Flow-Insensitive Pointer Analysis

Last time
- Interprocedural analysis
- Dimensions of precision (flow- and context-sensitivity)
- Flow-Sensitive Pointer Analysis

Today
- Flow-Insensitive Pointer Analysis

Flow-Insensitive and Context-Insensitive Pointer Analysis

The defining characteristics
- Ignore the control-flow graph, and assume that statements can execute in any order
- Rather than producing a solution for each program point, produce a single solution that is valid for the whole program

Flow-insensitive and Context-Insensitive pointer analyses
- Andersen-style analysis: the slowest and most precise
- Steensgaard analysis: the fastest and least precise
- All other flow-insensitive pointer analyses are hybrids of these two

Andersen-Style Pointer Analysis [1994]

Basic idea
- View pointer assignments as constraints
- Use these constraints to propagate points-to information

Andersen-style Pointer Analysis – Example 1

<table>
<thead>
<tr>
<th>Program</th>
<th>Flow-Sensitive Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a := &amp;b</code></td>
<td><code>a → { b }</code></td>
</tr>
<tr>
<td><code>c := a</code></td>
<td><code>c → { b }</code></td>
</tr>
<tr>
<td><code>a := &amp;d</code></td>
<td><code>a → { d }</code></td>
</tr>
<tr>
<td><code>e := a</code></td>
<td><code>e → { d }</code></td>
</tr>
</tbody>
</table>
### Andersen-style Pointer Analysis – Example 1

<table>
<thead>
<tr>
<th>Program</th>
<th>Constraints</th>
<th>Points-to Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a := &amp;b</code></td>
<td><code>a ⊇ { b, d }</code></td>
<td><code>a ↦ { b, d }</code></td>
</tr>
<tr>
<td><code>c := a</code></td>
<td><code>c ⊇ a</code></td>
<td><code>c ↦ { b, d }</code></td>
</tr>
<tr>
<td><code>a := &amp;d</code></td>
<td><code>e ⊇ a</code></td>
<td><code>e ↦ { b, d }</code></td>
</tr>
<tr>
<td><code>e := a</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We’ve reached a fixed point

**Terminology**

- **Base constraints**: Used to initialize the points-to sets
  Ex: `a := &b`
  Not needed after initialization
- **Simple constraints**: Involve variable names only
  Ex: `c := a`
- **Complex constraints**: Involve pointer dereferences
  Ex: `*a := c`

### Andersen-style Pointer Analysis – Example 2

<table>
<thead>
<tr>
<th>Program</th>
<th>Constraints</th>
<th>Points-to Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a := &amp;b</code></td>
<td><code>a ⊇ { b }</code></td>
<td><code>a ↦ { b, d }</code></td>
</tr>
<tr>
<td><code>c := &amp;d</code></td>
<td><code>c ⊇ { d }</code></td>
<td><code>c ↦ { d }</code></td>
</tr>
<tr>
<td><code>e := &amp;a</code></td>
<td><code>e ⊇ { a }</code></td>
<td><code>e ↦ { a }</code></td>
</tr>
<tr>
<td><code>f := a</code></td>
<td><code>f ⊇ a</code></td>
<td><code>f ↦ { b, d }</code></td>
</tr>
<tr>
<td><code>*e := c</code></td>
<td><code>*e ⊇ c</code></td>
<td><code>a ⊇ c</code></td>
</tr>
</tbody>
</table>

Notice that we create the constraint graph dynamically

---

### Andersen-Style Pointer Analysis

**Basic idea**

- View pointer assignments using a constraint graph
- Propagate points-to relations along the edges of the constraint graph, adding new edges as indirect constraints are resolved

**Constraint graph**

- One node for each variable
- One directed edge for each constraint

**Andersen-style analysis**

- Can be reduced to computing the transitive closure of a dynamic graph
- A well-studied problem for which the best known complexity is $O(n^3)$
Andersen-style Pointer Analysis – The Constraint Graph

Example 2

\[
\begin{array}{cccc}
& f & a & c & \{e \supseteq c\} \\
\downarrow & \downarrow & \downarrow & \downarrow & \\
\{b, d\} & \{b, d\} & \{d\} & \{a\}
\end{array}
\]

Andersen-style Pointer Analysis – Procedure Calls

Program

\[
\text{foo}(\text{int}* x)\{ \\
\quad \ldots \\
\quad \text{return } x; \\
\} \\
\]

Constraints

\[
\begin{align*}
& x \supseteq b \\
& a \supseteq x \\
& a := \text{foo}(&b)
\end{align*}
\]

How do we handle procedure calls?
- Insert constraints for copying actual parameters to formal parameters
- Insert constraints for copying return values

Steensgaard Pointer Analysis

Basic idea
- Further reduce precision by using equality constraints
- That is, information flows both ways, rather than from the right-hand side to the left-hand side of the constraint.

Tradeoffs
- Extremely imprecise
- A system of equality constraints can be solved in near-linear time
- Running time is \(O(n \cdot \alpha(n))\), where \(\alpha(n)\) is the inverse Ackermann's function.
- \(\alpha(2^{132}) < 4\)

Key idea
- The key to this algorithm is the UNION-FIND data structure.

Steensgaard Pointer Analysis – UNION-FIND

The UNION-FIND data structure
- Maintains a set of disjoint sets and supports two operations:
  - FIND(x) : return the set containing x.
  - UNION(x,y) : union the two sets containing x and y.

Set Representation
- Sets are represented by a distinguished element called the set representative
- Each set is an inverted tree, with nodes pointing to their parents and the set representative as the root.
Steensgaard Pointer Analysis – UNION-FIND

UNION (a, b)
- FIND(a)
- FIND(b)

\[ \begin{array}{cccc}
  a & b & c & d \\
\end{array} \]

Steensgaard Pointer Analysis – UNION-FIND

UNION (a, c)
- FIND(a)
- FIND(c)

\[ \begin{array}{cccc}
  a & b & c & d \\
\end{array} \]

Steensgaard Pointer Analysis – UNION-FIND

UNION (a, d)
- FIND(a)
- FIND(d)

\[ \begin{array}{cccc}
  a & b & c & d \\
\end{array} \]

Steensgaard Pointer Analysis – the Algorithm

merge(x, y)
{
  x = FIND(x); y = FIND(y);
  if (x == y) then return;
  UNION(x,y);
  merge(points-to(x),points-to(y));
}

for each constraint LHS = RHS
  merge(LHS,RHS)
### Steensgaard Pointer Analysis – Example 1

**Program**
- `a := &b`
- `c := a`
- `a := &d`
- `e := a`

**Constraints**
- `a = { b, d }`
- `c = a`
- `e = a`

**Points-to Relations**
- `a, c, e` --> `b, d`

### Steensgaard Pointer Analysis – Example 2

**Program**
- `a := &b`
- `c := &d`
- `e := &a`
- `f := a`
- `*e := c`

**Constraints**
- `a = { b }`
- `c = { d }`
- `e = { a }`
- `f = a`
- `*e = c`

**Points-to Relations**
- `a, f, c`
- `e`
- `b, d` and `d`

### Andersen vs. Steensgaard

**Andersen-style analysis**
- `int *a, *b, c, *d, e;`
- `1: a = &b;`
- `2: b = &c;`
- `3: d = &e;`
- `4: a = &d;`

Andersen-style analysis
- Due to statement 4

**Steensgaard analysis**
- Due to statement 4

### Concepts

**Flow-insensitive pointer analysis**

**Andersen-style analysis**
- Inclusion-based, subset-based
- Compute transitive closure of a dynamic graph
- Constraint graph
- Cycle elimination optimization

**Steensgaard-style analysis**
- Unification-based, equality-based
- Union-find data structure
Next Time

Lecture

– Context-Sensitive Pointer Analysis