sets $\leq$ to *superset*
Example: *Reach Lattice*

- Property space is the lattice of the subsets where
  - $D$ is the set of all definitions in the program
  - $\leq$ is the *subset* operation
    - *Join* is set union $\cup$, as needed for *Reach*, which is a *may* problem

- Lattice has a 0 being $\emptyset$, and a 1 being $D$
- Does the lattice satisfy the *Ascending Chain Condition*?
Reach Lattice

$D = \text{all definitions:}\{(x,1),(x,4),(a,3)\}$ \quad $\{(x,1),(x,4),(a,3)\}$

Poset is $2^D$, $\leq$ 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

Spring 21 CSCI 4450/6450, A Milanova
(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 $L$
- 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

and

- its transfer function space $F: L \rightarrow L$ is monotone
The transfer functions: \( f: L \to L \). Formally, function space \( F \) is such that

1. \( F \) contains all \( f_j \)
2. \( F \) contains the identity function \( \text{id}(x) = x \)
3. \( F \) is closed under composition
4. Each \( f \) is monotone
Monotonicity Property

- \( F : L \rightarrow L \) is monotone if and only if:
  
  (1) \( a, b \) in \( L \), \( f \) in \( F \) then \( a \leq b \iff f(a) \leq f(b) \)
  
  or (equivalently):
  
  (2) \( x, y \) in \( L \), \( f \) in \( F \) then \( f(x) \lor f(y) \leq f(x \lor y) \)

- Theorem: Definitions (1) and (2) are equivalent.
  
  - Show that (1) implies (2)
  
  - Show that (2) implies (1)
Distributivity Property

- \( F: L \rightarrow L \) is distributive if and only if \( x, y \) in \( L \), \( f \) in \( F \) then \( f(x \lor y) = f(x) \lor f(y) \)

- Every distributive function is also monotone but not the other way around

- Distributivity is a very nice property!
Monotonicity and Distributivity

Is classical \textit{Reach} distributive?

- Yes

To show distributivity:

For each $j$: \[(X \cup Y) \cap \text{pres}(j) \cup \text{gen}(j) = (X \cap \text{pres}(j)) \cup \text{gen}(j) \cup (Y \cap \text{pres}(j)) \cup \text{gen}(j)\]

\[(X \cup Y) \cap \text{pres}(j) \cup \text{gen}(j) = (X \cap \text{pres}(j)) \cup (Y \cap \text{pres}(j)) \cup \text{gen}(j) = (X \cap \text{pres}(j)) \cup \text{gen}(j) \cup (Y \cap \text{pres}(j)) \cup \text{gen}(j)\]
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.
/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue;
inReach(1) = UNDEF

for m = 2 to n do
  in(m) = 0
  inReach(m) = {}
W = {1,2,...,n} /* put every node on the worklist */

while W ≠ Ø do {
  remove j from W
  out(j) = \textit{f}_j(in(j))
  for i in successors(j)
    if out(j) \not\subseteq in(i) then {
      in(i) = out(j) \cup in(i)
      W = W \cup \{ i \}
    }
  outReach(j) = (inReach(j) \cap \text{pres}(j)) \cup \text{gen}(j)
  if outReach(j) \not\subseteq inReach(i)
    inReach(i) = outReach(j) \cup inReach(i)
}


/* 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 \}
}
Example. Reach with Bitvectors

\( (i, 1), (k, 1) \)

\[
\begin{align*}
i &= 0 \\
k &= 0
\end{align*}
\]

\( B1 \)

\( i < 0 \)

\( B2 \)

\( \text{mod}(i, 3) = 0 \)

\( B3 \)

\( \text{exit} \)

\( (k, 4) \)

\( k = k - 1 \)

\( B4 \)

\( (k, 5) \)

\( k = k + 1 \)

\( B5 \)

\( (i, 6) \)

\( i = i + 1 \)

\( B6 \)

\[ \begin{array}{cccccc}
& B1 & B2 & B3 & B4 & B5 & B6 \\
pres: & 00000 & 11111 & 11111 & 10001 & 10001 & 01110 \\
gen: & 11000 & 00000 & 00000 & 00100 & 00010 & 00001 \\
\end{array} \]

Bitvector:

\[
\begin{array}{ccccccc}
& (i, 1) & (k, 1) & (k, 4) & (k, 5) & (i, 6) \\
0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]
Initialization

\[ i=0 \]
\[ k=0 \]
\[ i<0 \mod(i,3) == 0 \]
\[ k=k-1 \]
\[ k=k+1 \]
\[ i=i+1 \]

\[ \text{out(B1)} = 11000 \]
\[ \text{in(B2)} = 00000 \]

\[ \text{pres: } 00000 \ 11111 \ 11111 \ 10001 \ 10001 \ 01110 \]
\[ \text{gen: } 11000 \ 00000 \ 00000 \ 00100 \ 00010 \ 00001 \]

\[ \text{out(B1)} = 11000 \]
\[ \text{out(B6)} = 00001 \]
Iteration

\[
i=0
\]

\[
k=0
\]

\[
i<0
\]

\[
\text{mod}(i,3) == 0
\]

\[
k=k-1
\]

\[
k=k+1
\]

\[
i=i+1
\]

\[
in(B2) = 11001
\]

\[
W = \{ \text{B2, B3, B4, B5, B6} \}
\]

\[
W = \{ \text{B3, B4, B5, B6} \}
\]

\[
W = \{ \text{B4, B5, B6} \}
\]

\[
W = \{ \text{B5, B6} \}
\]

\[
W = \{ \} \quad W = \{ \text{B2} \}
\]
Iteration

\[
\begin{align*}
i &= 0 \\
k &= 0
\end{align*}
\]

\[
\begin{align*}
i < 0 &\quad \mod(i, 3) == 0 \\
k &= k - 1 \\
k &= k + 1 \\
i &= i + 1
\end{align*}
\]

\[
\begin{array}{cccccccc}
B1 & B2 & B3 & B4 & B5 & B6 \\
pres: 00000 & 11111 & 11111 & 10001 & 10001 & 01110 \\
gen: 11000 & 00000 & 00000 & 00100 & 00010 & 00001
\end{array}
\]

\[
\begin{align*}
in(B2) &= 11111 \\
W &= \{ B2 \} \\
W &= \{ B3 \} \\
W &= \{ B4, B5 \} \\
W &= \{ B5, B6 \} \\
W &= \{ B6 \} \\
W &= \{ \}
\end{align*}
\]
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 L, and L satisfies the Ascending Chain Condition, out(j) changes at most h times where h is the height of the lattice L
Correctness Argument

Theorem: The worklist algorithm computes a solution that satisfies the dataflow equations.

Why?

Sketch of argument:

- Assume algorithm terminates and there is a $j$ such that $\text{in}(j) = \emptyset \{ \text{out}(i) \}$ does not hold.
- Thus, there is a path $p$ to $j$ such that $\text{out}(i) \not\subseteq \text{in}(j)$ where $i$ is the predecessor of $j$ in $p$.
- Now assume $p$ is the shortest such path and arrive at contradiction.
Theorem: The algorithm computes the least solution of the dataflow equations.

Historically though, this solution is called the maximal fixpoint solution (MFP)

I.e., For every node $j$, the worklist algorithm computes a solution of the dataflow equations called the $\text{MFP}(j) = \{\text{in}(j), \text{out}(j)\}$. For every other solution we have $\text{in}(j) \leq \text{in}'(j)$, $\text{out}(j) \leq \text{out}'(j)$ for every node $j$
Example

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

2. if \( z > 500 \)

3. skip

\[
\text{Equivalent to: } \text{in}_{\text{Avail}}(2) = \{x+y\} \ \lor \ \text{in}_{\text{Avail}}(2)
\]

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

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

- Non-distributive analyses
  - Constant propagation
  - Points-to analysis (next time)
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 $f_{n_k}(...f_{n_2}(f_1(init(1))))$

- The MOP at entry of $n$ is $V f_{n_k}(...f_{n_2}(f_1(init(1))))$ 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*
  - (Abstract interpretation 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 $\text{MFP} = \text{MOP}$!

- Unfortunately, for *monotone* problems this is not true. But we still have a *safe* solution: it is a theorem that for monotone problems, $\text{MFP} \geq \text{MOP}$.
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

- 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 a spurious definition; it is **unsafe** to err by omitting a definition \((x, k)\) that reaches a node
- **Safe** is “larger” than the MOP: \(\text{MOP} \leq X\). Since \(\leq\) in *Reach* is subset, safer solutions end up being larger sets (which is natural)
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. \(x=a \times b\)
2. if \(y \leq a \times b\)
3. \(a = a + 1\)
4. \(x = a \times b\)
5. goto 2

\{(x,1), (x,4)\} \quad \{(x,4), (a,3)\} \quad \{(x,1), (a,3)\}

\{(x,1)\} \quad \{(x,4)\} \quad \{(a,3)\}

\{\}\n
Spring 21 CSCI 4450/6450, A Milanova
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 saying that an expression is **NOT AVAILABLE** when it may be available; it is unsafe to err by adding an expression that is unavailable along some path.
- **Safe** means “larger” than the MOP under our partial order. In must $\leq$ is superset, “safer” solutions end up being smaller sets.
Safe Solutions: 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

- MOP ≤ X ≤ 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
  - Lattices (last class)
  - Transfer functions
  - Worklist algorithm
- MOP solution vs. MFP solution

- Non-distributive analyses
  - Constant propagation
  - Points-to analysis (next time)
Problem statement: Does variable $x$ hold a constant value at a given program point

Example:

1. $x = 1$
   - in(1): $x$ is not a 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 $L$
  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: 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.,
  \(<x=1, y=3, z=T> \lor <x=T, y=2, z=T> = <T, T, T>\)
Constant Propagation: Transfer Functions

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

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

- \( j: x = V_1 \text{ Op } V_2 \)

- \( f_j: \text{kill: } x \rightarrow \text{val} \)

- \( \text{gen:} \)

  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 \( x \rightarrow \bot \)

- 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→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>

Spring 21 CSCI 4450/6450, A Milanova
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)
More Product Lattices

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

- $L_x$:

```
  T
 odd | even
  ↓   ↓
```

#### Example program from MIT OCW Program Analysis

```
x = x + 1
y = y + 2
if (x >= 10)
  T
  y = 0
  x = x + 1
  y = y + 2
  ...
  F
  x = x + 1
  y = y + 2
  ...
  T
```
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., \(< x=+,y=T,z=0 >\)