April 23 — Pointer Analysis

Rutgers University
Pointer Analysis

• What is it?

Memory locations a pointer expression can refer to

Example:

```c
int x;
p = &x;
q = p;
```

*p and *q alias, as do x and *p, x and *q
Why pointer analysis?

• Fundamental in program analysis, bug-finding, program understanding, optimization and etc

• Tells us what memory locations code uses or modifies

• Examples:

  • Redundant expression elimination:

    \[ *p = a+b; \ y = a+b \]

  • Constant propagation: \( x = 3; \ *p = 4; \ y = x \)
Flow Sensitivity

- **Flow-sensitive**: pointer analysis computes for each program point what memory locations pointer expressions may refer to.

- **Flow-insensitive**: pointer analysis computes what memory locations pointer expression may refer to, at any time in program execution.

- Example:
May vs Must

• May analysis: aliasing that may occur during execution

• Must analysis: aliasing that must occur during execution

• Liveness analysis example:

  • \*p = \*q + 4;
Representation

- **Points-to pairs**: first elements points to the second
  - e.g., \((p \rightarrow b)\), \(*p\) and \(b\) alias

- **Pairs** that refer to the same memory location
  - e.g., \((*p,b), (*q,b), (*p,*q), (**r,b)\)

- **Equivalence sets**: sets that are aliases
  - e.g., \({*p, *q, b}\)
Memory Location Abstraction

• Describe memory location(s) a pointer expression may refer to

• Map concrete location(s) to “abstract locations/nodes”
  • One abstract node may represent one or more concrete memory locations

• Analysis clients need to know about this abstraction
  • May be difficult to compare different abstractions
Heap Abstraction

• One abstract node for every allocation site
  • One global variable per malloc
  • e.g., \texttt{12: x = malloc(...), x \rightarrow m12}

• Alternatives
  • One node for the entire heap
  • Different nodes for locations allocated in different calling contexts. Aka “context-sensitive”
Records & Structures

• Model each field of each structure
  
  • a.k.a. “field-sensistive”

  • `struct { int a, b; } x, y;` → `x.a, x.b, y.a, y.b`

• Or: Merge all fields of each structure variable
  
  • a.k.a., “field-independent”

  • `struct { int a, b; } x, y;` → `x.*, y.*`

• Or: Model each field of all structure variables
  
  • a.k.a, “field-based”

  • `struct { int a, b; } x, y;` → `*.a, *.b`
Arrays

• Merge all array elements together
  • int a[10]; —> a[*]

• Or use a separate abstraction for the first element
  • int a[10]; —> a[0] & a[1..9]
Flow-insensitive May Pointer Analysis

• Assume program consists of statements of form
  
  • \( p = &a \) (address of, includes allocation statements)
  
  • \( p = q \) (copy)
  
  • \( *p = q \) (store)
  
  • \( p = *q \) (load)
  
• Assume pointers \( p, q \) and address-taken variables \( a, b \) are disjoint
  
  • Can transform program to make this true
  
  • For any variable \( v \) for which this isn’t true, add statement \( p_v = &a_v \), and replace every occurrence of \( v \) with \( *p_v \)
Andersen-style Pointer Analysis

- View pointer assignments as subset constraints
- Use constraints to propagate points-to information

<table>
<thead>
<tr>
<th>Constraint type</th>
<th>Assignment</th>
<th>Constraint</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td><code>a = &amp;b</code></td>
<td><code>a ⊇ {b}</code></td>
<td><code>loc(b) ∈ pts(a)</code></td>
</tr>
<tr>
<td>Simple</td>
<td><code>a = b</code></td>
<td><code>a ⊇ b</code></td>
<td><code>pts(a) ⊇ pts(b)</code></td>
</tr>
<tr>
<td>Complex</td>
<td><code>a = *b</code></td>
<td><code>a ⊇ *b</code></td>
<td><code>∀v ∈ pts(b). pts(a) ⊇ pts(v)</code></td>
</tr>
<tr>
<td>Complex</td>
<td><code>*a = b</code></td>
<td><code>*a ⊇ b</code></td>
<td><code>∀v ∈ pts(a). pts(v) ⊇ pts(b)</code></td>
</tr>
</tbody>
</table>
Example

- Solve for sets \( \text{pts}(p) \)

\[
\begin{align*}
    p &= \&a; & p \supseteq \{a\} & \text{points}(p) = \{\} \\
    q &= p; & q \supseteq p & \text{points}(q) = \{\} \\
    p &= \&b; & p \supseteq \{b\} & \text{points}(r) = \{\} \\
    r &= p; & r \supseteq p & \text{points}(a) = \{\} \\
    & & & \text{points}(b) = \{\}
\end{align*}
\]
Another example

\[
p = \&a \\
q = \&b \\
*p = q; \\
r = \&c; \\
s = p; \\
t = *p; \\
*s = r;
\]

retrieved at the second iteration
Another example

\[ p = &a \]

\[ q = &b \]

\[ *p = q; \]

\[ r = &c; \]

\[ s = p; \]

\[ t = *p; \]

\[ *s = r; \]

retrieved at the second iteration
Anderson Style as Graph Closure

- Can also be modeled as graph closure problem
- One node for each pts(p), pts(a)

<table>
<thead>
<tr>
<th>Assgmt.</th>
<th>Constraint</th>
<th>Meaning</th>
<th>Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = &amp;b</td>
<td>a ⊇ {b}</td>
<td>b ∈ pts(a)</td>
<td>no edge</td>
</tr>
<tr>
<td>a = b</td>
<td>a ⊇ b</td>
<td>pts(a) ⊇ pts(b)</td>
<td>b → a</td>
</tr>
<tr>
<td>a = *b</td>
<td>a ⊇ *b</td>
<td>∀v ∈ pts(b), pts(a) ⊇ pts(v)</td>
<td>no edge</td>
</tr>
<tr>
<td>*a = b</td>
<td>*a ⊇ b</td>
<td>∀v ∈ pts(a), pts(v) ⊇ pts(b)</td>
<td>no edge</td>
</tr>
</tbody>
</table>

- Each node has an associated points-to set
- Compute transitive closure of graph, and adages according to the **subset constraints**
Fixed-point Workqueue Algorithm

- Initialize graph and points to sets using base and simple constraints
- Let $W = \{ v \mid \text{pts}(v) \neq \emptyset \}$ (all nodes with non-empty points to sets)
- While $W$ not empty
  - $v \leftarrow$ select from $W$
  - for each $a \in \text{pts}(v)$ do
    - for each constraint $p \supseteq v$
      - add edge $a \rightarrow p$, and add $a$ to $W$ if edge is new
    - for each constraint $*v \supseteq q$
      - add edge $q \rightarrow a$, and add $q$ to $W$ if edge is new
  - for each edge $v \rightarrow q$ do
    - $\text{pts}(q) = \text{pts}(q) \cup \text{pts}(v)$, and add $q$ to $W$ if $\text{pts}(q)$ changed
Same Example Modeled as Graph Closure Problem

\[
\begin{align*}
p &= \&a \\
q &= \&b \\
*p &= q; \\
r &= \&c; \\
s &= p; \\
t &= *p; \\
*s &= r;
\end{align*}
\]

\[
\begin{align*}
p \supseteq \{a\} \\
q \supseteq \{b\} \\
*p \supseteq q \\
r \supseteq \{c\} \\
s \supseteq p \\
t \supseteq *p \\
*s \supseteq r
\end{align*}
\]
Same Example Modeled as Graph Closure Problem

\[
\begin{align*}
p &= \&a \\
q &= \&b \\
*p &= q; \\
r &= \&c; \\
s &= p; \\
t &= \*p; \\
*s &= r;
\end{align*}
\]
Complexity

- Andersen-style pointer analysis is $O(n^3)$ for number of nodes in the graph
- Identify strongly connected component
  - Collapse strongly connected component into single node
  - Reduces $O(n^3)$ by reducing $n$
- Why? All nodes in an SCC will have the same points-to relation at the end
Steensgaard-style Analysis

- Also a constraint-based analysis

- Uses **equality constraints** instead of subset constraints
  - Collapse sets that are “equal”

- Less precise than Andersen-style, but more scalable

<table>
<thead>
<tr>
<th>Constraint type</th>
<th>Assignment</th>
<th>Constraint</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>a = &amp;b</td>
<td>a ≥ {b}</td>
<td>loc(b) ∈ pts(a)</td>
</tr>
<tr>
<td>Simple</td>
<td>a = b</td>
<td>a = b</td>
<td>pts(a) = pts(b)</td>
</tr>
<tr>
<td>Complex</td>
<td>a = *b</td>
<td>a = *b</td>
<td>∀v ∈ pts(b). pts(a) = pts(v)</td>
</tr>
<tr>
<td>Complex</td>
<td>*a = b</td>
<td>*a = b</td>
<td>∀v ∈ pts(a). pts(v) = pts(b)</td>
</tr>
</tbody>
</table>
Example

\[ x = \ast y \]

```
x -> z -> a
y -> w -> v -> b
```
Example
Example
Example
Steensgaard-style analysis results

- Can be efficiently implemented as union-find algorithm
  - Every node has at most one outgoing edge
  - Each statement needs to be processed just once
  - Has been shown to analyze programs with millions of lines of code in under a minute

<table>
<thead>
<tr>
<th>Name</th>
<th>Size (LoC)</th>
<th>Andersen(sec)</th>
<th>Steensgaard(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>triangle</td>
<td>1986</td>
<td>2.9</td>
<td>0.8</td>
</tr>
<tr>
<td>gzip</td>
<td>4584</td>
<td>1.7</td>
<td>1.1</td>
</tr>
<tr>
<td>li</td>
<td>6054</td>
<td>738.5</td>
<td>4.7</td>
</tr>
<tr>
<td>bc</td>
<td>6745</td>
<td>5.5</td>
<td>1.6</td>
</tr>
<tr>
<td>less</td>
<td>12152</td>
<td>1.9</td>
<td>1.5</td>
</tr>
<tr>
<td>make</td>
<td>15564</td>
<td>260.8</td>
<td>6.1</td>
</tr>
<tr>
<td>tar</td>
<td>18585</td>
<td>23.2</td>
<td>3.6</td>
</tr>
<tr>
<td>espresso</td>
<td>22050</td>
<td>1373.6</td>
<td>10.2</td>
</tr>
<tr>
<td>screen</td>
<td>24300</td>
<td>514.5</td>
<td>10.1</td>
</tr>
</tbody>
</table>

75MHz SuperSPARC, 256MB RAM

[Shapiro-Horwitz POPL’97]
Closing Remarks

• Pointer analysis: important and challenging
  • Can be used for live-variable analysis, constant propagation, call graph construction, bug-finding
  • Inclusion-based analysis (andersen-style)
  • Equality-Based analysis (steensgaard-style)
• Tradeoff between precision and efficiency
  • Based application and program analysis type
• The right evaluation metric is an ongoing debate
  • Size of points-to sets
  • Disamguated virtual calls? False data dependences removed? How much faster? False positive ratio in a bug-finding tool?