Flow-Insensitive Pointer Analysis

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

Today
- Flow-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 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>a := &amp;b</td>
<td>a \rightarrow { b }</td>
</tr>
<tr>
<td>c := a</td>
<td>c \rightarrow { b }</td>
</tr>
<tr>
<td>a := &amp;d</td>
<td>a \rightarrow { d }</td>
</tr>
<tr>
<td>e := a</td>
<td>e \rightarrow { d }</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>a := &amp;b</td>
<td>a ⊇ { b, d }</td>
<td>a → { b, d }</td>
</tr>
<tr>
<td>c := a</td>
<td>c ⊇ a</td>
<td>c → { b, d }</td>
</tr>
<tr>
<td>a := &amp;d</td>
<td>e ⊇ a</td>
<td>e → { b, d }</td>
</tr>
<tr>
<td>e := a</td>
<td></td>
<td>We’ve reached a fixed point</td>
</tr>
</tbody>
</table>

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>a := &amp;b</td>
<td>a ⊇ { b }</td>
<td>a → { b, d }</td>
</tr>
<tr>
<td>c := &amp;d</td>
<td>c ⊇ { d }</td>
<td>c → { d }</td>
</tr>
<tr>
<td>e := &amp;a</td>
<td>e ⊇ { a }</td>
<td>e → { a }</td>
</tr>
<tr>
<td>f := a</td>
<td>f ⊇ a</td>
<td>f → { b, d }</td>
</tr>
<tr>
<td>*e := c</td>
<td>*e ⊇ c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a ⊇ c</td>
<td></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

```
Example 2

Andersen-style Pointer Analysis – Cycle Elimination

Cycle Elimination
- The most important optimization for Andersen-style analysis
- Detect strongly-connected components in the constraint graph
- Collapse them into a single node

The rationale
- All nodes in the same SCC are guaranteed to have the same points-to relations at the end of the analysis

Complication
- Most SCCs are created dynamically during the analysis
- Cycle elimination must be performed dynamically for greatest effect
```
Andersen-style Pointer Analysis – Cycle Elimination

\[
\begin{array}{c}
\text{e} \\
\text{b} \\
\text{c} \\
\text{d} \\
\text{a}
\end{array}
\]

\[
\begin{array}{c}
\{w,x,y,z\} \\
\{w,x,y,z\} \\
\{w,x,y,z\} \\
\{w,x,y,z\} \\
\{w,x,y,z\}
\end{array}
\]
**Andersen-style Pointer Analysis – Procedure Calls**

<table>
<thead>
<tr>
<th>Program</th>
<th>Constraints</th>
</tr>
</thead>
</table>
| `foo(int* x){
  . . .
  return x;
}` | `x ⊇ { b }
  a ⊇ x` |
| `a := foo(&b)` | |

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**

- 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.
- $\alpha(2^{12}) < 4$

**Key idea**

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

UNION (a, b)

- FIND(a)
- FIND(b)

```
    a ——— b
    
    c
    
    d
```
Steensgaard Pointer Analysis – UNION-FIND

UNION \((a, c)\)
- \(\text{FIND}(a)\)
- \(\text{FIND}(c)\)

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

Steensgaard Pointer Analysis – UNION-FIND

UNION \((a, d)\)
- \(\text{FIND}(a)\)
- \(\text{FIND}(d)\)

\[ \begin{array}{c}
    a \rightarrow b \rightarrow c \rightarrow d \\
\end{array} \]
**UNION-FIND Optimizations**

Two key optimizations
- Path compression
- Union-by-rank
- Together these optimizations yield near-linear time operations

**Path compression**
- Avoid redundant searches for the set representative

**Union-by-rank**
- When performing the UNION operation, choose the set representative based on the sizes of the two sets

---

**Steensgaard Pointer Analysis – Path Compression**

\[ \text{UNION} \ (a, \ b) \]
- \( \text{FIND}(a) \)
- \( \text{FIND}(b) \)

![Diagram showing a to b, c, d]
Steensgaard Pointer Analysis – Path Compression

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

![Diagram](image1)

Steensgaard Pointer Analysis – Path Compression

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

![Diagram](image2)
Steensgaard Pointer Analysis – Union-by-Rank

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

\[ \begin{array}{cccc}
  a & b & c & d \\
\end{array} \]
Steensgaard Pointer Analysis – Union-by-Rank

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

What is the benefit of union-by-rank?
- It ensures that we follow as few parent pointers as possible
- Consider the cost of selecting d as the new set representative in this last union operation

Steensgaard Pointer Analysis – the Algorithm

```c
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

<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><img src="image1.png" alt="Diagram" /></td>
</tr>
<tr>
<td><code>c := a</code></td>
<td><code>c = a</code></td>
<td></td>
</tr>
<tr>
<td><code>a := &amp;d</code></td>
<td><code>e = a</code></td>
<td></td>
</tr>
<tr>
<td><code>e := a</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Steensgaard 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><img src="image2.png" alt="Diagram" /></td>
</tr>
<tr>
<td><code>c := &amp;d</code></td>
<td><code>c = { d }</code></td>
<td></td>
</tr>
<tr>
<td><code>e := &amp;a</code></td>
<td><code>e = { a }</code></td>
<td></td>
</tr>
<tr>
<td><code>f := a</code></td>
<td><code>f = a</code></td>
<td></td>
</tr>
<tr>
<td><code>*e := c</code></td>
<td><code>*e = c</code></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image2.png)
Andersen vs. Steensgaard

Andersen-style analysis

Steensgaard analysis

due to statement 4

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