Dataflow Analysis: Non-distributive Analyses

Announcements

- I'm submitting paperwork to make Program Analysis comm intensive
- Homework is due Tuesday
  - Lock your team by Sunday at 2pm
  - Bring a copy to class, and submit in Submitty
- Quiz 1 today

Outline of Today's Class

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

Monotone Dataflow Frameworks

- A problem fits into the dataflow framework if
  - its property space is a lattice $L$, ≤ that satisfies the Ascending Chain Condition
  - its merge operator $V$ is the join of $L$
  - its 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

```c
/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue; inReach(1) = UNDEF
W = {1,2,...,n} /* put every node on the worklist */
while W ≠ Ø do {
  remove j from W
  out(j) = f(in(j))
  for i in successors(j)
    if out(j) ≤ in(i) then {
      in(i) = out(j) V in(i)
      inReach(i) = outReach(j) U inReach(i)
      W = W U { i }
    }
}
```

Worklist Algorithm for Forward Dataflow Problems (slightly different)

```c
/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue; out(1) = f(in(1))
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(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:
  At each “phase”, at least one \( \text{out}(j) \) changes.
  Monotonicity of \( f_j \) entails that change of is up the chain: \( \text{out}'(j) \geq \text{out}(j) \).
  Since \( \text{out}(j) \) in \( L \), and \( L \) satisfies the Ascending Chain Condition, \( \text{out}(j) \) changes at most \( O(h) \) 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:
  Whenever \( j \) is processed, algorithm sets \( \text{out}(j) = f_j(\text{in}(j)) \).
  Whenever \( \text{out}(j) \) changes, algorithm puts successors on the list, so \( \text{in}(j) \) = \( V \{ \text{out}(j) \} \).
  So final solution will satisfy equations.

Precision Argument

- 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 \( \text{MFP}(j) = \{ \text{in}(j), \text{out}(j) \} \), such that every other solution \( \{ \text{in}'(j), \text{out}'(j) \} \) of the dataflow equations is \( \text{in}(j) \leq \text{in}'(j), \text{out}(j) \leq \text{out}'(j) \).

Example

\[
\begin{array}{c|c|c}
\text{Solution1} & \text{Solution2} \\
\hline
\emptyset & \emptyset \\
\{x+y\} & \{x+y\} \\
\emptyset & \emptyset \\
\{x+y\} & \emptyset \\
\{x+y\} & \emptyset \\
\end{array}
\]

Equivalent to: \( \text{MFP}(2) = \{x+y\} \) and recall that \( V \) is \( \cap \) (i.e., set intersection).

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 \( p=(1, n_2, n_3, \ldots, n_k) \), we compute \( f_{n_k} \ldots f_{n_2} (f_1 (\text{init}(1))) \)

- The MOP at entry of \( n \) 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

- The MOP is an abstract model for the best solution computable with this kind of static analysis

- It is a common assumption in this kind of static analysis that all program paths are executable
  (Axiomatic semantics frameworks abstract state more precisely, and can reason along paths)

- The MFP is the solution computed by the worklist algorithm
MOP vs. MFP

- For distributive problems we have $\text{MFP} = \text{MOP}$!

- 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 overestimates the “best” possible dataflow solution, i.e., $X \geq \text{MOP}$

- Historically, an acceptable solution 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 set union. It is safe to err by saying that a fact reaches a node when in fact it doesn’t

  - E.g., intuitively, it is safe to err by saying that a definition $(x,k)$ reaches a node, when in fact it MAY NOT REACH that node

  - Safe entails a set “larger” then the MOP under our partial order. Our definition of $\leq$ (which is natural). So “safer” solutions are larger sets

Safe Solutions: Reach

- $U = \{ (x,1), (x,4), (a,3) \}$

- $s$ is the subset relation $\subseteq$

- $1. \text{x=q*b}$

- $2. \text{if y=x*b}$

- $3. \text{a=a+1}$

- $4. \text{x=a*b}$

- $5. \text{goto 3}$

Safe Solutions: Avail

- $U = \{ a*b, a+1, y*z \}$

- $s$ is the superset relation $\supseteq$

- $1. \text{x=a*b}$

- $2. \text{if y*z<=a*b}$

- $3. \text{a=a+1}$

- $4. \text{x=a*b}$

- $5. \text{goto 2}$
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 problem, we have no way of knowing the "ground truth"

Outline of Today's Class

- Dataflow frameworks
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis

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 const
  - out(1): x is 1

- 2. y = z + w
  - x = 2
  - in(2): x is 2
  - 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!
  - out(4): x is 2

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

- 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

- $S := S; S \mid x = V \mid x = V \cdot Op \cdot V$
- $Op := + \cdot | - \cdot | * \cdot | /$
- $V := x \mid y \mid z \mid C$

- $x, y, z$ are program variables
- $C$ is constant
- We have to define transfer functions for $x = V$ and $x = V \cdot Op \cdot V$
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 ( \perp ))</td>
<td>( x ) is unknown</td>
</tr>
</tbody>
</table>

Constant Propagation: Lattice

- Lattice \( L_x \leq \)

- Dataflow lattice \( L \) is the product of \( L_x \)
- \( l_1,l_2 \) in \( L \) if \( l_1 \leq l_2 \) for every variable \( x \)
- \( l_1 \lor l_2 \) amounts to \( l_1 \lor l_2 \) for every variable \( x \)
- Merge operator is join of \( L \)
- Does the product lattice satisfy the ACC?

Constant Propagation: Transfer Functions

\[ j: \begin{align*}
    x &= c \\
    f_j: &\text{ kill } x \to \text{val}, \text{ generate } x \to c \\
    j: x &= y \\
    f_j: &\text{ kill } x \to \text{val}, \text{ add } x \to \text{val'}, \text{ s.t. } y \to \text{val'} \text{ in in}(j), \text{ val and val'} \text{ are one of} \\
    \perp : &\text{ bottom (unknown)} \\
    c: &\text{ constant} \\
    T: &\text{ top (not a constant)}
\end{align*} \]

Example

1. \( \text{if (b>0)} \)
2. \( x=1 \)
3. \( y=2 \)
4. \( z=x+y \)
5. \( w=10*z \)

Not Distributive! A Counter Example

1. \( \text{if (b>0)} \)
2. \( x=1 \)
3. \( y=2 \)
4. \( z=x+y \)
5. \( w=10*z \)

\( f_4(f_2(f_1(T))) \) implies \( z \to 3 \)
\( f_4(f_3(f_1(T))) \) implies \( z \to 3 \)
Thus, MOP at 5
\( f_4(f_2(f_1(T))) \lor f_4(f_3(f_1(T))) \)
implies \( z \to 3 \)

MFP at 5 implies \( z \to \perp \)
(i.e., \( z \) is NOT a const)
Points-to Analysis

- Problem statement: What memory locations may a pointer variable point to?
- Many applications!
  - Enables compiler optimization
  1. a = 1;
  2. *p = b;
  3. s = a * a;
  - Static debugging tools, static taint analysis tools

Example 1:

```c
int a, b;
int *p1, *p2;
p1 = &a;
p2 = p1;
*p2 = 1;
```

Points-to Analysis: Example

Example 1:    Example 2:

```c
int a, b;
int *p1, *p2;
p1 = &a;
p2 = p1;
*p2 = 1;
```

Points-to Analysis: Syntax

- 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

- 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.,
    - \( p3 \)
    - \( V \) is points-to graph union
    - \( 0 \) of \( L \) is empty graph
    - \( 1 \) of \( L \) is complete graph

The Problem with Updates

- Updates (4) \( f_{pq} \) (also known as destructive updates) … are a pain
- If we drop (4) from our language we get
  1. \( p = &a \quad p = \text{cons}(a, ') \)
  2. \( p = q \quad p = q \)
     (actual-to-parameter assignment)
  3. \( p = *q \quad p = \text{car}(q) \)

- Research problems!
**Points-to Analysis is Monotone**

To argue monotonicity we must show that if incoming points-to graph $Pt_1$ is $\leq$ (subset) of another incoming points-to graph $Pt_2$, then $f(Pt_1) \leq f(Pt_2)$ for each $f$

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 if 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. $\text{if (n>0)}$
2. $p=\&a; \quad q=\&b;$
3. $p=\&a; \quad q=\&b;$
4. $p=*q \quad q=\&b;$
5. $p=*q \quad q=\&b;$

$\text{in}_{\text{PT}}(4)$: $\text{out}_{\text{PT}}(3)$

$\text{in}_{\text{PT}}(5)$: $\text{out}_{\text{PT}}(4)$

**Andersen’s Points-to Analysis**

- Commonly attributed to Lars Andersen [1994]
- More approximation than our earlier formulation: gets rid of “kill”, thus, can maintain a single points-to graph
  - Straightforward analysis algorithm
  - Commonly referred to as flow-insensitive, context-insensitive analysis
  - Formulated in terms of subset constraints

**Andersen’s Points-to Analysis**

$\text{pts}(p)$ denotes the points-to set of $p$

1. $p = \&a \quad \{ a \} \subseteq \text{pts}(p)$
2. $p = q \quad \text{pts}(q) \subseteq \text{pts}(p)$
3. $p = *q \quad \text{for each } x \text{ in } \text{pts}(q). \text{pts}(x) \subseteq \text{pts}(p)$
4. $*p = q \quad \text{for each } x \text{ in } \text{pts}(p). \text{pts}(q) \subseteq \text{pts}(x)$

Use worklist-like algorithm to compute least solution of these constraints
Next Class

- Classical Analysis for Object-oriented Programs
  - Call graph construction
  - CHA, RTA, XTA, Points-to analysis

- Introduction to Soot