Dataflow Analysis: Non-distributive Analysis and Approximations

Announcements
- HW1 is due today
- HW2 is out
  - Get started with your Submitty git repos, Soot, and Jimple, etc.
  - On Thursday we’ll talk about
    - Program analysis frameworks and IRs
    - Soot and Ghidra
    - Class analysis (RTA, XTA, etc.)
    - Your homework analyses

Outline of Today’s Class
- MFP solution vs. MOP solution
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis
- Analysis scope and approximations

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 a predecessor of j }
  out(j) = f_j(in(j))
  if out(j) changed then
    W = W U { k | k is a successor of j }
}

Termination Argument
- Theorem: the algorithm terminates
- Sketch of argument:
  At each “phase”, at least one out(j) changes.
  Monotonicity of f_j entails that change is up the chain: out(j) ≥ out(i). Since out(j) in L, and L satisfies the Ascending Chain Condition, 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, algorithms sets out(j) = f_j(in(j)). Whenever out(j) changes, algorithm puts successors on the list, so in(j) = V { out(i) }.
  So final solution will satisfy equations.
**Precision Argument**

Theorem: The algorithm computes the least solution of the dataflow equations.

- The solution computed by the worklist algorithm is called the MFP solution. (Maximal Fixpoint Solution is a historical name)
- Meaning: 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 $\text{in}(j) \leq \text{in}'(j), \text{out}(j) \leq \text{out}'(j)$

**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 $f_{n_k}(...f_{n_2}(f_1(\text{InitValue})))$
- The MOP at entry of $n$ is $\bigvee f_{n_k}(...f_{n_2}(f_1(\text{InitValue})))$ over all paths $p$ from 1 to $n$

**MOP vs. MFP**

- The MOP is the best solution computable with this kind of static analysis
  - It is a common assumption in this kind of static analysis that every path from entry to $n$ is executable
  - Abstract interpretation, axiomatic semantics, and symbolic execution abstract state more precisely, and can rule out some paths
- The MFP is the solution computed by the worklist algorithm

**Safety of a Dataflow Solution**

- A safe (another term: sound) solution $X$ over approximates 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 safe means sets that are larger than the MOP
- It is safe to say that a dataflow fact reaches a node when in fact it does not
- E.g., in Reaching Definitions, it is safe to say that a definition $(x,k)$ reaches a node, when in fact it DOES NOT REACH that node
Safe Solutions: Reach

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

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

1. \( x = a \times b \)
2. if \( y < a \times b \)
3. \( a = a + 1 \)
4. \( x = a \times b \)
5. goto 3

\( (x,1) \) \( (x,4) \) \( (a,3) \)
\( (x,1),(x,4) \) \( (x,1),(a,3) \) \( (x,4),(a,3) \)
\( (x,1),(x,4),(a,3) \)

Spring 20 CSCI 4450/6450, A Milanova

Safe Solutions

- In most problems safe means smaller sets
- It is safe to say that a fact does not reach a node when in fact it does
- E.g., in \( \text{Avail} \), it is safe to say that an expression is NOT AVAILABLE at a node \( n \) when in fact it is available

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 \( \leq X \leq Y \): \( X \) is more precise than \( Y \)
- 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

- MFP solution vs. MOP solution
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis
- Analysis scope and approximations

Constant Propagation (Simple)

- Problem statement: What variables hold constant values across all execution paths

Example:

1. \( x = 1 \)
   - if \( b > 0 \)
   - \( \text{in}(1): x \) is not const
2. \( y = z + w \)
   - \( x = 2 \)
   - \( \text{in}(2): x \) is 1
   - \( \text{out}(2): x \) is 1
3. \( y = 0 \)
   - \( \text{in}(3): x \) is 1
   - \( \text{out}(3): x \) is 1
4. \( z = 10 \times x \)
   - \( \text{out}(4): x \) is NOT a const!
Aside: Defining an Analysis

- Define program syntax
  - In practice, we deal with a lot richer syntax
- Define property space
  - The abstract program state that approximates the concrete program state
- Define transfer function space over syntax
  - Transfer functions add semantics to syntactic constructs
    - E.g., \( x = y \text{ op } z \)

If property space has desired properties

- is a lattice \( L \), \( \preceq \) that satisfies the Ascending Chain Condition
- merge operator \( V \) is the join of \( L \) and
- Function space \( F: L \rightarrow L \) is monotone
then the analysis fits the monotone dataflow framework; we can use the worklist algorithm to compute the MFP solution.

Constant Propagation: Syntax

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

- \( x, y, z \) 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 ( \bot ))</td>
<td>( x ) is unknown</td>
</tr>
</tbody>
</table>

Lattice \( L_x, \leq \)

Dataflow lattice \( L \) is the product lattice of \( L_x \)

- \( 1, l_2 \) in \( L \), \( 1 \leq l_2 \) if \( 1 \leq l_2 \) for each variable \( x \)
- \( 1 \text{ V } l_2 \) amounts to \( 1 \text{ V } l_2 \) for each 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>, \text{ etc.} \>

are lattice elements

E.g.,

\(<x=1, y=2, z=T> \leq <x=T, y=2, z=T>\)

E.g.,

\(<x=1, y=3, z=T> \text{ V } <x=T, y=2, z=T> = <T, T, T>\)
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
- \( \bot \): 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

\[ \begin{align*}
1. & \quad \text{if } (b > 0) \\
2. & \quad x = 1 \\
3. & \quad y = 2 \\
4. & \quad z = x + y \\
5. & \quad w = 10 \cdot z
\end{align*} \]

Constant Propagation is 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 computes \( z \rightarrow 3 \)

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

More Product Lattices

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

\[ \begin{align*}
L_x: & \quad T \\
& \quad \text{odd} \\
& \quad \text{even} \\
0: & \quad \text{even} \\
1: & \quad \text{odd} \\
\end{align*} \]

E.g., \( < x = +, y = T, z = 0 > \)

More Product Lattices

Problem statement: Is integer variable \( x \) odd or even at program point \( n? \)
- \( L_x: \)
  - \( T \rightarrow y \rightarrow T \)
  - \( x \rightarrow T, y \rightarrow \text{even} \)
  - \( x \rightarrow T, y \rightarrow \text{even} \)
  - \( x \rightarrow T, y \rightarrow \text{even} \)

Spring 20 CSCI 4450/6450, A. Milanova
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;
- 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: 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
  - \( \ldots \) or in simpler terms, lattice elements are points-to graphs, e.g., \( p3 \)
  - \( V \) is points-to graph union \( p1 \)
  - \( 0 \) of \( L \) is empty graph \( p2 \)
  - \( 1 \) of \( L \) is complete graph \( a \)

Points-to Analysis: Transfer Functions

(1) \( \text{f}_{\text{p=}&q} \): “kill” all points-to edges from \( p \) to \( q \)
(2) \( \text{f}_{\text{p=q}} \): “kill” all points-to edges from \( p \) to every \( x \), such that \( q \) points to \( x \) in incoming points-to graph in \( j \)
(3) \( \text{f}_{\text{p=*q}} \): “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) \( \text{f}_{\text{p=q}} \): 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 \( P_{t_1} \) is \( \preceq \) (subset of) \( P_{t_2} \), then \( f(P_{t_1}) \preceq f(P_{t_2}) \) for each transfer function \( f \)

(1) \( P_{t_1} \preceq P_{t_2} \) then \( \text{f}_{\text{p=}&q} (P_{t_1}) \preceq \text{f}_{\text{p=}&q} (P_{t_2}) \)
(2) \( P_{t_1} \preceq P_{t_2} \) then \( \text{f}_{\text{p=q}} (P_{t_1}) \preceq \text{f}_{\text{p=q}} (P_{t_2}) \)
(3) \( P_{t_1} \preceq P_{t_2} \) then \( \text{f}_{\text{p=*q}} (P_{t_1}) \preceq \text{f}_{\text{p=*q}} (P_{t_2}) \)
(4) \( P_{t_1} \preceq P_{t_2} \) then \( \text{f}_{\text{p=q}} (P_{t_1}) \preceq \text{f}_{\text{p=q}} (P_{t_2}) \)
... but it is not distributive!

- Because of updates!

**Points-to Analysis is Not Distributive**

```
\[
p = &x; q = &y; p = &z; q = &w;
\]

\[
*p = q
\]

\[
\begin{align*}
\text{Pt}_1 : & \quad \text{in}_{\text{Pt}_1} (4) = \text{out}_{\text{Pt}_1} (3) \\
\text{Pt}_2 : & \quad \text{in}_{\text{Pt}_2} (5) = \text{out}_{\text{Pt}_1} (4)
\end{align*}
\]

What if \( *p = q \) 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**

\[
\begin{align*}
\text{MFP} & : \quad \emptyset \\
\text{MOP?} & : \quad \emptyset
\end{align*}
\]

**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 what is essentially our worklist algorithm
  - Commonly referred to as a flow-insensitive, context-insensitive analysis

**Andersen’s Points-to Analysis: Examples**

**Example 1:**

- \( p_1 = &a \)
- \( p_2 = p_1 \)
- \( *p_2 = 1 \)

**Example 2:**

- \( p_3 = &p_1 \)
- \( p_1 = &a \)
- \( q = p_3 \)
- \( r = *q \)
- \( p_1 = &b \)

**Andersen’s Points-to Analysis**

pts\( (p) \) denotes the points-to set of \( p \)

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

Use worklist-like algorithm to compute least solution of these constraints
Outline of Today’s Class

- MFP solution vs. MOP solution
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis
- Analysis scope and approximations

Analysis Scope

- **Intra-procedural 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
- **Inter-procedural 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 CFGs and solutions 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
- “Declarative” flow-insensitive analysis:
  ```
  S = 0
  do {
    for each node j do
      S = f(S) v S
  } while S changes
  ```
Flow Insensitivity

- “Operational”, worklist-like algorithm:
  
  ```
  S = 0, W = { 1, 2, …, n } /* all nodes */
  while W ≠ Ø do {
    remove j from W
    S = f_j(S) ∪ S
    if S changed then
      W = W ∪ { k | k is “successor” of j }
  }
  ``

- “Successors” here does not refer to successor nodes in CFG, but nodes k whose transfer function f_k may contribute to S as a result of the change at 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

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

```
int 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