General Data Flow Frameworks

Uday P. Khedker

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

May 2011
Part 1

About These Slides
Copyright

These slides constitute the lecture notes for

- MACS L111 Advanced Data Flow Analysis course at Cambridge University, and
- CS 618 Program Analysis course at IIT Bombay.

They have been made available under GNU FDL v1.2 or later (purely for academic or research use) as teaching material accompanying the book:


Apart from the above book, some slides are based on the material from the following books

Outline

• Modelling General Flows
• Constant Propagation
• Faint Variables Analysis
• Pointer Analyses
• Heap Reference Analysis

Important Note:
• Focus on intuitions conveyed through examples rather than formal definitions
Part 2

Modelling General Flows
Part 3

Precise Modelling of General Flows
Complexity of Constant Propagation?

1

2

\[ a = b + 1 \]

3

\[ b = c + 1 \]

4

\[ c = d + 1 \]

5

\[ d = 2 \]
Complexity of Constant Propagation?

Iteration #1

1

2

3

4

5

1

2

3

4

5

\[ a = b + 1 \]

\[ b = c + 1 \]

\[ c = d + 1 \]

\[ d = 2 \]

\[ a = b + 1 \]

\[ b = c + 1 \]

\[ c = d + 1 \]

\[ d = 2 \]
Complexity of Constant Propagation?

Iteration #1

1. $a = b + 1$
2. $b = c + 1$
3. $c = d + 1$
4. $d = 2$

Iteration #2

1. $a = b + 1$
2. $b = c + 1$
3. $c = d + 1$
4. $c = 3$
5. $d = 2$
Complexity of Constant Propagation?

Iteration #1
1. $a = b + 1$
2. $b = c + 1$
3. $c = d + 1$
4. $d = 2$

Iteration #2
1. $a = b + 1$
2. $b = c + 1$
3. $c = 3$
4. $d = 2$

Iteration #3
1. $a = b + 1$
2. $b = 4$
3. $c = 3$
4. $d = 2$
Complexity of Constant Propagation?

\begin{align*}
1. & \quad a = b + 1 \\
2. & \quad b = c + 1 \\
3. & \quad c = d + 1 \\
4. & \quad d = 2 \\
\end{align*}

\begin{align*}
\text{Iteration \#1} \\
1. & \quad a = b + 1 \\
2. & \quad b = c + 1 \\
3. & \quad c = d + 1 \\
4. & \quad d = 2 \\
\end{align*}

\begin{align*}
\text{Iteration \#2} \\
1. & \quad a = b + 1 \\
2. & \quad b = c + 1 \\
3. & \quad c = 3 \\
4. & \quad d = 2 \\
\end{align*}

\begin{align*}
\text{Iteration \#3} \\
1. & \quad a = b + 1 \\
2. & \quad b = 4 \\
3. & \quad c = 3 \\
4. & \quad d = 2 \\
\end{align*}

\begin{align*}
\text{Iteration \#4} \\
1. & \quad a = 5 \\
2. & \quad b = 3 \\
3. & \quad c = d + 1 \\
4. & \quad d = 2 \\
\end{align*}
## Loop Closures of Flow Functions

![Diagram](image)

<table>
<thead>
<tr>
<th>Paths Terminating at $p_2$</th>
<th>Data Flow Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1, p_2$</td>
<td>$x$</td>
</tr>
<tr>
<td>$p_1, p_2, p_3, p_2$</td>
<td>$f(x)$</td>
</tr>
<tr>
<td>$p_1, p_2, p_3, p_2, p_3, p_2$</td>
<td>$f(f(x)) = f^2(x)$</td>
</tr>
<tr>
<td>$p_1, p_2, p_3, p_2, p_3, p_2, p_3, p_2$</td>
<td>$f(f(f(x))) = f^3(x)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Loop Closures of Flow Functions

Paths Terminating at $p_2$ | Data Flow Value
---|---
$p_1, p_2$ | $x$
$p_1, p_2, p_3, p_2$ | $f(x)$
$p_1, p_2, p_3, p_2, p_3, p_2$ | $f(f(x)) = f^2(x)$
$p_1, p_2, p_3, p_2, p_3, p_2, p_3, p_2$ | $f(f(f(x))) = f^3(x)$
... | ...

- For static analysis we need to summarize the value at $p_2$ by a value which is safe after any iteration.

$$f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap f^4(x) \sqcap \ldots$$
Loop Closures of Flow Functions

Paths Terminating at $p_2$ | Data Flow Value
---|---
$p_1, p_2$ | $x$
$p_1, p_2, p_3, p_2$ | $f(x)$
$p_1, p_2, p_3, p_2, p_3, p_2$ | $f(f(x)) = f^2(x)$
$p_1, p_2, p_3, p_2, p_3, p_2, p_3, p_2$ | $f(f(f(x))) = f^3(x)$
... | ...

- For static analysis we need to summarize the value at $p_2$ by a value which is safe after any iteration.

$$f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap f^4(x) \sqcap \ldots$$

- $f^*$ is called the loop closure of $f$. 
Loop Closures in Bit Vector Frameworks

- Flow functions in bit vector frameworks have constant Gen and Kill

\[
\begin{align*}
  f^*(x) &= x \cap f(x) \cap f^2(x) \cap f^3(x) \cap \ldots \\
  f^2(x) &= f(Gen \cup (x - Kill)) \\
         &= Gen \cup ((Gen \cup (x - Kill)) - Kill) \\
         &= Gen \cup ((Gen - Kill) \cup (x - Kill)) \\
         &= Gen \cup (Gen - Kill) \cup (x - Kill) \\
         &= Gen \cup (x - Kill) = f(x) \\
  f^*(x) &= x \cap f(x)
\end{align*}
\]
Loop Closures in Bit Vector Frameworks

• Flow functions in bit vector frameworks have constant Gen and Kill

\[
\begin{align*}
  f^*(x) & = x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap \ldots \\
  f^2(x) & = f (\text{Gen} \cup (x - \text{Kill})) \\
  & = \text{Gen} \cup ((\text{Gen} \cup (x - \text{Kill})) - \text{Kill}) \\
  & = \text{Gen} \cup ((\text{Gen} - \text{Kill}) \cup (x - \text{Kill})) \\
  & = \text{Gen} \cup (\text{Gen} - \text{Kill}) \cup (x - \text{Kill}) \\
  & = \text{Gen} \cup (x - \text{Kill}) = f(x) \\
  f^*(x) & = x \sqcap f(x)
\end{align*}
\]

• *Loop Closures of Bit Vector Frameworks are 2-bounded.*
Loop Closures in Bit Vector Frameworks

- Flow functions in bit vector frameworks have constant Gen and Kill

\[ f^*(x) = x \cap f(x) \cap f^2(x) \cap f^3(x) \cap \ldots \]
\[ f^2(x) = f(Gen \cup (x - Kill)) \]
\[ = Gen \cup ((Gen \cup (x - Kill)) - Kill) \]
\[ = Gen \cup ((Gen - Kill) \cup (x - Kill)) \]
\[ = Gen \cup (Gen - Kill) \cup (x - Kill) \]
\[ = Gen \cup (x - Kill) = f(x) \]
\[ f^*(x) = x \cap f(x) \]

- Loop Closures of Bit Vector Frameworks are 2-bounded.

- Intuition: Since Gen and Kill are constant, same things are generated or killed in every application of \( f \).
  Multiple applications of \( f \) are not required unless the input value changes.
Larger Values of Loop Closure Bounds

- Fast Frameworks $\equiv$ 2-bounded frameworks (e.g. bit vector frameworks)
  Both these conditions must be satisfied
  - *Separability*
    Data flow values of different entities are independent
  - *Constant or Identity Flow Functions*
    Flow functions for an entity are either constant or identity

- Non-fast frameworks
  At least one of the above conditions is violated
Separability

\[ f : L \mapsto L \text{ is } \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \text{ where } \hat{h}_i \text{ computes the value of } \hat{x}_i \]
Separability

\[ f : L \leftrightarrow L \text{ is } \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \text{ where } \hat{h}_i \text{ computes the value of } \hat{x}_i \]

Separable

Non-Separable

Example: All bit vector frameworks
Example: Constant Propagation
Separability

\[ f : L \mapsto L \text{ is } \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \text{ where } \hat{h}_i \text{ computes the value of } \hat{x}_i \]

**Separable**

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]
\[ \downarrow \]
\[ f \]
\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]

**Non-Separable**

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]
\[ \downarrow \]
\[ f \]
\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]

Example: All bit vector frameworks
Example: Constant Propagation
Separability

\( f : L \leftrightarrow L \) is \( \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \) where \( \hat{h}_i \) computes the value of \( \hat{x}_i \)

**Separable**

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]

\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]

**Non-Separable**

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]

\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]

Example: All bit vector frameworks

Example: Constant Propagation
Separability

\[ f : L \mapsto \hat{L} \text{ is } \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \text{ where } \hat{h}_i \text{ computes the value of } \hat{x}_i \]

**Separable**

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]

\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]

\[ \hat{h} : \hat{L} \mapsto \hat{L} \]

**Non-Separable**

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]

\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]

Example: All bit vector frameworks

Example: Constant Propagation

May 2011

Uday Khedker
Separability

\[ f : L \leftrightarrow L \text{ is } \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \text{ where } \hat{h}_i \text{ computes the value of } \hat{x}_i \]

Separable

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]

\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]

\[ \hat{h} : \hat{L} \leftrightarrow \hat{L} \]

Example: All bit vector frameworks

Non-Separable

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]

\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]

Example: Constant Propagation
Separability

\[ f : L \leftrightarrow L \text{ is } \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \text{ where } \hat{h}_i \text{ computes the value of } \hat{x}_i \]

Separable

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]

\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]

\[ \hat{h} : \hat{L} \leftrightarrow \hat{L} \]

Example: All bit vector frameworks

Non-Separable

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]

\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]

\[ \hat{h} : \hat{L} \leftrightarrow \hat{L} \]

Example: Constant Propagation
Separability of Bit Vector Frameworks

- \( \hat{L} \) is \( \{0, 1\} \), \( L \) is \( \{0, 1\}^m \)
- \( \hat{\cap} \) is either boolean AND or boolean OR
- \( \hat{\top} \) and \( \hat{\bot} \) are 0 or 1 depending on \( \hat{\cap} \).
- \( \hat{h} \) is a *bit function* and could be one of the following:

<table>
<thead>
<tr>
<th>Raise</th>
<th>Lower</th>
<th>Propagate</th>
<th>Negate</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
<td><img src="image3" alt="Diagram" /></td>
<td><img src="image4" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Separability of Bit Vector Frameworks

- $\hat{L}$ is $\{0, 1\}$, $L$ is $\{0, 1\}^m$
- $\hat{\cap}$ is either boolean AND or boolean OR
- $\hat{\top}$ and $\hat{\bot}$ are 0 or 1 depending on $\hat{\cap}$.
- $\hat{h}$ is a **bit function** and could be one of the following:

<table>
<thead>
<tr>
<th>Raised</th>
<th>Lowered</th>
<th>Propagated</th>
<th>Negated</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{\top}$</td>
<td>$\hat{\bot}$</td>
<td>$\hat{\top}$</td>
<td>$\hat{\bot}$</td>
</tr>
<tr>
<td>$\hat{\bot}$</td>
<td>$\hat{\top}$</td>
<td>$\hat{\bot}$</td>
<td>$\hat{\top}$</td>
</tr>
<tr>
<td>$\hat{\bot}$</td>
<td>$\hat{\bot}$</td>
<td>$\hat{\bot}$</td>
<td>$\hat{\bot}$</td>
</tr>
<tr>
<td>$\hat{\bot}$</td>
<td>$\hat{\bot}$</td>
<td>$\hat{\bot}$</td>
<td>$\hat{\bot}$</td>
</tr>
</tbody>
</table>

Non-monotonicity
Boundedness of Constant Propagation

\[ a = 1 \]
\[ a = b + 1 \]
\[ b = c + 1 \]
\[ c = a + 1 \]
Boundedness of Constant Propagation

Summary flow function:
(data flow value at node 7)

\[ f(\langle v_a, v_b, v_c \rangle) = \langle 1 \sqcap (v_b + 1), (v_c + 1), (v_a + 1) \rangle \]
Boundedness of Constant Propagation

Summary flow function:
(data flow value at node 7)

\[ f(\langle v_a, v_b, v_c \rangle) = \langle 1 \sqcap (v_b + 1), (v_c + 1), (v_a + 1) \rangle \]

\[ f^0(\top) = \langle \top, \top, \top \rangle \]
\[ f^1(\top) = \langle 1, \top, \top \rangle \]
Boundedness of Constant Propagation

Summary flow function:
(data flow value at node 7)

\[
f(\langle v_a, v_b, v_c \rangle) = \langle 1 \sqcap (v_b + 1), (v_c + 1), (v_a + 1) \rangle
\]

\[
f^0(\top) = \langle \hat{\top}, \hat{\top}, \hat{\top} \rangle
\]
\[
f^1(\top) = \langle 1, \hat{\top}, \hat{\top} \rangle
\]
\[
f^2(\top) = \langle 1, \hat{\top}, 2 \rangle
\]
Boundedness of Constant Propagation

Summary flow function:
(data flow value at node 7)

\[ f(\langle v_a, v_b, v_c \rangle) = \langle 1 \sqcap (v_b + 1), (v_c + 1), (v_a + 1) \rangle \]

\[ f^0(\top) = \langle \hat{\top}, \hat{\top}, \hat{\top} \rangle \]
\[ f^1(\top) = \langle 1, \hat{\top}, \hat{\top} \rangle \]
\[ f^2(\top) = \langle 1, \hat{\top}, 2 \rangle \]
\[ f^3(\top) = \langle 1, 3, 2 \rangle \]
Boundedness of Constant Propagation

Summary flow function:
(data flow value at node 7)

\[ f(\langle v_a, v_b, v_c \rangle) = \langle 1 \sqcap (v_b + 1), (v_c + 1), (v_a + 1) \rangle \]

\[ f^0(\top) = \langle \top, \top, \top \rangle \]
\[ f^1(\top) = \langle 1, \top, \top \rangle \]
\[ f^2(\top) = \langle 1, \top, 2 \rangle \]
\[ f^3(\top) = \langle 1, 3, 2 \rangle \]
\[ f^4(\top) = \langle \bot, 3, 2 \rangle \]
Boundedness of Constant Propagation

Summary flow function:
(data flow value at node 7)

\[ f(\langle v_a, v_b, v_c \rangle) = \langle 1 \sqcap (v_b + 1), (v_c + 1), (v_a + 1) \rangle \]

\[ f^0(\top) = \langle \hat{\top}, \hat{\top}, \hat{\top} \rangle \]
\[ f^1(\top) = \langle 1, \hat{\top}, \hat{\top} \rangle \]
\[ f^2(\top) = \langle 1, \hat{\top}, 2 \rangle \]
\[ f^3(\top) = \langle 1, 3, 2 \rangle \]
\[ f^4(\top) = \langle \bot, 3, 2 \rangle \]
\[ f^5(\top) = \langle \bot, 3, \bot \rangle \]
Boundedness of Constant Propagation

Summary flow function:
(data flow value at node 7)

\[ f(\langle v_a, v_b, v_c \rangle) = \langle 1 \sqcap (v_b + 1), (v_c + 1), (v_a + 1) \rangle \]

\[ f^0(\top) = \langle \top, \top, \top \rangle \]
\[ f^1(\top) = \langle 1, \top, \top \rangle \]
\[ f^2(\top) = \langle 1, \top, 2 \rangle \]
\[ f^3(\top) = \langle 1, 3, 2 \rangle \]
\[ f^4(\top) = \langle \bot, 3, 2 \rangle \]
\[ f^5(\top) = \langle \bot, 3, \bot \rangle \]
\[ f^6(\top) = \langle \bot, \bot, \bot \rangle \]
Boundedness of Constant Propagation

Summary flow function:
(data flow value at node 7)

\[ f(\langle v_a, v_b, v_c \rangle) = \langle 1 \sqcap (v_b + 1), (v_c + 1), (v_a + 1) \rangle \]

\[
\begin{align*}
    f^0(\top) &= \langle \hat{\top}, \hat{\top}, \hat{\top} \rangle \\
    f^1(\top) &= \langle 1, \hat{\top}, \hat{\top} \rangle \\
    f^2(\top) &= \langle 1, \hat{\top}, 2 \rangle \\
    f^3(\top) &= \langle 1, 3, 2 \rangle \\
    f^4(\top) &= \langle \bot, 3, 2 \rangle \\
    f^5(\top) &= \langle \bot, 3, \bot \rangle \\
    f^6(\top) &= \langle \bot, \bot, \bot \rangle \\
    f^7(\top) &= \langle \bot, \bot, \bot \rangle 
\end{align*}
\]
Boundedness of Constant Propagation

\[ f^*(\top) = \prod_{i=0}^{6} f^i(\top) \]
Boundedness of Constant Propagation

The moral of the story:

- The data flow value of every variable could change twice
The moral of the story:

- The data flow value of every variable could change twice
- In the worst case, only one change may happen in every step of a function application
The moral of the story:

- The data flow value of every variable could change twice
- In the worst case, only one change may happen in every step of a function application
- Maximum number of steps: $2 \times |\text{Var}|$
Boundedness of Constant Propagation

The moral of the story:

- The data flow value of every variable could change twice
- In the worst case, only one change may happen in every step of a function application
- Maximum number of steps: $2 \times |\text{Var}|$
- Boundedness parameter $k$ is $(2 \times |\text{Var}|) + 1$
Modelling Flow Functions for General Flows

- General flow functions can be written as

\[ f_n(X) = (X - \text{Kill}_n(X)) \cup \text{Gen}_n(X) \]

where \( \text{Gen} \) and \( \text{Kill} \) have constant and dependent parts

\[ \text{Gen}_n(X) = \text{ConstGen}_n \cup \text{DepGen}_n(X) \]
\[ \text{Kill}_n(X) = \text{ConstKill}_n \cup \text{DepKill}_n(X) \]
Modelling Flow Functions for General Flows

- General flow functions can be written as
  \[ f_n(X) = (X - \text{Kill}_n(X)) \cup \text{Gen}_n(X) \]

  where \( \text{Gen} \) and \( \text{Kill} \) have constant and dependent parts

  \[ \text{Gen}_n(X) = \text{ConstGen}_n \cup \text{DepGen}_n(X) \]
  \[ \text{Kill}_n(X) = \text{ConstKill}_n \cup \text{DepKill}_n(X) \]

- The dependent parts take care of
  - dependence across different entities as well as
  - dependence on the value of the same entity in the argument \( X \)
Modelling Flow Functions for General Flows

• General flow functions can be written as

\[ f_n(X) = (X - \text{Kill}_n(X)) \cup \text{Gen}_n(X) \]

where Gen and Kill have constant and dependent parts

\[ \text{Gen}_n(X) = \text{ConstGen}_n \cup \text{DepGen}_n(X) \]
\[ \text{Kill}_n(X) = \text{ConstKill}_n \cup \text{DepKill}_n(X) \]

• The dependent parts take care of
  ▶ dependence across different entities as well as
  ▶ dependence on the value of the same entity in the argument \( X \)

• Bit vector frameworks are a special case

\[ \text{DepGen}_n(X) = \text{DepKill}_n(X) = \emptyset \]
Component Lattice for Integer Constant Propagation

\[(\wedge)\]

- Overall lattice \( L \) is the product of \( \hat{L} \) for all variables.
- \( \sqcap \) and \( \hat{\sqcap} \) get defined by \( \sqsubseteq \) and \( \hat{\sqsubseteq} \).

<table>
<thead>
<tr>
<th>( \hat{\sqcap} )</th>
<th>( \langle v, ? \rangle )</th>
<th>( \langle v, \times \rangle )</th>
<th>( \langle v, c_1 \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle v, ? \rangle )</td>
<td>( \langle v, ? \rangle )</td>
<td>( \langle v, \times \rangle )</td>
<td>( \langle v, c_1 \rangle )</td>
</tr>
<tr>
<td>( \langle v, \times \rangle )</td>
<td>( \langle v, \times \rangle )</td>
<td>( \langle v, \times \rangle )</td>
<td>( \langle v, \times \rangle )</td>
</tr>
<tr>
<td>( \langle v, c_2 \rangle )</td>
<td>( \langle v, c_2 \rangle )</td>
<td>( \langle v, \times \rangle )</td>
<td>( \text{If } c_1 = c_2 \text{ then } \langle v, c_1 \rangle \text{ else } \langle v, \times \rangle )</td>
</tr>
</tbody>
</table>
Flow Functions for Constant Propagation

- Flow function for $r = a_1 * a_2$

<table>
<thead>
<tr>
<th>Function</th>
<th>$\langle a_1, ? \rangle$</th>
<th>$\langle a_1, \times \rangle$</th>
<th>$\langle a_1, c_1 \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle a_2, ? \rangle$</td>
<td>$\langle r, ? \rangle$</td>
<td>$\langle r, \times \rangle$</td>
<td>$\langle r, ? \rangle$</td>
</tr>
<tr>
<td>$\langle a_2, \times \rangle$</td>
<td>$\langle r, \times \rangle$</td>
<td>$\langle r, \times \rangle$</td>
<td>$\langle r, \times \rangle$</td>
</tr>
<tr>
<td>$\langle a_2, c_2 \rangle$</td>
<td>$\langle r, ? \rangle$</td>
<td>$\langle r, \times \rangle$</td>
<td>$\langle r, (c_1 * c_2) \rangle$</td>
</tr>
</tbody>
</table>
## Defining Data Flow Equations for Constant Propagation

<table>
<thead>
<tr>
<th></th>
<th>$ConstGen_n$</th>
<th>$DepGen_n(X)$</th>
<th>$ConstKill_n$</th>
<th>$DepKill_n(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v = c$, $c \in \text{Const}$</td>
<td>${\langle v, c \rangle}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${\langle v, d \rangle</td>
</tr>
<tr>
<td>$v = e$, $e \in \text{Expr}$</td>
<td>$\emptyset$</td>
<td>${\langle v, \text{eval}(e,X)\rangle}$</td>
<td>$\emptyset$</td>
<td>${\langle v, d \rangle</td>
</tr>
<tr>
<td>$\text{read}(v)$</td>
<td>${\langle v, \times \rangle}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${\langle v, d \rangle</td>
</tr>
<tr>
<td>other</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
## Defining Data Flow Equations for Constant Propagation

<table>
<thead>
<tr>
<th></th>
<th>$\text{ConstGen}_n$</th>
<th>$\text{DepGen}_n(X)$</th>
<th>$\text{ConstKill}_n$</th>
<th>$\text{DepKill}_n(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v = c, c \in \text{Const}$</td>
<td>${\langle v, c \rangle}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${\langle v, d \rangle</td>
</tr>
<tr>
<td>$v = e, e \in \text{Expr}$</td>
<td>$\emptyset$</td>
<td>${\langle v, \text{eval}(e, X) \rangle}$</td>
<td>$\emptyset$</td>
<td>${\langle v, d \rangle</td>
</tr>
<tr>
<td>$\text{read}(v)$</td>
<td>${\langle v, \times \rangle}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${\langle v, d \rangle</td>
</tr>
<tr>
<td>$\text{other}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

### eval($a_1 \text{ op } a_2, X$)

<table>
<thead>
<tr>
<th></th>
<th>$\langle a_1, ? \rangle \in X$</th>
<th>$\langle a_1, \times \rangle \in X$</th>
<th>$\langle a_1, c_1 \rangle \in X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle a_2, ? \rangle \in X$</td>
<td>?</td>
<td>$\times$</td>
<td>?</td>
</tr>
<tr>
<td>$\langle a_2, \times \rangle \in X$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
</tr>
<tr>
<td>$\langle a_2, c_2 \rangle \in X$</td>
<td>?</td>
<td>$\times$</td>
<td>$c_1 \text{ op } c_2$</td>
</tr>
</tbody>
</table>
Example Program for Constant Propagation

n1: `read (e);`

n2: `a = 7; b = 2; f = e; if (f > 0)`

- **true** path:
  - n3: `a = 2; if (f ≥ e + 2)`
    - **true** path:
      - n6: `if (f ≥ e + 1)`
    - **false** path:
      - n4: `b = c + 1; if (b ≥ 7)`
        - **true** path:
          - n5: `f = f + 1;`
        - **false** path:
          - n7: `c = d * a;`

- **false** path:
  - n8: `d = a + b;`

n10: `e = a + b;`
## Result of Constant Propagation

<table>
<thead>
<tr>
<th></th>
<th>Iteration #1</th>
<th>Changes in iteration #2</th>
<th>Changes in iteration #3</th>
<th>Changes in iteration #4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$In_{n_1}$</td>
<td>$\uparrow, \uparrow, \uparrow, \uparrow, \uparrow, \uparrow$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Out_{n_1}$</td>
<td>$\uparrow, \uparrow, \uparrow, \uparrow, \downarrow, \uparrow$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$In_{n_2}$</td>
<td>$\uparrow, \uparrow, \uparrow, \uparrow, \downarrow, \uparrow$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Out_{n_2}$</td>
<td>$7, 2, \uparrow, \uparrow, \downarrow, \uparrow$</td>
<td>$\uparrow, 2, \uparrow, 3, \downarrow, \downarrow$</td>
<td>$\uparrow, 2, 6, 3, \downarrow, \downarrow$</td>
<td>$\uparrow, \downarrow, 6, 3, \downarrow, \downarrow$</td>
</tr>
<tr>
<td>$In_{n_3}$</td>
<td>$7, 2, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$\uparrow, 2, \uparrow, 3, \downarrow, \downarrow$</td>
<td>$\uparrow, 2, 6, 3, \downarrow, \downarrow$</td>
<td>$\uparrow, \downarrow, 6, 3, \downarrow, \downarrow$</td>
</tr>
<tr>
<td>$Out_{n_3}$</td>
<td>$2, 2, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$2, 2, \uparrow, 3, \downarrow, \downarrow$</td>
<td>$2, 2, 6, 3, \downarrow, \downarrow$</td>
<td>$2, \downarrow, 6, 3, \downarrow, \downarrow$</td>
</tr>
<tr>
<td>$In_{n_4}$</td>
<td>$2, 2, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$2, 2, \uparrow, 3, \downarrow, \downarrow$</td>
<td>$2, 2, 6, 3, \downarrow, \downarrow$</td>
<td>$2, \downarrow, 6, 3, \downarrow, \downarrow$</td>
</tr>
<tr>
<td>$Out_{n_4}$</td>
<td>$2, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$2, \uparrow, \uparrow, 3, \downarrow, \downarrow$</td>
<td>$2, 7, 6, 3, \downarrow, \downarrow$</td>
<td>$2, \downarrow, 6, 3, \downarrow, \downarrow$</td>
</tr>
<tr>
<td>$In_{n_5}$</td>
<td>$2, \uparrow, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$2, \uparrow, \uparrow, 3, \downarrow, \downarrow$</td>
<td>$2, 7, 6, 3, \downarrow, \downarrow$</td>
<td>$2, \downarrow, 6, 3, \downarrow, \downarrow$</td>
</tr>
<tr>
<td>$Out_{n_5}$</td>
<td>$2, \uparrow, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$2, \uparrow, \uparrow, 3, \downarrow, \downarrow$</td>
<td>$2, 7, 6, 3, \downarrow, \downarrow$</td>
<td>$2, \downarrow, 6, 3, \downarrow, \downarrow$</td>
</tr>
<tr>
<td>$In_{n_6}$</td>
<td>$2, 2, \uparrow, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$2, 2, \uparrow, 3, \downarrow, \downarrow$</td>
<td>$2, 2, 6, 3, \downarrow, \downarrow$</td>
<td>$2, \downarrow, 6, 3, \downarrow, \downarrow$</td>
</tr>
<tr>
<td>$Out_{n_6}$</td>
<td>$2, 2, \uparrow, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$2, 2, \uparrow, 3, \downarrow, \downarrow$</td>
<td>$2, 2, 6, 3, \downarrow, \downarrow$</td>
<td>$2, \downarrow, 6, 3, \downarrow, \downarrow$</td>
</tr>
<tr>
<td>$In_{n_7}$</td>
<td>$2, 2, \uparrow, \uparrow, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$2, 2, \uparrow, 3, \downarrow, \downarrow$</td>
<td>$2, 2, 6, 3, \downarrow, \downarrow$</td>
<td>$2, \downarrow, 6, 3, \downarrow, \downarrow$</td>
</tr>
<tr>
<td>$Out_{n_7}$</td>
<td>$2, 2, \uparrow, \uparrow, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$2, 2, \uparrow, 3, \downarrow, \downarrow$</td>
<td>$2, 2, 6, 3, \downarrow, \downarrow$</td>
<td>$2, \downarrow, 6, 3, \downarrow, \downarrow$</td>
</tr>
<tr>
<td>$In_{n_8}$</td>
<td>$2, 2, \uparrow, \uparrow, \uparrow, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$2, 2, \uparrow, 3, \downarrow, \downarrow$</td>
<td>$2, 2, 6, 3, \downarrow, \downarrow$</td>
<td>$2, \downarrow, 6, 3, \downarrow, \downarrow$</td>
</tr>
<tr>
<td>$Out_{n_8}$</td>
<td>$2, 2, \uparrow, \uparrow, \uparrow, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$2, 2, \uparrow, 4, \downarrow, \downarrow$</td>
<td>$2, 2, 6, 4, \downarrow, \downarrow$</td>
<td>$2, \downarrow, 6, \downarrow, \downarrow$</td>
</tr>
<tr>
<td>$In_{n_9}$</td>
<td>$2, 2, \uparrow, \uparrow, \uparrow, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$2, 2, 6, \uparrow, \downarrow, \downarrow$</td>
<td>$2, \downarrow, 6, \downarrow, \downarrow$</td>
<td></td>
</tr>
<tr>
<td>$Out_{n_9}$</td>
<td>$2, 2, \uparrow, \uparrow, \uparrow, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$2, 2, 6, 3, \downarrow, \downarrow$</td>
<td>$2, \downarrow, 6, 3, \downarrow, \downarrow$</td>
<td></td>
</tr>
<tr>
<td>$In_{n_{10}}$</td>
<td>$\uparrow, 2, \uparrow, \uparrow, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$\uparrow, 2, \uparrow, 3, \downarrow, \downarrow$</td>
<td>$\uparrow, 6, 3, \downarrow, \downarrow$</td>
<td>$\uparrow, \downarrow, 6, 3, \downarrow, \downarrow$</td>
</tr>
<tr>
<td>$Out_{n_{10}}$</td>
<td>$\uparrow, 2, \uparrow, \uparrow, \uparrow, \uparrow, \uparrow, \downarrow, \downarrow$</td>
<td>$\uparrow, 2, \uparrow, 3, \downarrow, \downarrow$</td>
<td>$\uparrow, 6, 3, \downarrow, \downarrow$</td>
<td>$\uparrow, \downarrow, 6, 3, \downarrow, \downarrow$</td>
</tr>
</tbody>
</table>
Monotonicity of Constant Propagation

- Flow function $f_n(X) = (X - \text{Kill}_n(X)) \cup \text{Gen}_n(X)$ where
  \[
  \text{Gen}_n(X) = \text{ConstGen}_n \cup \text{DepGen}_n(X) \\
  \text{Kill}_n(X) = \text{ConstKill}_n \cup \text{DepKill}_n(X)
  \]

- $\text{ConstGen}_n$ and $\text{ConstKill}_n$ are trivially monotonic

- To show $X_1 \subseteq X_2 \Rightarrow \text{DepGen}_n(X_1) \subseteq \text{DepGen}_n(X_2)$
  we need to show that $X_1 \subseteq X_2 \Rightarrow \text{eval}(e, X_1) \subseteq \text{eval}(e, X_2)$.
  This follows from definition of $\text{eval}(e, X)$.

- To show $X_1 \subseteq X_2 \Rightarrow (X_1 - \text{DepKill}_n(X_1)) \subseteq (X_2 - \text{DepKill}_n(X_2))$
  observe that $\text{DepKill}_n$ removes the pair corresponding to the variable modified in statement $n$.
  Data flow values of other variables remain unaffected.
Conditional Constant Propagation

An execution trace of the program when the value read for variable $e$ is some number $x \leq 0$:

1. **Read**: $e$
2. \[ a = 7; b = 2; f = e; \]
   - If $f > 0$
     - **Block**: \[ a = 2; \]
       - If $f \geq e + 2$
         - **Block**: \[ b = c + 1; \]
           - If $b \geq 7$
             - **Block**: \[ f = f + 1; \]
           - Otherwise: $f = f + 1$
         - Otherwise: $f = f + 1$
       - Otherwise: $f = f + 1$
     - Otherwise: $f = f + 1$
   - Otherwise: $f = f + 1$

3. **Block**: \[ c = d * a; \]
4. **Block**: \[ d = a + b; \]
5. **Block**: \[ d = a + 1; \]
6. **Block**: \[ f = f + 1; \]
7. **Block**: \[ e = a + b; \]
Conditional Constant Propagation

An execution trace of the program when the value read for variable \( e \) is some number \( x \leq 0 \)

\[
\begin{align*}
\text{n1: } & \quad \text{read}(e); \\
\text{n2: } & \quad a = 7; \ b = 2; \ f = e; \quad \text{if}(f > 0) \\
\text{n3: } & \quad a = 2; \quad \text{if}(f \geq e + 2) \\
\text{n4: } & \quad b = c + 1; \quad \text{if}(b \geq 7) \\
\text{n5: } & \quad f = f + 1; \\
\text{n6: } & \quad \text{if}(f \geq e + 1) \\
\text{n7: } & \quad c = d \ast a; \\
\text{n8: } & \quad d = a + b; \\
\text{n9: } & \quad d = a + 1; \ f = f + 1 \\
\text{n10: } & \quad e = a + b;
\end{align*}
\]

May 2011
Uday Khedker
Conditional Constant Propagation

An execution trace of the program when the value read for variable \( e \) is some number \( x \leq 0 \)

```
read(e);

a = 7; b = 2; f = e;
if (f > 0)

if (f > 0)
\[ n_2 \]

if (f \geq e + 2)
\[ n_3 \]

if (b \geq 7)
\[ n_4 \]

f = f + 1;
\[ n_5 \]

c = d * a;
\[ n_7 \]

d = a + b;
\[ n_8 \]

d = a + 1;
f = f + 1
\[ n_9 \]

e = a + b;
\[ n_{10} \]
```
Conditional Constant Propagation

An execution trace of the program when the value read for variable \( e \) is some number \( x \leq 0 \)

\[ \langle 2, 2, ?, ?, x, x \rangle \]

\[ \langle 2, 2, ?, ?, x, x \rangle \]

\[ \langle 2, 2, ?, 4, x, x \rangle \]
Conditional Constant Propagation

An execution trace of the program when the value read for variable
\( e \) is some number \( x \leq 0 \)

\[
\begin{align*}
&n_1: \text{read (e);} \\
&n_2: \begin{align*}
  &a = 7; \ b = 2; \ f = e; \\
  &\text{if } (f > 0)
\end{align*} \\
&\quad \begin{cases}
  n_3: a = 2; & \text{if } (f \geq e + 2) \\
  n_4: b = c + 1; & \text{if } (b \geq 7) \\
  n_5: f = f + 1;
\end{cases}
\end{align*}
\]

\[
\begin{align*}
&n_6: \text{if } (f \geq e + 1) \\
&n_7: c = d \times a; \\
&n_8: d = a + b; \\
&n_9: d = a + 1; \ f = f + 1
\end{align*}
\]

\[
\begin{align*}
&n_{10}: \ e = a + b;
\end{align*}
\]

\[
\langle 2, 2, ?, ?, x, x \rangle \\
\langle 2, 2, ?, ?, x, x \rangle \\
\langle 2, 2, ?, 4, x, x \rangle \\
\langle 2, 2, ?, 3, x, x+1 \rangle
\]
Conditional Constant Propagation

An execution trace of the program when the value read for variable e is some number $x \leq 0$

- $n_1$: read (e);
- $n_2$: $a = 7; b = 2; f = e; \text{if } (f > 0)$
- $n_3$: $a = 2; \text{if } (f \geq e + 2)$
  - $n_4$: $b = c + 1; \text{if } (b \geq 7)$
  - $n_5$: $f = f + 1$
- $n_6$: $\text{if } (f \geq e + 1)$
- $n_7$: $c = d \times a$
- $n_8$: $d = a + b$
- $n_9$: $d = a + 1; f = f + 1$
- $n_{10}$: $e = a + b$
An execution trace of the program when the value read for variable $e$ is some number $x \leq 0$. 

```
read (e);

n2
a = 7; b = 2; f = e;
if (f > 0)
```

```
n3
a = 2;
if (f \geq e + 2)
```

```
n4
b = c + 1;
if (b \geq 7)
```

```
n5
f = f + 1;
```

```
n6
if (f \geq e + 1)
```

```
n7
c = d * a;
```

```
n8
d = a + b;
```

```
n9
d = a + 1;
f = f + 1
```

```
n10
e = a + b;
```
An execution trace of the program when the value read for variable \( e \) is some number \( x \leq 0 \)

\[
\begin{align*}
n_1 & \quad \text{read (e);} \\
n_2 & \quad a = 7; b = 2; f = e; \\
& \quad \text{if (} f > 0 \text{)} \\
& \quad a = 2; \\
& \quad \text{if (} f \geq e + 2 \text{)} \\
& \quad b = c + 1; \\
& \quad \text{if (} b \geq 7 \text{)} \\
& \quad f = f + 1; \\
& \quad c = d \ast a; \\
& \quad d = a + b; \\
& \quad d = a + 1; \\
& \quad f = f + 1 \\
n_{10} & \quad e = a + b;
\end{align*}
\]
Conditional Constant Propagation

An execution trace of the program when the value read for variable e is some number $x \leq 0$

```
read (e);
```

```
a = 7; b = 2; f = e;
if (f > 0)
```

```
if (f ≥ e + 2)
a = 2;
```

```
b = c + 1;
if (b ≥ 7)
f = f + 1;
```

```
c = d * a;
```

```
d = a + b;
```

```
if (f ≥ e + 1)
d = a + 1;
f = f + 1
```

```
e = a + b;
```

```
false
true
false
false
true
false
false
```
Conditional Constant Propagation

An execution trace of the program when the value read for variable \( e \) is some number \( x \leq 0 \)

1. \( n_1 \): `read (e);`
2. \( n_2 \): `a = 7; b = 2; f = e; if (f > 0)`
3. \( n_3 \): `a = 2; if (f \geq e + 2)`
4. \( n_4 \): `b = c + 1; if (b \geq 7)`
5. \( n_5 \): `f = f + 1;`
6. \( n_6 \): `if (f \geq e + 1)`
7. \( n_7 \): `c = d * a;`
8. \( n_8 \): `d = a + b;`
9. \( n_9 \): `d = a + 1; f = f + 1`
10. \( n_{10} \): `e = a + b;`
Conditional Constant Propagation

An execution trace of the program when the value read for variable $e$ is some number $x \leq 0$

$\langle 2, 2, 6, 3, x, x+2 \rangle$

$\langle 2, 7, 6, 3, x, x+2 \rangle$
Conditional Constant Propagation

An execution trace of the program when the value read for variable $e$ is some number $x \leq 0$.
Conditional Constant Propagation

An execution trace of the program when the value read for variable e is some number \( x \leq 0 \)

regardless of the input value of e, b is constant in the loop and constant propagation cannot discover it

\[
\langle 2, 2, 6, \hat{\perp}, \hat{\perp} \rangle
\]

May 2011

Uday Khedker
Let \( \langle s, X \rangle \) denote an augmented data flow value where \( s \in \{\text{reachable}, \text{notReachable}\} \) and \( X \in L \).

If we can maintain the invariant \( s = \text{notReachable} \Rightarrow X = T \), then the meet can be defined as

\[
\langle s_1, X_1 \rangle \cap_c \langle s_2, X_2 \rangle = \langle s_1 \cap_c s_2, X_1 \cap X_2 \rangle
\]
Data Flow Equations for Conditional Constant Propagation

\[\begin{align*}
\text{In}_n &= \begin{cases} 
\langle \text{reachable}, \text{BI} \rangle & \text{if } n \text{ is Start} \\
\cap_{p \in \text{pred}(n)} g_{p \rightarrow n}(\text{Out}_p) & \text{otherwise}
\end{cases} \\
\text{Out}_n &= \begin{cases} 
\langle \text{reachable}, f_n(X) \rangle & \text{if } \text{In}_n = \langle \text{reachable}, X \rangle \\
\langle \text{notReachable}, \top \rangle & \text{otherwise}
\end{cases}
\end{align*}\]

\[g_{m \rightarrow n}(s, X) = \begin{cases} 
\langle \text{notReachable}, \top \rangle & \text{if } \text{evalCond}(m, X) \neq \text{undefined} \text{ and } \text{evalCond}(m, X) \neq \text{label}(m \rightarrow n) \\
\langle s, X \rangle & \text{otherwise}
\end{cases}\]
## Conditional Constant Propagation

<table>
<thead>
<tr>
<th></th>
<th>Iteration #1</th>
<th>Changes in iteration #2</th>
<th>Changes in iteration #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ln_{n1}$</td>
<td>$R, \langle \top, \top, \top, \top, \top, \top \rangle$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$out_{n1}$</td>
<td>$R, \langle \top, \top, \top, \top, \bot, \top \rangle$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ln_{n2}$</td>
<td>$R, \langle \top, \top, \top, \top, \bot, \bot \rangle$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$out_{n2}$</td>
<td>$R, \langle 7, 2, \top, \top, \bot, \bot \rangle$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ln_{n3}$</td>
<td>$R, \langle 7, 2, \top, \top, \bot, \bot \rangle$</td>
<td>$R, \langle \bot, 2, \top, 3, \bot, \bot \rangle$</td>
<td>$R, \langle \bot, 2, 6, 3, \bot, \bot \rangle$</td>
</tr>
<tr>
<td>$out_{n3}$</td>
<td>$R, \langle 2, 2, \top, \top, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, \top, 3, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, 6, 3, \bot, \bot \rangle$</td>
</tr>
<tr>
<td>$ln_{n4}$</td>
<td>$R, \langle 2, 2, \top, \top, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, \top, 3, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, 6, 3, \bot, \bot \rangle$</td>
</tr>
<tr>
<td>$out_{n4}$</td>
<td>$R, \langle 2, \top, \top, \bot, \bot \rangle$</td>
<td>$R, \langle 2, \top, 3, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 7, 6, 3, \bot, \bot \rangle$</td>
</tr>
<tr>
<td>$ln_{n5}$</td>
<td>$R, \langle 2, \top, \top, \bot, \bot \rangle$</td>
<td>$R, \langle 2, \top, 3, \bot, \bot \rangle$</td>
<td>$N, T = \langle \top, \top, \top, \top, \top, \top \rangle$</td>
</tr>
<tr>
<td>$out_{n5}$</td>
<td>$R, \langle 2, \top, \top, \bot, \bot \rangle$</td>
<td>$R, \langle 2, \top, 3, \bot, \bot \rangle$</td>
<td>$N, T = \langle \top, \top, \top, \top, \top, \top \rangle$</td>
</tr>
<tr>
<td>$ln_{n6}$</td>
<td>$R, \langle 2, 2, \top, \top, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, \top, 3, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, 6, 3, \bot, \bot \rangle$</td>
</tr>
<tr>
<td>$out_{n6}$</td>
<td>$R, \langle 2, 2, \top, \top, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, \top, 3, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, 6, 3, \bot, \bot \rangle$</td>
</tr>
<tr>
<td>$ln_{n7}$</td>
<td>$R, \langle 2, 2, \top, \top, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, \top, 3, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, 6, 3, \bot, \bot \rangle$</td>
</tr>
<tr>
<td>$out_{n7}$</td>
<td>$R, \langle 2, 2, \top, \top, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, \top, 3, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, 6, 3, \bot, \bot \rangle$</td>
</tr>
<tr>
<td>$ln_{n8}$</td>
<td>$R, \langle 2, 2, \top, \top, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, \top, 3, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, 6, 3, \bot, \bot \rangle$</td>
</tr>
<tr>
<td>$out_{n8}$</td>
<td>$R, \langle 2, 2, \top, \top, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, \top, 3, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, 6, 3, \bot, \bot \rangle$</td>
</tr>
<tr>
<td>$ln_{n9}$</td>
<td>$R, \langle 2, 2, \top, 4, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, \top, 4, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, 6, 4, \bot, \bot \rangle$</td>
</tr>
<tr>
<td>$out_{n9}$</td>
<td>$R, \langle 2, 2, \top, 4, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, \top, 4, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, 6, 4, \bot, \bot \rangle$</td>
</tr>
<tr>
<td>$ln_{n10}$</td>
<td>$R, \langle 2, 2, \top, 4, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, \top, 4, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, 6, 4, \bot, \bot \rangle$</td>
</tr>
<tr>
<td>$out_{n10}$</td>
<td>$R, \langle 2, 2, \top, 4, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, \top, 4, \bot, \bot \rangle$</td>
<td>$R, \langle 2, 2, 6, 4, \bot, \bot \rangle$</td>
</tr>
</tbody>
</table>
Part 4

Faint Variables Analysis
A variable is faint if it is dead or is used in computing faint variables.
Faint Variables Analysis

A variable is faint if it is dead or is used in computing faint variables.

1. $y = x$

2. `print (x)`

\[ \{x, y\} \]

\[ \text{Gen}_2 = \emptyset \]
\[ \text{Kill}_2 = \{x\} \]
A variable is faint if it is dead or is used in computing faint variables.

\[
y = x\\
\{y\}
\]

\[
\text{print } (x)\\\{x, y\}
\]

\[
\text{Gen}_2 = \emptyset \\
\text{Kill}_2 = \{x\}
\]

\[
\text{Gen}_1 = \{y\} \\
\text{Kill}_1 = \emptyset
\]
Faint Variables Analysis

A variable is faint if it is dead or is used in computing faint variables.

\[
y = x
\]

\[
\text{print (} x \text{)}
\]

\[
\{x, y\}
\]

\[
\text{Gen}_2 = \emptyset \quad \text{Gen}_1 = \{y\}
\]

\[
\text{Kill}_2 = \{x\} \quad \text{Kill}_1 = \emptyset
\]
A variable is faint if it is dead or is used in computing faint variables.

Faintness of $x$ is killed by the print statement (i.e. $x$ becomes live)
Faint Variables Analysis

A variable is faint if it is dead or is used in computing faint variables.

\[
y = x
\]

1. \(\{y\}\)
2. \(\{x, y\}\)

\[
\text{print}(x)
\]

Gen\(_2\) = \(\emptyset\)  
Kill\(_2\) = \(\{x\}\)

Gen\(_1\) = \(\{y\}\)  
Kill\(_1\) = \(\emptyset\)

1. \(\{y\}\)
2. \(\{x, y\}\)

\[
\text{print}(y)
\]

Gen\(_2\) = \(\emptyset\)  
Kill\(_2\) = \(\{y\}\)

Faintness of \(x\) is killed by the print statement (i.e. \(x\) becomes live)
A variable is faint if it is dead or is used in computing faint variables.

Faintness of x is killed by the print statement (i.e. x becomes live)
Faint Variables Analysis

A variable is faint if it is dead or is used in computing faint variables.

\[
y = x
\]

1. \( y = x \)
   \[
   \{y\}
   \]

2. \( \text{print} \ (x) \)
   \[
   \{x, y\}
   \]

\[
\text{Gen}_2 = \emptyset \quad \text{Gen}_1 = \{y\}
\]
\[
\text{Kill}_2 = \{x\} \quad \text{Kill}_1 = \emptyset
\]

Faintness of \( x \) is killed by the print statement (i.e. \( x \) becomes live)

May 2011
Faint Variables Analysis

1. $y = x$

2. `print (y); print (x);`
Faint Variables Analysis

1 \( y = x \)

2 print (y); print (x); \{x, y\}

\( \text{Gen}_2 = \emptyset \)
\( \text{Kill}_2 = \{x, y\} \)
Faint Variables Analysis

1. \( y = x \)

2. \( \text{print (y); print (x);} \)

\[ \text{Gen}_2 = \emptyset \]
\[ \text{Gen}_1 = \{ y \} \]
\[ \text{Kill}_2 = \{ x, y \} \]
\[ \text{Kill}_1 = \{ x \} \]
Faint Variables Analysis

\[ y = x \]

1 \( \{ y \} \)

\[ \emptyset \]

2 \( \text{print \ (y); print \ (x);} \)

\( \{ x, y \} \)

\( \text{Gen}_2 = \emptyset \quad \text{Gen}_1 = \{ y \} \)

\( \text{Kill}_2 = \{ x, y \} \quad \text{Kill}_1 = \{ x \} \)

Faintness of \( x \) is killed both by the print statement and by the assignment to \( y \) (i.e. \( x \) becomes live)
Data Flow Equations for Faint Variables Analysis

\[
\begin{align*}
\text{In}_n &= f_n(\text{Out}_n) \\
\text{Out}_n &= \begin{cases} 
\text{BI} & \text{n is End} \\
\bigcap_{s \in \text{succ}(n)} \text{In}_s & \text{otherwise}
\end{cases}
\end{align*}
\]

where,

\[
f_n(X) = (X - (\text{ConstKill}_n \cup \text{DepKill}_n(X)))
\]
\[
\cup (\text{ConstGen}_n \cup \text{DepGen}_n(X))
\]

and BI contains all local variables
Flow Function Components for Faint Variables Analysis

<table>
<thead>
<tr>
<th>Statement</th>
<th>read(x) (assigning value from input)</th>
<th>use(x) (not in assignment)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = e, \ e \in \text{Expr}$</td>
<td>${x}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>ConstGen$_n$</td>
<td>$x \notin \text{Opd}(e) \Rightarrow {x}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$x \in \text{Opd}(e) \Rightarrow \emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>ConstKill$_n$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>DepGen$_n(X)$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>DepKill$_n(X)$</td>
<td>$x \notin X \Rightarrow \text{Opd}(e) \cap \text{Var}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$x \in X \Rightarrow \emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Note: For statement $x = e$, $f_n(X)$ is an identity function if $x \in \text{Opd}(e)$
Faint Variable Analysis

- What is $\hat{L}$ for faint variables analysis?
- Is faint variables analysis a bit vector framework?
- Is faint variables analysis distributive? Monotonic?
Distributivity of Faint Variables Analysis

Prove that

\[ \forall X_1, X_2 \in L, \ f_n(X_1 \cap X_2) = f_n(X_1) \cap f_n(X_2) \]
Distributivity of Faint Variables Analysis

Prove that

$$\forall X_1, X_2 \in L, \ f_n(X_1 \cap X_2) = f_n(X_1) \cap f_n(X_2)$$

- $ConstGen_n$, $DepGen_n$, and $ConstKill_n$ are trivially distributive.

Assume that $DepKill_n$ is $\emptyset$.

$$f_n(X) = (X - ConstKill_n) \cup ConstGen_n \cup DepGen_n(X)$$

Since $DepGen_n(X) = \emptyset$, the flow function has only constant parts!
Distributivity of Faint Variables Analysis

To show that

\[(X_1 \cap X_2) - \text{DepKill}_n(X_1 \cap X_2) = (X_1 - \text{DepKill}_n(X_1)) \cap (X_2 - \text{DepKill}_n(X_2))\]
Distributivity of Faint Variables Analysis

To show that

\[(X_1 \cap X_2) - \text{DepKill}_n(X_1 \cap X_2) = (X_1 - \text{DepKill}_n(X_1)) \cap (X_2 - \text{DepKill}_n(X_2))\]

- If \( n \) is an assignment statement \( x = e \), and \( x \notin X_1 \cap X_2 \). Assume that \( x \) is neither in \( X_1 \) nor in \( X_2 \).

\[(X_1 \cap X_2) - \text{DepKill}_n(X_1 \cap X_2)
= (X_1 \cap X_2) - (\text{Opd}(e) \cap \text{Var})
= (X_1 - (\text{Opd}(e) \cap \text{Var})) \cap (X_2 - (\text{Opd}(e) \cap \text{Var}))
= (X_1 - \text{DepKill}_n(X_1)) \cap (X_2 - \text{DepKill}_n(X_2))\]

What if \( x \) is in \( X_1 \) but not in \( X_2 \)?
Distributivity of Faint Variables Analysis

To show that

\[(X_1 \cap X_2) - \text{DepKill}_n(X_1 \cap X_2) = (X_1 - \text{DepKill}_n(X_1)) \cap (X_2 - \text{DepKill}_n(X_2))\]

- If \( n \) is an assignment statement \( x = e \), and \( x \notin X_1 \cap X_2 \). Assume that \( x \) is neither in \( X_1 \) nor in \( X_2 \).

\[
(X_1 \cap X_2) - \text{DepKill}_n(X_1 \cap X_2) \\
= (X_1 \cap X_2) - (\text{Opd}(e) \cap \text{Var}) \\
= (X_1 - (\text{Opd}(e) \cap \text{Var})) \cap (X_2 - (\text{Opd}(e) \cap \text{Var})) \\
= (X_1 - \text{DepKill}_n(X_1)) \cap (X_2 - \text{DepKill}_n(X_2))
\]

What if \( x \) is in \( X_1 \) but not in \( X_2 \)?

- In all other cases, \( \text{DepKill}_n(X) = \emptyset \).
Example Program for Faint Variables Analysis

```
1

n1: d = 0;

n2: if d ≥ 3;

false

n3: if d ≥ 2;

false

true

n4: a = b;

true

false

n5: if d ≥ 1;

true

false

n6: b = c;

false

n7: read (c);

true

false

n8: d = d + 1;

true

false

n9: print a;

May 2011 Uday Khedker
```
## Result of Faint Variables Analysis

<table>
<thead>
<tr>
<th>Node</th>
<th>Iteration #1</th>
<th>Changes in Iteration #2</th>
<th>Changes in Iteration #3</th>
<th>Changes in Iteration #4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$Out_n$</td>
<td>$In_n$</td>
<td>$Out_n$</td>
<td>$In_n$</td>
</tr>
<tr>
<td>$n_9$</td>
<td>${a, b, c, d}$</td>
<td>${b, c, d}$</td>
<td>${b, c}$</td>
<td>${c}$</td>
</tr>
<tr>
<td>$n_8$</td>
<td>${a, b, c, d}$</td>
<td>${a, b, c, d}$</td>
<td>${b, c}$</td>
<td>${b, c}$</td>
</tr>
<tr>
<td>$n_7$</td>
<td>${a, b, c, d}$</td>
<td>${a, b, c, d}$</td>
<td>${b, c}$</td>
<td>${b, c}$</td>
</tr>
<tr>
<td>$n_6$</td>
<td>${a, b, c, d}$</td>
<td>${a, b, c, d}$</td>
<td>${b, c}$</td>
<td>${b, c}$</td>
</tr>
<tr>
<td>$n_5$</td>
<td>${a, b, c, d}$</td>
<td>${a, b, c}$</td>
<td>${b, c}$</td>
<td>${b, c}$</td>
</tr>
<tr>
<td>$n_4$</td>
<td>${a, b, c, d}$</td>
<td>${a, b, c, d}$</td>
<td>${b, c}$</td>
<td>${a, c}$</td>
</tr>
<tr>
<td>$n_3$</td>
<td>${a, b, c}$</td>
<td>${a, b, c}$</td>
<td>${c}$</td>
<td>${c}$</td>
</tr>
<tr>
<td>$n_2$</td>
<td>${b, c}$</td>
<td>${b, c}$</td>
<td>${c}$</td>
<td>${c}$</td>
</tr>
<tr>
<td>$n_1$</td>
<td>${b, c}$</td>
<td>${b, c, d}$</td>
<td>${c}$</td>
<td>${c, d}$</td>
</tr>
</tbody>
</table>
Part 5

Pointer Analyses
Code Optimization In Presence of Pointers

1. q = p;
2. while (...) {
3.     q = q→next;
4. }
5. p→data = r1;
6. print (q→data);
7. p→data = r2;
8. r4 = p→data + r3;

Program

Memory graph at statement 5

- Is p→data live at the exit of line 5? Can we delete line 5?
Code Optimization In Presence of Pointers

1. \( q = p; \)
2. do {
3. \( q = q \rightarrow \text{next}; \)
4. while (…)
5. \( p \rightarrow \text{data} = r1; \)
6. print \((q \rightarrow \text{data});\)
7. \( p \rightarrow \text{data} = r2; \)
8. \( r4 = p \rightarrow \text{data} + r3; \)

- Is \( p \rightarrow \text{data} \) live at the exit of line 5? Can we delete line 5?
Code Optimization In Presence of Pointers

1. \( q = p; \)
2. do 
3. \( q = q \rightarrow \text{next}; \)
4. while (…)
5. \( p \rightarrow \text{data} = r1; \)
6. print \((q \rightarrow \text{data});\)
7. \( p \rightarrow \text{data} = r2; \)
8. \( r4 = p \rightarrow \text{data} + r3; \)

Program

Memory graph at statement 5

- Is \( p \rightarrow \text{data} \) live at the exit of line 5? Can we delete line 5?
- No, if \( p \) and \( q \) can be possibly aliased.
Code Optimization In Presence of Pointers

1. q = p;
2. do {
3.   q = q→next;
4. while (…)
5. p→data = r1;
6. print (q→data);
7. p→data = r2;
8. r4 = p→data + r3;

Program

Memory graph at statement 5

- Is p→data live at the exit of line 5? Can we delete line 5?
- No, if p and q can be possibly aliased.
- Yes, if p and q are definitely not aliased.
Code Optimization In Presence of Pointers

Original Program

\[ a = 5 \]
\[ x = \&a \]
\[ b = *x \]
Code Optimization In Presence of Pointers

Original Program

Constant Propagation
without aliasing
Code Optimization In Presence of Pointers

Original Program

\[ a = 5 \]
\[ x = &a \]
\[ b = *x \]

Constant Propagation without aliasing

\[ a = 5 \]
\[ x = &a \]
\[ b = *x \]

Constant Propagation with aliasing

\[ a = 5 \]
\[ x = &a \]
\[ b = 5 \]
The World of Pointer Analysis

Alias Analysis
- Alias analysis of reference parameters, fields of unions, array indices
- Alias analysis of data pointers

Pointer Analysis
- Points-to analysis of data and function pointers
The Mathematics of Pointer Analysis

In the most general situation

- Alias analysis is undecidable. 

- Flow insensitive alias analysis is NP-hard 
  Horwitz [TOPLAS 1997]

- Points-to analysis is undecidable 
  Chakravarty [POPL 2003]
Motivation for a Good Science of Pointer Analysis

- To quote Hind [PASTE 2001]
Motivation for a Good Science of Pointer Analysis

- To quote Hind [PASTE 2001]
  - Fortunately many approximations exist
Motivation for a Good Science of Pointer Analysis

- To quote Hind [PASTE 2001]
  - Fortunately many approximations exist
  - Unfortunately too many approximations exist!
Motivation for a Good Science of Pointer Analysis

- To quote Hind [PASTE 2001]
  - Fortunately many approximations exist
  - Unfortunately too many approximations exist!

- Pointer analysis enables not only precise data analysis but also precise control flow analysis.
Motivation for a Good Science of Pointer Analysis

- To quote Hind [PASTE 2001]
  - Fortunately many approximations exist
  - Unfortunately too many approximations exist!

- Pointer analysis enables not only precise data analysis but also precise control flow analysis.

- Needs to scale to large programs.
Motivation for a Good Science of Pointer Analysis

• To quote Hind [PASTE 2001]
  ▶ Fortunately many approximations exist
  ▶ Unfortunately too many approximations exist!

• Pointer analysis enables not only precise data analysis but also precise control flow analysis.

• Needs to scale to large programs.

• Engineering of pointer analysis is much more dominant than the science of pointer analysis.

⇒ Results in many questionable perceptions.
Alias Information Vs. Points-To Information

1. $x = \&a$

2. $b = x$
Alias Information Vs. Points-To Information

“\(x\) Points-To \(a\)"
denoted \(\triangleright a\)
Alias Information Vs. Points-To Information

1. \( x = &a \)  
   - "\( x \) Points-To \( a \)"
   - denoted \( x \rightarrow a \)

2. \( b = x \)  
   - "\( x \) and \( b \) are Aliases"
   - denoted \( x \equiv b \)
Alias Information Vs. Points-To Information

1. \( x = &a \) denoted \( x \rightarrow a \)

2. \( b = x \) denoted \( x \equiv b \)

Symmetric and Reflexive
Alias Information Vs. Points-To Information

1. \( x = \&a \)
   - "x Points-To a"
   - denoted \( x \rightarrow a \)

2. \( b = x \)
   - "x and b are Aliases"
   - denoted \( x \equiv b \)

Neither
Symmetric
Nor Reflexive

Symmetric
and
Reflexive

May 2011
Uday Khedker
Alias Information Vs. Points-To Information

1. \[ x = \&a \]
   - "x Points-To a"
   - denoted \( x \rightarrow a \)

2. \[ b = x \]
   - "x and b areAliases"
   - denoted \( x \equiv b \)

- What about transitivity?

- Neither Symmetric Nor Reflexive
- Symmetric and Reflexive
Alias Information Vs. Points-To Information

1. \( x = &a \) denoted \( x \rightarrow a \)

2. \( b = x \) denoted \( x \equiv b \)

- What about transitivity?
  - Points-To: No.

Neither Symmetric Nor Reflexive

Symmetric and Reflexive
Alias Information Vs. Points-To Information

1. \( x = &a \) denoted \( x \rightarrow a \)

2. \( b = x \) denoted \( x \equiv b \)

- What about transitivity?
  - Points-To: No.
  - Alias: Depends.
Must Points-To Information

1 \( x = \& a \)

2

3

4

1. \( x = \& a \)
2. 
3. 
4. 

Diagram: A variable \( x \) points to the address of \( a \).
Must Points-To Information

1. \( x = \&a \)

2. 3.

4.

\( \cdots \)

a

x

a

x

b

b
May Points-To Information

\[
\begin{align*}
1 &: x = &a \\
2 &: x = &b \\
3 &: \\
4 &: \\
\end{align*}
\]
May Points-To Information

1. $x = &a$

2. $x = &b$

3. 

4. 

- Diagram showing pointers $a$, $x$, and $b$.
Must Alias Information

1. $x = &a$
2. $b = x$
3. $y = b$

Diagram:
- Node 1: $x = &a$
- Node 2: $b = x$
- Node 5: $y = b$
- Node 3 and 4: Interconnected with arrows indicating flow of alias information.
Must Alias Information

1. \( x = \&a \)
2. \( b = x \)
3. \( y = b \)
4. 
5. 

\[ a \]
\[ x \]
\[ b \]
\[ y \]
Must Alias Information

1. \( x = \&a \)
2. \( b = x \)
3. 
4. 
5. \( y = b \)
Must Alias Information

\[ x = &a \]

\[ b = x \]

\[ y = b \]

\[ x \equiv b \text{ and } b \equiv y \Rightarrow x \equiv y \]
May Alias Information

1. $x = \&a$
2. $b = \&z$
3. $b = x$
4. $y = b$
5. 

The diagram illustrates the alias relationships between variables $a$, $b$, $y$, and $z$. The arrow from 1 to 2 indicates that $x$ and $b$ can alias $a$ and $z$, respectively, due to the assignment operations.

May 2011 Uday Khedker
May Alias Information

1. \( x = \&a \)
2. \( b = \&z \)
3. \( b = x \)
4. \( y = b \)
5. (Blank)

Diagram:

- Node 1: \( x = \&a \)
- Node 2: \( b = \&z \)
- Node 3: \( b = x \)
- Node 4: \( y = b \)
- Node 5: (Blank)
May Alias Information

1. \( x = \&a \)
2. \( b = \&z \)
3. \( b = x \)
4. \( y = b \)
5.  

\( a \)

\( x \)

\( b \)

\( y \)

\( z \)
May Alias Information

1. \( x = &a \)
2. \( b = &z \)
3. \( b = x \)
4. \( y = b \)
5. (Empty box)

Symbols:
- \( a \), \( b \), \( y \), \( z \)
May Alias Information

1. \( x = \&a \)

2. \( b = \&z \)

3. \( b = x \)

4. \( y = b \)

5. (No change)
May Alias Information

\[
x = &a \\
b = &z \\
b = x \\
y = b
\]

\[
x \equal{≈} b \text{ and } b \not\equal{≈} y \not\Rightarrow x \equal{≈} y
\]
### A Comparison of Points-To and Alias Relations

<table>
<thead>
<tr>
<th>Asgn.</th>
<th>Memory</th>
<th>Points-to</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>(<em>x = y</em></td>
<td><img src="image1" alt="Before" /></td>
<td>(x \mapsto u) (y \mapsto z)</td>
<td>Existing (\ast x \triangleleft u) (\ast y \triangleleft z)</td>
</tr>
<tr>
<td></td>
<td><img src="image2" alt="After" /></td>
<td>(y \mapsto u)</td>
<td>New Direct (\ast x \triangleleft y) (\ast y \triangleleft u)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(u \mapsto z)</td>
<td>New Indirect (\ast u \triangleleft z) (\ast \ast x \triangleleft z)</td>
</tr>
<tr>
<td>(*x = <em>y</em></td>
<td><img src="image3" alt="Before" /></td>
<td>(x \mapsto v) (y \mapsto z) (z \mapsto u)</td>
<td>Existing (\ast x \triangleleft v) (\ast z \triangleleft u) (\ast \ast y \triangleleft u)</td>
</tr>
<tr>
<td></td>
<td><img src="image4" alt="After" /></td>
<td>(y \mapsto u)</td>
<td>New Direct (\ast x \triangleleft *y) (\ast z \triangleleft v) (\ast v \triangleleft *y)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(v \mapsto u)</td>
<td>New Indirect (\ast \ast x \triangleleft u) (\ast v \triangleleft u)</td>
</tr>
</tbody>
</table>

* May 2011 Uday Khedker*
Strong and Weak Updates

1. \( x = \&a \)
2. \( y = \&b \)
   \( w = \&c \)
3. \( z = \&x \)
4. \( z = \&y \)
5. \( *z = \text{null} \)
   \( *w = \text{null} \)
Strong and Weak Updates

Weak update: Modification of $x$ or $y$ due to $*z$ in block 5
Strong and Weak Updates

Weak update: Modification of $x$ or $y$ due to $*z$ in block 5

Strong update: Modification of $c$ due to $*w$ in block 5
Strong and Weak Updates

Weak update: Modification of $x$ or $y$ due to $*z$ in block 5

Strong update: Modification of $c$ due to $*w$ in block 5

How is this concept related to May/Must nature of information?
What About Heap Data?

• Compile time entities, abstract entities, or summarized entities

• Three options:
  ▶ Represent all heap locations by a single abstract heap location
  ▶ Represent all heap locations of a particular type by a single abstract heap location
  ▶ Represent all heap locations allocated at a given memory allocation site by a single abstract heap location

• Summarization: Usually based on the length of pointer expression

• No clean and elegant solution exists
Left and Right Locations in Pointer Assignments

For an assignment statement $lhs_n = rhs_n$

- Left Locations

<table>
<thead>
<tr>
<th>$lhs_n$</th>
<th>$ConstLeftL_n$</th>
<th>$DepLeftL_n(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>${x}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\ast x$</td>
<td>$\emptyset$</td>
<td>${y \mid (x\rightarrow y) \in X}$</td>
</tr>
</tbody>
</table>

- Right Locations

<table>
<thead>
<tr>
<th>$rhs_n$</th>
<th>$ConstRightL_n$</th>
<th>$DepRightL_n(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$\emptyset$</td>
<td>${y \mid (x\rightarrow y) \in X}$</td>
</tr>
<tr>
<td>$\ast x$</td>
<td>$\emptyset$</td>
<td>${z \mid {x\rightarrow y, y\rightarrow z} \subseteq X}$</td>
</tr>
<tr>
<td>$&amp; x$</td>
<td>${x}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Gen and Kill Components

\[
\begin{align*}
\text{ConstGen}_n & = \{ x \mapsto y \mid x \in \text{ConstLeft} L_n, y \in \text{ConstRight} L_n \} \\
\text{DepGen}_n(X) & = \{ x \mapsto y \mid (x \in \text{ConstLeft} L_n, y \in \text{DepRight} L_n(X)), \text{ or } \\
& \quad (x \in \text{DepLeft} L_n(X), y \in \text{ConstRight} L_n), \text{ or } \\
& \quad (x \in \text{DepLeft} L_n(X), y \in \text{DepRight} L_n(X)) \} \\
\text{ConstKill}_n & = \{ x \mapsto y \mid x \in \text{ConstLeft} L_n \} \\
\text{DepKill}_n(X) & = \{ x \mapsto y \mid x \in \text{DepLeft} L_n(X) \}
\end{align*}
\]
**DepKill**(X) in May and Must Points-To Analysis

- **May Points-To analysis**
  - A points-to pair should be removed only if it must be removed along all paths
  - **DepKill**(X) should remove only strong updates
  - X should be Must Points-To information

- **Must Points-To analysis**
  - A points-to pair should be removed if it can be removed along some path
  - **DepKill**(X) should remove all weak updates
  - X should be May Points-To information

- Must Points-To $\subseteq$ May Points-To
DepKill($X$) in May and Must Points-To Analysis

\[ a = \&b \]
\[ c = \&a \]
\[ c = \&d \]
\[ \ast c = e \]

May 2011 Uday Khedker
DepKill(\(X\)) in May and Must Points-To Analysis

\[
\begin{align*}
\text{MustIn}_4 &= \{a \rightarrow b\} \\
\text{DepLeftL}_4(\text{MustIn}_4) &= \emptyset
\end{align*}
\]
$\text{DepKill}(X)$ in May and Must Points-To Analysis

May 2011 Uday Khedker
DepKill(X) in May and Must Points-To Analysis

1. \( a = &b \)
2. \( c = &a \)
3. \( c = &d \)
4. \( *c = e \)
5. \( a \rightarrow b \) at block 5 along path 1, 3, 4, 5 but not along path 1, 2, 4, 5.

\[
\begin{align*}
\text{MustIn}_4 &= \{ a \rightarrow b \} \\
\text{DepLeftL}_4(\text{MustIn}_4) &= \emptyset \\
\text{MayIn}_4 &= \{ a \rightarrow b, c \rightarrow a, c \rightarrow d \} \\
\text{DepLeftL}_4(\text{MayIn}_4) &= \{ a, d \}
\end{align*}
\]
DepKill(X) in May and Must Points-To Analysis

\[
\begin{align*}
1 & \quad a = \&b \\
2 & \quad c = \&a \\
3 & \quad c = \&d \\
4 & \quad \ast c = e \\
5 & \quad \\
\end{align*}
\]

MustIn₄ = \{a\rightarrow b\}
DepLeftL₄(MustIn₄) = \emptyset
MayIn₄ = \{a\rightarrow b, c\rightarrow a, c\rightarrow d\}
DepLeftL₄(MayIn₄) = \{a, d\}

- a\rightarrow b at block 5 along path 1, 3, 4, 5 but not along path 1, 2, 4, 5.
- a\rightarrow b \in MayIn₅ but a\rightarrow b \not\in MustIn₅
DepKill(X) in May and Must Points-To Analysis

\[ \text{MustIn}_4 = \{ a \mapsto b \} \], \quad \text{DepLeftL}_4(\text{MustIn}_4) = \emptyset

\[ \text{MayIn}_4 = \{ a \mapsto b, c \mapsto a, c \mapsto d \} \]
\[ \text{DepLeftL}_4(\text{MayIn}_4) = \{ a, d \} \]

- \( a \mapsto b \) at block 5 along path 1, 3, 4, 5 but not along path 1, 2, 4, 5.
- \( a \mapsto b \in \text{MayIn}_5 \) but \( a \mapsto b \notin \text{MustIn}_5 \)
- If \( \text{DepKill}_n \) for \( \text{MayOut}_4 \) is defined in terms of \( \text{MayIn}_4 \) then \( a \mapsto b \notin \text{MayOut}_4 \) because \( a \) is in \( \text{DepLeftL}_4(\text{MayIn}_4) \).
DepKill(\(X\)) in May and Must Points-To Analysis

\[
\begin{align*}
1 \quad &a = \& b \\
2 \quad &c = \& a \\
3 \quad &c = \& d \\
4 \quad &*c = e \\
5 \quad &
\end{align*}
\]

\begin{align*}
\text{DepLeftL}_4(\text{MustIn}_4) &= \emptyset \\
\text{MustIn}_4 &= \{ \text{a}\rightarrow\text{b} \} \\
\text{MayIn}_4 &= \{ \text{a}\rightarrow\text{b}, \text{c}\rightarrow\text{a}, \text{c}\rightarrow\text{d} \} \\
\text{DepLeftL}_4(\text{MayIn}_4) &= \{ \text{a}, \text{d} \}
\end{align*}

- \(a\rightarrow b\) at block 5 along path 1, 3, 4, 5 but not along path 1, 2, 4, 5.
- \(a\rightarrow b\) \(\in\) \text{MayIn}_5 but \(a\rightarrow b\) \(\notin\) \text{MustIn}_5
- If \(\text{DepKill}_n\) for \(\text{MayOut}_4\) is defined in terms of \(\text{MayIn}_4\) then \(a\rightarrow b\) \(\notin\) \(\text{MayOut}_4\) because \(a\) is in \(\text{DepLeftL}_4(\text{MayIn}_4)\)
- If \(\text{DepKill}_4\) for \(\text{MustOut}_4\) is defined in terms of \(\text{MustIn}_4\) then \(a\rightarrow b\) \(\in\) \(\text{MustOut}_4\) because \(a\) is not in \(\text{DepLeftL}_4(\text{MustIn}_4)\)
Data Flow Equations for Points-To Analysis

\[\text{MayIn}_n = \begin{cases} \text{BI} & \text{if } n \text{ is Start} \\ \bigcup_{p \in \text{pred}(n)} \text{MayOut}_n & \text{otherwise} \end{cases}\]

\[\text{MayOut}_n = f_n(\text{MayIn}_n, \text{MustIn}_n)\]

\[\text{MustIn}_n = \begin{cases} \text{BI} & \text{if } n \text{ is Start} \\ \bigcap_{p \in \text{pred}(n)} \text{MustOut}_n & \text{otherwise} \end{cases}\]

\[\text{MustOut}_n = f_n(\text{MustIn}_n, \text{MayIn}_n)\]

\[f_n(X_1, X_2) = (X_1 - \text{Kill}_n(X_2)) \cup \text{Gen}_n(X_1)\]
Approximating May and Must Alias and Points-To Information

- May Alias: Every pointer variable is aliased to every pointer variable.
- Must Alias: Every pointer variable is aliased only to itself.
Approximating May and Must Alias and Points-To Information

- May Alias: Every pointer variable is aliased to every pointer variable.
- Must Alias: Every pointer variable is aliased only to itself.
- May Points-To: Every pointer variable points to every location.
Approximating May and Must Alias and Points-To Information

- May Alias: Every pointer variable is aliased to every pointer variable.
- Must Alias: Every pointer variable is aliased only to itself.
- May Points-To: Every pointer variable points to every location.
- Must Points-To: No pointer variable points to any location.
Approximating May and Must Alias and Points-To Information

- May Alias: Every pointer variable is aliased to every pointer variable.
- Must Alias: Every pointer variable is aliased only to itself.
- May Points-To: Every pointer variable points to every location.
- Must Points-To: No pointer variable points to any location.
- Both May and Must analyses need not be performed.
Approximating May and Must Alias and Points-To Information

- **May Alias**: Every pointer variable is aliased to every pointer variable.
- **Must Alias**: Every pointer variable is aliased only to itself.
- **May Points-To**: Every pointer variable points to every location.
- **Must Points-To**: No pointer variable points to any location.
- Both May and Must analyses need not be performed.

*In every case, the approximation uses the ⊥ element of the lattice.*
Example Program for Points-To Analysis

- Variables and points-to sets:
  \[ \mathbb{Var} = \{ a, b, c, d \} \]
  \[ \mathbb{U} = \{ a \rightarrow a, a \rightarrow b, a \rightarrow c, a \rightarrow d, b \rightarrow a, b \rightarrow b, b \rightarrow d, b \rightarrow d, c \rightarrow a, c \rightarrow b, c \rightarrow c, c \rightarrow d, d \rightarrow a, d \rightarrow b, d \rightarrow c, d \rightarrow d \} \]
Example Program for Points-To Analysis

- Variables and points-to sets:
  \[ \text{Var} = \{a, b, c, d\} \]
  \[ \mathbb{U} = \{a \mapsto a, a \mapsto b, a \mapsto c, a \mapsto d, \]
  \[ b \mapsto a, b \mapsto b, b \mapsto d, b \mapsto d, \]
  \[ c \mapsto a, c \mapsto b, c \mapsto c, c \mapsto d, \]
  \[ d \mapsto a, d \mapsto b, d \mapsto c, d \mapsto d \} \]

- \( L_{\text{may}} = \langle 2^\mathbb{U}, \supseteq \rangle \), \( T_{\text{may}} = \emptyset \), \( \bot_{\text{may}} = \mathbb{U} \)
Example Program for Points-To Analysis

- Variables and points-to sets:
  \[ \Var = \{ a, b, c, d \} \]
  \[ \U = \{ a \mapsto a, a \mapsto b, a \mapsto c, a \mapsto d, b \mapsto a, b \mapsto b, b \mapsto d, b \mapsto d, c \mapsto a, c \mapsto b, c \mapsto c, c \mapsto d, d \mapsto a, d \mapsto b, d \mapsto c, d \mapsto d \} \]

- \( L_{\text{may}} = \langle 2^\U, \supseteq \rangle \), \( \top_{\text{may}} = \emptyset \), \( \bot_{\text{may}} = \U \)

- \( L_{\text{must}} = \hat{L}_a \times \hat{L}_b \times \hat{L}_c \times \hat{L}_d \)
  The component lattice \( \hat{L}_a \) is:
  \[ \{ a \mapsto a, a \mapsto b, a \mapsto c, a \mapsto d \} \]
  \[ \{ a \mapsto a \} \{ a \mapsto b \} \{ a \mapsto c \} \{ a \mapsto d \} \]
  \[ \emptyset \]
# Result of Pointer Analysis

<table>
<thead>
<tr>
<th></th>
<th>Iteration #1</th>
<th>Changes in Iteration #2</th>
<th>Changes in Iteration #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{MayIn}_n )</td>
<td>( \emptyset )</td>
<td>( { a \mapsto b, a \mapsto d, b \mapsto b, b \mapsto d, c \mapsto d } )</td>
<td>( { a \mapsto b, a \mapsto d, b \mapsto b, b \mapsto d, c \mapsto b, c \mapsto d } )</td>
</tr>
<tr>
<td>( \text{MustIn}_n )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{MayOut}_n )</td>
<td>( { b \mapsto d } )</td>
<td>( { a \mapsto b, a \mapsto d, b \mapsto b, b \mapsto d, c \mapsto d } )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{MustOut}_n )</td>
<td>( { b \mapsto d } )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

| \( \text{MayIn}_{n_2} \) | \( \{ b \mapsto d \} \) | \( \{ a \mapsto b, a \mapsto d, b \mapsto b, b \mapsto d, c \mapsto d \} \) | \( \emptyset \) |
| \( \text{MustIn}_{n_2} \) | \( \{ b \mapsto d \} \) | \( \emptyset \) | \( \emptyset \) |
| \( \text{MayOut}_{n_2} \) | \( \{ b \mapsto d, c \mapsto d \} \) | \( \{ a \mapsto b, a \mapsto d, b \mapsto b, b \mapsto d, c \mapsto b, c \mapsto d \} \) | \( \emptyset \) |
| \( \text{MustOut}_{n_2} \) | \( \{ b \mapsto d, c \mapsto d \} \) | \( \emptyset \) | \( \emptyset \) |

| \( \text{MayIn}_{n_3} \) | \( \{ b \mapsto d, c \mapsto d \} \) | \( \{ a \mapsto b, a \mapsto d, b \mapsto b, b \mapsto d, c \mapsto d \} \) | \( \emptyset \) |
| \( \text{MustIn}_{n_3} \) | \( \{ b \mapsto d, c \mapsto d \} \) | \( \emptyset \) | \( \emptyset \) |
| \( \text{MayOut}_{n_3} \) | \( \{ a \mapsto b, b \mapsto d, c \mapsto d \} \) | \( \{ a \mapsto b, b \mapsto b, b \mapsto d, c \mapsto b, c \mapsto d \} \) | \( \emptyset \) |
| \( \text{MustOut}_{n_3} \) | \( \{ a \mapsto b, b \mapsto d, c \mapsto d \} \) | \( \{ a \mapsto b \} \) | \( \emptyset \) |
# Result of Pointer Analysis

<table>
<thead>
<tr>
<th></th>
<th>Iteration #1</th>
<th>Changes in Iteration #2</th>
<th>Changes in Iteration #3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MayInₙ₄</strong></td>
<td>{a \rightarrow b, b \rightarrow d, c \rightarrow d}</td>
<td>{a \rightarrow b, b \rightarrow b, b \rightarrow d, c \rightarrow b, c \rightarrow d}</td>
<td></td>
</tr>
<tr>
<td><strong>MustInₙ₄</strong></td>
<td>{a \rightarrow b, b \rightarrow d, c \rightarrow d}</td>
<td>{a \rightarrow b}</td>
<td></td>
</tr>
<tr>
<td><strong>MayOutₙ₄</strong></td>
<td>{a \rightarrow b, b \rightarrow b, c \rightarrow d}</td>
<td>{a \rightarrow b, b \rightarrow b, c \rightarrow b, c \rightarrow d}</td>
<td></td>
</tr>
<tr>
<td><strong>MustOutₙ₄</strong></td>
<td>{a \rightarrow b, b \rightarrow b, c \rightarrow d}</td>
<td>{a \rightarrow b, b \rightarrow b}</td>
<td></td>
</tr>
<tr>
<td><strong>MayInₙ₅</strong></td>
<td>{b \rightarrow d, c \rightarrow d}</td>
<td>{a \rightarrow b, a \rightarrow d, b \rightarrow b, b \rightarrow d, c \rightarrow b, c \rightarrow d}</td>
<td></td>
</tr>
<tr>
<td><strong>MustInₙ₅</strong></td>
<td>{b \rightarrow d, c \rightarrow d}</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td><strong>MayOutₙ₅</strong></td>
<td>{a \rightarrow c, b \rightarrow d, c \rightarrow d}</td>
<td>{a \rightarrow c, b \rightarrow b, b \rightarrow d, c \rightarrow b, c \rightarrow d}</td>
<td></td>
</tr>
<tr>
<td><strong>MustOutₙ₅</strong></td>
<td>{a \rightarrow c, b \rightarrow d, c \rightarrow d}</td>
<td>{a \rightarrow c}</td>
<td></td>
</tr>
</tbody>
</table>
# Result of Pointer Analysis

<table>
<thead>
<tr>
<th></th>
<th>Iteration #1</th>
<th>Changes in Iteration #2</th>
<th>Changes in Iteration #3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MayIn</strong>&lt;sub&gt;n&lt;sub&gt;6&lt;/sub&gt;&lt;/sub&gt;</td>
<td>{a\rightarrow b, a\rightarrow c, b\rightarrow b, b\rightarrow d, c\rightarrow d}</td>
<td>{a\rightarrow b, a\rightarrow c, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d}</td>
<td></td>
</tr>
<tr>
<td><strong>MustIn</strong>&lt;sub&gt;n&lt;sub&gt;6&lt;/sub&gt;&lt;/sub&gt;</td>
<td>{c\rightarrow d}</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td><strong>MayOut</strong>&lt;sub&gt;n&lt;sub&gt;6&lt;/sub&gt;&lt;/sub&gt;</td>
<td>{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow d}</td>
<td>{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d}</td>
<td></td>
</tr>
<tr>
<td><strong>MustOut</strong>&lt;sub&gt;n&lt;sub&gt;6&lt;/sub&gt;&lt;/sub&gt;</td>
<td>{c\rightarrow d}</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td><strong>MayIn</strong>&lt;sub&gt;n&lt;sub&gt;7&lt;/sub&gt;&lt;/sub&gt;</td>
<td>{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow d}</td>
<td>{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d}</td>
<td></td>
</tr>
<tr>
<td><strong>MustIn</strong>&lt;sub&gt;n&lt;sub&gt;7&lt;/sub&gt;&lt;/sub&gt;</td>
<td>{c\rightarrow d}</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td><strong>MayOut</strong>&lt;sub&gt;n&lt;sub&gt;7&lt;/sub&gt;&lt;/sub&gt;</td>
<td>{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow d, d\rightarrow d}</td>
<td>{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d, d\rightarrow b, d\rightarrow d}</td>
<td></td>
</tr>
<tr>
<td><strong>MustOut</strong>&lt;sub&gt;n&lt;sub&gt;7&lt;/sub&gt;&lt;/sub&gt;</td>
<td>{c\rightarrow d}</td>
<td>{}</td>
<td></td>
</tr>
</tbody>
</table>
Non-Distributivity of Points-To Analysis

May Points-To

\[ n_1 \]

\[ n_2 \quad x = &z \quad n_3 \quad y = &w \]

\[ n_4 \quad *x = y \]

Must Points-To

\[ n_1 \]

\[ n_2 \quad b = &c \quad n_3 \quad b = &e \]

\[ c = &d \quad e = &d \]

\[ n_4 \quad a = *b \]
Non-Distributivity of Points-To Analysis

May Points-To

Must Points-To

\[ z \rightarrow w \] is spurious
Non-Distributivity of Points-To Analysis

May Points-To

$\text{n}_1$

$\text{n}_2 \ x = &z$

$\text{n}_3 \ y = &w$

$\text{n}_4 \ *x = y$

$z \rightarrow w$ is spurious

Must Points-To

$\text{n}_1$

$\text{n}_2 \ b = &c$

$\text{n}_3 \ c = &d$

$\text{n}_4 \ a = *b$

$\text{n}_2 \ b = &e$

$\text{n}_3 \ e = &d$

$a \rightarrow d$ is missing
Part 6

Heap Reference Analysis
Motivating Example for Heap Liveness Analysis

If the while loop is not executed even once.

1. \( w = x \)  // \( x \) points to \( m_a \)
2. while (\( x.data < max \))
3. \( x = x.rptr \)
4. \( y = x.lptr \)
5. \( z = \text{New class of} \ z \)
6. \( y = y.lptr \)
7. \( z.sum = x.data + y.data \)

Stack

Heap
Motivating Example for Heap Liveness Analysis

If the while loop is executed once.

1. \( w = x \)  // \( x \) points to \( m_a 
2. while (x.data < max)
3. \( x = x.rptr \)
4. \( y = x.lptr \)
5. \( z = \text{New class of } z \)
6. \( y = y.lptr \)
7. \( z.sum = x.data + y.data \)

Stack

Heap
Motivating Example for Heap Liveness Analysis

If the while loop is executed twice.

```plaintext
1. w = x  // x points to ma
2. while (x.data < max)
3.   x = x.rptr
4. y = x.lptr
5. z = New class_of_z
6. y = y.lptr
7. z.sum = x.data + y.data
```
The Moral of the Story

• Mappings between access expressions and l-values keep changing

• This is a *rule* for heap data
  For stack and static data, it is an *exception*!

• Static analysis of programs has made significant progress for stack and static data.

What about heap data?

▶ Given two access expressions at a program point, do they have the same l-value?
▶ Given the same access expression at two program points, does it have the same l-value?
Our Solution

\[y = z = \text{null}\]

1 \hspace{1cm} w = x
\hspace{1cm} w = \text{null}

2 \hspace{1cm} \text{while} \ (x.\text{data} < \text{max})
\hspace{1cm} \{ \hspace{1cm} x.\text{lptr} = \text{null} \}
\hspace{1cm} x = x.\text{rptr}
\hspace{1cm} x.\text{rptr} = x.\text{lptr}.\text{rptr} = \text{null}
\hspace{1cm} x.\text{lptr}.\text{lptr}.\text{lptr} = \text{null}
\hspace{1cm} x.\text{lptr}.\text{lptr}.\text{rptr} = \text{null}

3 \hspace{1cm} y = x.\text{lptr}
\hspace{1cm} x.\text{lptr} = y.\text{rptr} = \text{null}
\hspace{1cm} y.\text{lptr}.\text{lptr} = y.\text{lptr}.\text{rptr} = \text{null}

4 \hspace{1cm} z = \text{New class of z}
\hspace{1cm} z.\text{lptr} = z.\text{rptr} = \text{null}

5 \hspace{1cm} y = y.\text{lptr}
\hspace{1cm} y.\text{lptr} = y.\text{rptr} = \text{null}

6 \hspace{1cm} z.\text{sum} = x.\text{data} + y.\text{data}
\hspace{1cm} x = y = z = \text{null} \]
Our Solution

y = z = null

1  w = x
   w = null

2  while (x.data < max)
   {  x.lptr = null
      x = x.rptr  }
   x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null

3  y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null

4  z = New class of z
   z.lptr = z.rptr = null

5  y = y.lptr
   y.lptr = y.rptr = null

6  y = y.lptr
   y.lptr = y.rptr = null

7  z.sum = x.data + y.data

x = y = z = null

While loop is not executed even once

Stack

Heap
Our Solution

1. \( y = z = \text{null} \)
2. \( w = x \)
   \( w = \text{null} \)
3. \( \text{while} \ (x.\text{data} < \text{max}) \)
   \{ \( x.\text{lptr} = \text{null} \)
   \( x = x.\text{rptr} \) \}
   \( x.\text{rptr} = x.\text{lptr.}\text{rptr} = \text{null} \)
   \( x.\text{lptr.}\text{lptr.}\text{lptr} = \text{null} \)
   \( x.\text{lptr.}\text{lptr.}\text{rptr} = \text{null} \)
4. \( y = x.\text{lptr} \)
   \( x.\text{lptr} = y.\text{rptr} = \text{null} \)
   \( y.\text{lptr.}\text{lptr} = y.\text{lptr.}\text{rptr} = \text{null} \)
5. \( z = \text{New class of z} \)
   \( z.\text{lptr} = z.\text{rptr} = \text{null} \)
6. \( y = y.\text{lptr} \)
   \( y.\text{lptr} = y.\text{rptr} = \text{null} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
   \( x = y = z = \text{null} \)

While loop is not executed even once
Our Solution

```plaintext
y = z = null

1 w = x
   w = null

2 while (x.data < max)
   { x.lptr = null
3     x = x.rptr
   }
   x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null

4 y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null

5 z = New class of z
   z.lptr = z.rptr = null

6 y = y.lptr
   y.lptr = y.rptr = null

7 z.sum = x.data + y.data
   x = y = z = null
```

While loop is not executed even once
Our Solution

\[
y = z = \text{null}
\]
1. \(w = x\)
   \(w = \text{null}\)
2. \(\text{while} (x.\text{data} < \text{max})\)
   \[
   \{ \\
   x.\text{lptr} = \text{null} \\
   x = x.\text{rptr} \\
   \}
   \]
3. \(x.\text{rptr} = x.\text{lptr}.\text{rptr} = \text{null}\)
   \(x.\text{lptr}.\text{lptr}.\text{lptr} = \text{null}\)
   \(x.\text{lptr}.\text{lptr}.\text{rptr} = \text{null}\)
4. \(y = x.\text{lptr}\)
   \(x.\text{lptr} = y.\text{rptr} = \text{null}\)
   \(y.\text{lptr}.\text{lptr} = y.\text{lptr}.\text{rptr} = \text{null}\)
5. \(z = \text{New class of } z\)
   \(z.\text{lptr} = z.\text{rptr} = \text{null}\)
6. \(y = y.\text{lptr}\)
   \(y.\text{lptr} = y.\text{rptr} = \text{null}\)
7. \(z.\text{sum} = x.\text{data} + y.\text{data}\)
   \(x = y = z = \text{null}\)

*While loop is not executed even once*
Our Solution

1. \( y = z = \text{null} \)

2. \( w = x \)
   \( w = \text{null} \)

3. \( \text{while} (x\.data < \text{max}) \{
      \quad x\.lptr = \text{null}
      \quad x = x\.rptr
   \} \)

4. \( x\.rptr = x\.lptr\.rptr = \text{null} \)

5. \( x\.lptr\.lptr\.lptr = \text{null} \)

6. \( x\.lptr\.lptr\.rptr = \text{null} \)

7. \( y = x\.lptr \)

8. \( x\.lptr = y\.rptr = \text{null} \)

9. \( y\.lptr\.lptr = y\.lptr\.rptr = \text{null} \)

10. \( z = \text{New class of } z \)

11. \( z\.lptr = z\.rptr = \text{null} \)

12. \( y = y\.lptr \)

13. \( y\.lptr = y\.rptr = \text{null} \)

14. \( z\.sum = x\.data + y\.data \)

15. \( x = y = z = \text{null} \)

While loop is not executed even once
Our Solution

1. \( y = z = \text{null} \)
2. \( \text{while} (x.\text{data} < \text{max}) \{
   \text{\quad} x.lptr = \text{null}
   \text{\quad} x = x.rptr
\} \)
3. \( x.\text{rptr} = x.lptr.rptr = \text{null} \)
4. \( y = x.lptr \)
5. \( z = \text{New class of z} \)
6. \( y.lptr.lptr = y.lptr.rptr = \text{null} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
8. \( x = y = z = \text{null} \)

While loop is not executed even once

Stack

Heap
y = z = null

1  w = x
   w = null

2  while (x.data < max)
   { x.lptr = null
     x = x.rptr }
   x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null

3  y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null

4  z = New class
   z.lptr = z.rptr = null

5  y = y.lptr
   y.lptr = y.rptr = null

6  y = y.lptr
   y.lptr = y.rptr = null

7  z.sum = x.data + y.data
   x = y = z = null

While loop is not executed even once
Our Solution

While loop is executed once

1. `w = x
   w = null
2. while (x.data < max)
   {
      x.lptr = null
      x = x.rptr
   }
   x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
3. `y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
4. `z = New class of z
   z.lptr = z.rptr = null
5. `y = y.lptr
   y.lptr = y.rptr = null
6. `z.sum = x.data + y.data
   x = y = z = null
Our Solution

```plaintext
y = z = null
1 w = x
   w = null
2 while (x.data < max)
   {
   x.lptr = null
   x = x.rptr
   }  x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null
3 y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
4 z = New class of z
   z.lptr = z.rptr = null
5 y = y.lptr
   y.lptr = y.rptr = null
6 z.sum = x.data + y.data
7 x = y = z = null
```

While loop is executed twice
Some Observations

y = z = null
1 w = x
   w = null
2 while (x.data < max)
   { x.lptr = null
3      x = x.rptr }  
   x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null
4 y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
5 z = New class_of_z
   z.lptr = z.rptr = null
6 y = y.lptr
   y.lptr = y.rptr = null
7 z.sum = x.data + y.data
   x = y = z = null

Node i is live but link a → i is nullified

Stack
Heap

May 2011

Uday Khedker
Some Observations

New access expressions are created. Can they cause exceptions?

```
y = z = null
1 w = x
   w = null
2 while (x.data < max)
   { x.lptr = null
3       x = x.rptr
   }
   x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null
4 y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
5 z = New class_of_z
   z.lptr = z.rptr = null
6 y = y.lptr
   y.lptr = y.rptr = null
7 z.sum = x.data + y.data
   x = y = z = null
```
An Overview of Heap Reference Analysis

- A reference (called a *link*) can be represented by an *access path*.
  Eg. “\( x \rightarrow lptr \rightarrow rptr \)”

- A link may be accessed in multiple ways

- Setting links to null
  - *Alias Analysis*. Identify all possible ways of accessing a link
  - *Liveness Analysis*. For each program point, identify “dead” links (i.e. links which are not accessed after that program point)
  - *Availability and Anticipability Analyses*. Dead links should be reachable for making null assignment.
  - *Code Transformation*. Set “dead” links to null
Assumptions

For simplicity of exposition

- Java model of heap access
  - Root variables are on stack and represent references to memory in heap.
  - Root variables cannot be pointed to by any reference.

- Simple extensions for C++
  - Root variables can be pointed to by other pointers.
  - Pointer arithmetic is not handled.
Key Idea #1: Access Paths Denote Links

- Root variables: x, y, z
- Field names: rptr, lptr
- Access path: \(x \rightarrow \text{rptr} \rightarrow \text{lptr}\)
  Semantically, sequence of “links”
- Frontier: name of the last link
- Live access path: If the link corresponding to its frontier is used in future
What Makes a Link Live?

Assuming that a statement is the last statement in the program, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

Reading a link for accessing the contents of the corresponding target object:

<table>
<thead>
<tr>
<th>Example</th>
<th>Objects read</th>
<th>Live access paths</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sum = x.rptr.data</code></td>
<td><code>x, O_1, O_2</code></td>
<td><code>x, x\rightarrow\text{rptr}</code></td>
</tr>
<tr>
<td><code>if (x.rptr.data &lt; \text{sum})</code></td>
<td><code>x, O_1, O_2</code></td>
<td><code>x, x\rightarrow\text{rptr}</code></td>
</tr>
</tbody>
</table>
What Makes a Link Live?

Assuming that a statement is the last statement in the program, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

Reading a link for *copying the contents of the corresponding target object*:

<table>
<thead>
<tr>
<th>Example</th>
<th>Objects read</th>
<th>Live access paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y = x.rptr$</td>
<td>$x, O_1$</td>
<td>$x$</td>
</tr>
</tbody>
</table>
What Makes a Link Live?

Assuming that a statement is the last statement in the program, if nullifying a link \textit{read} in the statement can change the semantics of the program, then the link is live.

\textit{Reading a link for \textit{copying the contents} of the corresponding target object:}

<table>
<thead>
<tr>
<th>Example</th>
<th>Objects read</th>
<th>Live access paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y = x.\text{rptr} )</td>
<td>( x, O_1 )</td>
<td>( x )</td>
</tr>
<tr>
<td>( x.\text{lptr} = y )</td>
<td>( x, O_1, y )</td>
<td>( x )</td>
</tr>
</tbody>
</table>
What Makes a Link Live?

Assuming that a statement is the last statement in the program, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

Reading a link for comparing the address of the corresponding target object:

<table>
<thead>
<tr>
<th>Example</th>
<th>Objects read</th>
<th>Live access paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x.lptr == null)</td>
<td>x, O₁</td>
<td>x, x → lptr</td>
</tr>
</tbody>
</table>

May 2011 Uday Khedker
What Makes a Link Live?

Assuming that a statement is the last statement in the program, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

Reading a link for comparing the address of the corresponding target object:

<table>
<thead>
<tr>
<th>Example</th>
<th>Objects read</th>
<th>Live access paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x.lptr == null)</td>
<td>x, $O_1$</td>
<td>x, x $\rightarrow$ lptr</td>
</tr>
<tr>
<td>if (y == x.lptr)</td>
<td>x, $O_1$, y</td>
<td>x, x $\rightarrow$ lptr, y</td>
</tr>
</tbody>
</table>

Stack

Heap

$O_1$

$O_2$

$O_3$
Liveness Analysis

Statement involving memory references

Program

Semantic Information

Live Access Paths
Effect of the statement on the access paths
Live Access Paths
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]

\[ \ldots = x.r.d \]
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]

\[ \ldots = x.r.d \]

\[ \{ x, x \rightarrow r \} \]
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]

\[ \ldots = x.r.d \]

Analysis

\( \{ x, x \rightarrow r \} \)
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]

\[ \ldots = x.r.d \]

Analysis

\[ \{ x, x \rightarrow r \} \]
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]

... = x.r.d

Analysis

\[ \{x, x \rightarrow r\} \]
Key Idea #2: Transfer of Access Paths

- **Generated**
  - Constant: \{x\}
  - Dependent: \{x \rightarrow n, x \rightarrow n \rightarrow r\}

- **Killed**
  - Constant: \{x, x \rightarrow r\}
  - Dependent: \emptyset

- Analysis
  - \{x, x \rightarrow r\}

- \( x \) after the assignment is same as the \( x \rightarrow n \) before the assignment

\[ x = x.n \]

\[ \ldots = x.r.d \]
Key Idea #3: Liveness Closure Under Link Aliasing

\[ x = y \]

\[ \ldots = x.n \]

\[ x \text{ and } y \text{ are node aliases} \]
Key Idea #3 : Liveness Closure Under Link Aliasing

\[ x = y \]

\[ \ldots = x.n \]

\[ x \text{ and } y \text{ are node aliases} \]

\[ x.n \text{ and } y.n \text{ are link aliases} \]
Key Idea #3: Liveness Closure Under Link Aliasing

- \( x = y \)
- \( \ldots = x.n \)
- \( x \) and \( y \) are node aliases
- \( x.n \) and \( y.n \) are link aliases
- \( x \rightarrow n \) is live \( \Rightarrow \) \( y \rightarrow n \) is live
Key Idea #3: Liveness Closure Under Link Aliasing

\[ x = y \]

\[ \ldots = x.n \]

\[ x \text{ and } y \text{ are node aliases} \]
\[ x.n \text{ and } y.n \text{ are link aliases} \]
\[ x \rightarrow n \text{ is live } \Rightarrow y \rightarrow n \text{ is live} \]

Nullifying \( y \rightarrow n \) will have the side effect of nullifying \( x \rightarrow n \)
Explicit and Implicit Liveness

\[ x = y \]

\[ \ldots = x.n \]

\[ x \rightarrow n \text{ is live} \Rightarrow y \rightarrow n \text{ is live} \]
Explicit and Implicit Liveness

\[ x = y \]

\[ \ldots = x.n \]

\[ x \rightarrow n \] is live \( \Rightarrow \) \[ y \rightarrow n \] is live

\[ y \rightarrow n \] is implicitly live
\[ x \rightarrow n \] is explicitly live
Key Idea #4: Explicit Liveness Covers Entire Heap Usage

- Explicit Liveness at $p$
  Liveness purely due to the program beyond $p$.
  The effect of execution before $p$ is not incorporated.
Key Idea #4: Explicit Liveness Covers Entire Heap Usage

- **Explicit Liveness at** \( p \)
  Liveness purely due to the program beyond \( p \).
  The effect of execution before \( p \) is not incorporated.

- **Implicit Liveness at** \( p \)
  Access paths that become live under link alias closure.
Key Idea #4: Explicit Liveness Covers Entire Heap Usage

- Explicit Liveness at $p$
  Liveness purely due to the program beyond $p$. The effect of execution before $p$ is not incorporated.

- Implicit Liveness at $p$
  Access paths that become live under link alias closure.
    - The set of implicitly live access paths may not be prefix closed.
Key Idea #4: Explicit Liveness Covers Entire Heap Usage

- Explicit Liveness at $p$
  Liveness purely due to the program beyond $p$.
  The effect of execution before $p$ is not incorporated.

- Implicit Liveness at $p$
  Access paths that become live under link alias closure.
  - The set of implicitly live access paths may not be prefix closed.
  - These *paths* are not accessed, their frontiers are accessed through some other access path
Key Idea #4: Explicit Liveness Covers Entire Heap Usage

- **Explicit Liveness at** $p$
  Liveness purely due to the program beyond $p$.
The effect of execution before $p$ is not incorporated.

- **Implicit Liveness at** $p$
  Access paths that become live under link alias closure.
  - The set of implicitly live access paths may not be prefix closed.
  - These *paths* are not accessed, their frontiers are accessed through some other access path

---

*Every live link in the heap is the Frontier of some explicitly live access path.*
### Notation for Defining Flow Functions for Explicit Liveness

**base**($\rho_x$) = longest proper prefix of $\rho_x$

**prefixes**($\rho_x$) = \{\rho'_{\rho_x} | \rho'_{\rho_x} is a prefix of $\rho_x$\}

**summary**($S$) = \{\rho_x \rightarrow \ast | \rho_x \in S\}

<table>
<thead>
<tr>
<th>$\rho_x$</th>
<th>frontier($\rho_x$)</th>
<th>base($\rho_x$)</th>
<th>prefixes($\rho_x$)</th>
<th>summary({$\rho_x$})</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \rightarrow n \rightarrow r$</td>
<td>$r$</td>
<td>$x \rightarrow n$</td>
<td>{x, $x \rightarrow n, x \rightarrow n \rightarrow r$}</td>
<td>{x $\rightarrow n \rightarrow r \rightarrow \ast$}</td>
</tr>
<tr>
<td>$x \rightarrow r \rightarrow n$</td>
<td>$n$</td>
<td>$x \rightarrow r$</td>
<td>{x, $x \rightarrow r, x \rightarrow r \rightarrow n$}</td>
<td>{x $\rightarrow r \rightarrow n \rightarrow \ast$}</td>
</tr>
<tr>
<td>$x \rightarrow n$</td>
<td>$n$</td>
<td>$x$</td>
<td>{x, $x \rightarrow n$}</td>
<td>{x $\rightarrow n \rightarrow \ast$}</td>
</tr>
<tr>
<td>$x \rightarrow r$</td>
<td>$r$</td>
<td>$x$</td>
<td>{x, $x \rightarrow r$}</td>
<td>{x $\rightarrow r \rightarrow \ast$}</td>
</tr>
<tr>
<td>x</td>
<td>$\mathcal{E}$</td>
<td>${x}$</td>
<td>{x}</td>
<td>{x $\rightarrow \ast$}</td>
</tr>
</tbody>
</table>

- **empty access path**
- **0 or more occurrences of any field name**
Notation for Defining Flow Functions for Explicit Liveness

\[
\begin{align*}
\text{base}(\rho_x) &= \text{longest proper prefix of } \rho_x \\
\text{prefixes}(\rho_x) &= \{\rho'_x \mid \rho'_x \text{ is a prefixe of } \rho_x\} \\
\text{summary}(S) &= \{\rho_x \mapsto * \mid \rho_x \in S\}
\end{align*}
\]

<table>
<thead>
<tr>
<th>(\rho_x)</th>
<th>(\text{frontier}(\rho_x))</th>
<th>(\text{base}(\rho_x))</th>
<th>(\text{prefixes}(\rho_x))</th>
<th>(\text{summary}({\rho_x}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x \to n \to r)</td>
<td>(r)</td>
<td>(x \to n)</td>
<td>({x, x \to n, x \to n \to r})</td>
<td>({x \to n \to r \to *})</td>
</tr>
<tr>
<td>(x \to r \to n)</td>
<td>(n)</td>
<td>(x \to r)</td>
<td>({x, x \to r, x \to r \to n})</td>
<td>({x \to r \to n \to *})</td>
</tr>
<tr>
<td>(x \to n)</td>
<td>(n)</td>
<td>(x)</td>
<td>({x, x \to n})</td>
<td>({x \to n \to *})</td>
</tr>
<tr>
<td>(x \to r)</td>
<td>(r)</td>
<td>(x)</td>
<td>({x, x \to r})</td>
<td>({x \to r \to *})</td>
</tr>
<tr>
<td>(x)</td>
<td>(x)</td>
<td>(\mathcal{E})</td>
<td>({x})</td>
<td>({x \to *})</td>
</tr>
</tbody>
</table>
## Flow Functions for Explicit Liveness Analysis

<table>
<thead>
<tr>
<th>Statement</th>
<th>ConstKill</th>
<th>DepKill($X$)</th>
<th>ConstGen</th>
<th>DepGen($X$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use $\alpha_y$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\text{prefixes}(\text{base}(\rho_y))$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Use $\alpha_y . d$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\text{prefixes}(\rho_y)$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\alpha_x = \text{new}$</td>
<td>${\rho_x \rightarrow \ast}$</td>
<td>$\emptyset$</td>
<td>$\text{prefixes}(\text{base}(\rho_x))$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\alpha_x = \text{Null}$</td>
<td>${\rho_x \rightarrow \ast}$</td>
<td>$\emptyset$</td>
<td>$\text{prefixes}(\text{base}(\rho_x))$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\alpha_x = \alpha_y$</td>
<td>${\rho_x \rightarrow \ast}$</td>
<td>$\emptyset$</td>
<td>$\text{prefixes}(\text{base}(\rho_x)) \cup \text{prefixes}(\text{base}(\rho_y))$</td>
<td>${\rho_y \rightarrow \sigma \mid \rho_x \rightarrow \sigma \in X}$</td>
</tr>
<tr>
<td>$\text{End}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\text{summary}(\text{Globals})$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>other</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Flow Functions for Explicit Liveness Analysis

<table>
<thead>
<tr>
<th>Statement</th>
<th>ConstKill</th>
<th>DepKill($X$)</th>
<th>ConstGen</th>
<th>DepGen($X$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use $\alpha_y$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>prefixes(base($\rho_y$))</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Use $\alpha_y.d$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>prefixes($\rho_y$)</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\alpha_x = new$</td>
<td>${\rho_x \rightarrow *}$</td>
<td>$\emptyset$</td>
<td>prefixes(base($\rho_x$))</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\alpha_x = Null$</td>
<td>${\rho_x \rightarrow *}$</td>
<td>$\emptyset$</td>
<td>prefixes(base($\rho_x$))</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\alpha_x = \alpha_y$</td>
<td>${\rho_x \rightarrow *}$</td>
<td>$\emptyset$</td>
<td>prefixes(base($\rho_x$)) $\cup$ prefixes(base($\rho_y$))</td>
<td>${\rho_y \rightarrow \sigma \mid \rho_x \rightarrow \sigma \in X}$</td>
</tr>
<tr>
<td>End</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>summary(Globals)</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>other</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
### Flow Functions for Explicit Liveness Analysis

<table>
<thead>
<tr>
<th>Statement</th>
<th>ConstKill</th>
<th>DepKill($X$)</th>
<th>ConstGen</th>
<th>DepGen($X$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use $\alpha_y$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>prefixes(base($\rho_y$))</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Use $\alpha_y.d$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>prefixes($\rho_y$)</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\alpha_x = \text{new}$</td>
<td>${\rho_x \rightarrow *}$</td>
<td>$\emptyset$</td>
<td>prefixes(base($\rho_x$))</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\alpha_x = \text{Null}$</td>
<td>${\rho_x \rightarrow *}$</td>
<td>$\emptyset$</td>
<td>prefixes(base($\rho_x$))</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\alpha_x = \alpha_y$</td>
<td>${\rho_x \rightarrow *}$</td>
<td>$\emptyset$</td>
<td>prefixes(base($\rho_x$)) $\cup$ prefixes(base($\rho_y$))</td>
<td>${\rho_y \rightarrow \sigma \mid \rho_x \rightarrow \sigma \in X}$</td>
</tr>
<tr>
<td>End</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>summary(Globals)</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>other</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

**End of procedure**
## Flow Functions for Handling Procedure Calls in Computing

### Explicit Liveness

<table>
<thead>
<tr>
<th>Statement</th>
<th>ConstKill</th>
<th>DepKill($X$)</th>
<th>ConstGen</th>
<th>DepGen($X$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha_x = f(\alpha_y)$</td>
<td>${\rho_x \rightarrow \ast}$</td>
<td>$\emptyset$</td>
<td>$\text{prefixes}(\text{base}(\rho_x)) \cup \text{prefixes}(\text{base}(\rho_y)) \cup \text{summary}(({\rho_y} \cup \text{Globals}))$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\text{return } \alpha_y$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\text{prefixes}(\text{base}(\rho_y)) \cup \text{summary}(({\rho_y})$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Computing Explicit Liveness Using Sets of Access Paths

\[ x = x.n \]

\[ \{x, x \rightarrow r\} \]

\[ \ldots = x.r.d \]
Computing Explicit Liveness Using Sets of Access Paths

\[ x = x.n \]
\[ \{ x, x \rightarrow r \} \]
\[ ... = x.r.d \]
Computing Explicit Liveness Using Sets of Access Paths

$$x = x.n$$

$$\{x, x \rightarrow r\}$$

$$\ldots = x.r.d$$

Extended with $$r$$
Computing Explicit Liveness Using Sets of Access Paths

\[
\begin{align*}
    x &= x.n \\
    \{x, x \to n, n \to r\} \\
    \{x, x \to r\} \\
    \{x, x \to r\} \\
    \ldots &= x.r.d \\
\end{align*}
\]
Computing Explicit Liveness Using Sets of Access Paths

\[
x = x.n
\]

\[
\{x, x \rightarrow n, x \rightarrow n \rightarrow r\}
\]

\[
\{x, x \rightarrow r\}
\]

\[
\{x, x \rightarrow r\}
\]

\[
\ldots = x.r.d
\]
Anticipability of Heap References: An All Paths problem

\[ x = x.n \]

\[ \{ x, x \rightarrow n, x \rightarrow n \rightarrow r \} \]

\[ \{ x, x \rightarrow r \} \]

\[ \{ x, x \rightarrow r \} \]

\[ \ldots = x.r.d \]
Anticipability of Heap References: An All Paths problem

\[
\begin{align*}
\text{Analysis} & \\
& \quad \{x, x \rightarrow n, x \rightarrow n \rightarrow r\} \\
& \quad \{x, x \rightarrow r\} \cap \{x, x \rightarrow n, x \rightarrow n \rightarrow r\} \\
& \quad \{x, x \rightarrow r\} \\
& \quad \ldots = x.r.d
\end{align*}
\]
Anticipability of Heap References: An *All Paths* problem

\[
\begin{align*}
\text{Analysis} & \quad \{x, x \to n, x \to n \to r\} \\
\{x\} & \quad \{x, x \to r\} \\
\ldots = x \cdot r \cdot d
\end{align*}
\]
Anticipability of Heap References: An *All Paths* problem

\[ x = x.n \]

\[ \{x\} \]

\[ \{x, x \rightarrow r\} \]

\[ \ldots = x.r.d \]
Liveness of Heap References: An *Any Path* problem

\[
x = x.n
\]

\[
\{x, x \rightarrow n, x \rightarrow n \rightarrow r\}
\]

\[
\{x, x \rightarrow r\}
\]

\[
\{x, x \rightarrow r\}
\]

\[
\ldots = x.r.d
\]
Liveness of Heap References: An Any Path problem

\[
x = x.n
\]

\[
\{x, x \rightarrow r\} \cup \{x, x \rightarrow n, x \rightarrow n \rightarrow r\}
\]

\[
\{x, x \rightarrow r\}
\]

\[
\ldots = x.r.d
\]
Liveness of Heap References: An *Any Path* problem

\[ x \rightarrow n \text{ extended with } r, n, \text{ and } n \rightarrow r \]

\[ \{x, x \rightarrow r, x \rightarrow n, x \rightarrow n \rightarrow r\} \]

\[ \{x, x \rightarrow r\} \]

\[ \ldots = x.\_d \]
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem

\[ \{x, x \rightarrow n, x \rightarrow n \rightarrow n, x \rightarrow n \rightarrow r, x \rightarrow n \rightarrow n \rightarrow r\} \]

\[ \{x, x \rightarrow r, x \rightarrow n, x \rightarrow n \rightarrow r\} \]

\[ \{x, x \rightarrow r\} \]

\[ \ldots = x \cdot r \cdot d \]
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem

$$x = x.n$$

- $$\{x, x \rightarrow n, x \rightarrow n \rightarrow r, x \rightarrow n \rightarrow n \rightarrow r, x \rightarrow n \rightarrow \cdots \rightarrow n \rightarrow r\}$$
- $$\{x, x \rightarrow r, x \rightarrow n, x \rightarrow n \rightarrow r, x \rightarrow n \rightarrow \cdots \rightarrow n \rightarrow r\}$$
- $$\{x, x \rightarrow r\}$$
- $$\ldots = x.r.d$$

*Infinite Number of Unbounded Access Paths*
Key Idea #5: Using Graphs as Data Flow Values

Finite Number of Bounded Structures
Key Idea #6: Include Program Point in Graphs

1. \( x = x.n \)

\[ \{ x, x \rightarrow n, x \rightarrow n \rightarrow n, x \rightarrow n \rightarrow n \rightarrow n, \ldots \} \]

Different occurrences of \( n \)'s in an access path are Indistinguishable

2. \( x = x.n \)

\[ \{ x, x \rightarrow n, x \rightarrow n \rightarrow n, x \rightarrow n \rightarrow n \rightarrow r \} \]

Different occurrences of \( n \)'s in an access path are Distinct
Key Idea #6: Include Program Point in Graphs

\[
\{ x, x \rightarrow n, x \rightarrow n \rightarrow n, x \rightarrow n \rightarrow n \rightarrow n, \ldots \} 
\]

Different occurrences of n’s in an access path are \textbf{Indistinguishable}

Access Graph:

\[
\text{x} \rightarrow n \rightarrow n_1 \rightarrow n \rightarrow n_2 \rightarrow r \rightarrow r_2
\]
Key Idea #6: Include Program Point in Graphs

1. \( x = x.n \)

\( \{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow n, \ldots\} \)

Different occurrences of n’s in an access path are **Indistinguishable**

Access Graph: \( x \rightarrow n \rightarrow n_1 \rightarrow n \)

2. \( \ldots = x.n.r.d \)

\( \{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow r\} \)

Different occurrences of n’s in an access path are **Distinct**

Access Graph: \( x \rightarrow n \rightarrow n_1 \rightarrow n_2 \rightarrow r \rightarrow r_2 \)
Inclusion of Program Point Facilitates Summarization

1

2

= x.n.d

3

x = x.r

4

= x.n.d
Inclusion of Program Point Facilitates Summarization

\[ x = x.r \]

\[ x = x.n.d \]

\[ G_4 x \rightarrow n n_4 \]

\[ G_4 x \rightarrow n n_4 \]
Inclusion of Program Point Facilitates Summarization
Inclusion of Program Point Facilitates Summarization

\[ x = x.n.d \]

\[ x = x.n.d \]

\[ x = x.r \]

\[ x = x.n.d \]

May 2011

Uday Khedker
Inclusion of Program Point Facilitates Summarization

\[ G_1 = G_2 \cup G_3 \]
Inclusion of Program Point Facilitates Summarization

Iteration #1

Analysis

1. $x = x.n$

2. $\ldots = x.r.d$
Inclusion of Program Point Facilitates Summarization

Iteration #1

1. \( x = x.n \)

2. \( \ldots = x.r.d \)

\( \xrightarrow{r} r_2 \)
Inclusion of Program Point Facilitates Summarization

Iteration #1

1. $x = x.n$
2. $\ldots = x.r.d$

May 2011 Uday Khedker
Inclusion of Program Point Facilitates Summarization

Iteration #1

1. $x = x.n$

2. $\ldots = x.r.d$

Analysis

May 2011 Uday Khedker
Inclusion of Program Point Facilitates Summarization

Iteration #1

Analysis

1. \( x = x.n \)

2. \( \ldots = x.r.d \)

\[ x \rightarrow n \rightarrow n_1 \rightarrow r \rightarrow r_2 \]

\[ x \rightarrow r \rightarrow r_2 \]

\[ x \rightarrow r \rightarrow r_2 \]
Inclusion of Program Point Facilitates Summarization

Iteration #2

1. \( x = x.n \)

2. \( \ldots = x.r.d \)
Inclusion of Program Point Facilitates Summarization

Iteration #2

May 2011
Inclusion of Program Point Facilitates Summarization

Analysis

1. $x = x \cdot n$

2. $\ldots = x \cdot r \cdot d$

Iteration #2

May 2011 Uday Khedker
Inclusion of Program Point Facilitates Summarization

Iteration #2

May 2011 Uday Khedker
Inclusion of Program Point Facilitates Summarization

Iteration #3

1. $x = x.n$
2. $\ldots = x.r.d$

$\bigcup_G x \rightarrow n_1 \rightarrow r_2 \rightarrow n \\ n_1 \rightarrow r_2$
Inclusion of Program Point Facilitates Summarization

Analysis

1 $x = x.n$

2 $\ldots = x.r.d$

Iteration #3

1 $n \rightarrow n_1 \rightarrow r_2$

2 $x \rightarrow n_1 \rightarrow r_2$

3 $x \rightarrow r_2$

May 2011 Uday Khedker
Inclusion of Program Point Facilitates Summarization

1. $x = x.n$

2. $\ldots = x.r.d$

Iteration #3

```
x n1 n1 r2
x n1 r2
x r2
```

May 2011
Uday Khedker
Inclusion of Program Point Facilitates Summarization

Iteration #3

1. $x = x.n$

2. $\ldots = x.r.d$

May 2011 Uday Khedker
Access Graph and Memory Graph

Program Fragment

1

\[ x.l = y.r \]

2

\[ \text{if } (x.l.n == y.r.n) \]
Access Graph and Memory Graph

Program Fragment

\[ x.l = y.r \]

1

if \((x.l.n == y.r.n)\)

2

Memory Graph

\[ x \rightarrow l \]
\[ y \rightarrow r \]
\[ n \rightarrow \]

May 2011

Uday Khedker
Access Graph and Memory Graph

Program Fragment

\[ x.l = y.r \]

if \((x.l.n == y.r.n)\)

Memory Graph

[Diagram of Access Graphs]

Access Graphs
**Access Graph and Memory Graph**

Program Fragment

\[ x.l = y.r \]

1

\[ \text{if } (x.l.n == y.r.n) \]

2

- **Memory Graph**: Captures the shape of heap
  - Nodes represent locations and edges represent links (i.e. pointers).
Access Graph and Memory Graph

Program Fragment

```
x.l = y.r
```

1

```
if (x.l.n == y.r.n)
```

2

- **Memory Graph:** Captures the shape of heap
  Nodes represent locations and edges represent links (i.e. pointers).

- **Access Graphs:** Captures the usage (or access) pattern of heap
  Nodes represent dereference of links at particular statements. Memory locations are implicit.
Lattice of Access Graphs

- Finite number of nodes in an access graph for a variable
- \( \sqcap \) induces a partial order on access graphs
  \( \Rightarrow \) a finite (and hence complete) lattice
  \( \Rightarrow \) All standard results of classical data flow analysis can be extended to this analysis.

*Termination and boundedness, convergence on MFP, complexity etc.*
Access Graph Operations

- **Union.** $G \cup G'$
- **Path Removal.** $G \ominus \rho$ removes those access paths in $G$ which have $\rho$ as a prefix.
- **Factorization (/).**
- **Extension.**
### Semantics of Access Graph Operations

- \( P(G, M) \) is the set of paths in graph \( G \) terminating on nodes in \( M \). For graph \( G_i, M_i \) is the set of all nodes in \( G_i \).
- \( S \) is the set of remainder graphs and \( P(S, M_s) \) is the set of all paths in all remainder graphs in \( S \).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Access Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union</td>
<td>( P(G_3, M_3) \supseteq P(G_1, M_1) \cup P(G_2, M_2) )</td>
</tr>
<tr>
<td>Path Removal</td>
<td>( P(G_2, M_2) \supseteq P(G_1, M_1) - { \rho \rightarrow\sigma \mid \rho \rightarrow\sigma \in P(G_1, M_1) } )</td>
</tr>
<tr>
<td>Factorization</td>
<td>( P(S, M_s) = { \sigma \mid \rho' \rightarrow\sigma \in P(G_1, M_1), \rho' \in P(G_2, M) } )</td>
</tr>
<tr>
<td>Extension</td>
<td>( P(G_2, M_2) = \emptyset )</td>
</tr>
<tr>
<td></td>
<td>( P(G_2, M_2) \supseteq P(G_1, M_1) \cup { \rho \rightarrow\sigma \mid \rho \in P(G_1, M), \sigma \in P(S, M_s) } )</td>
</tr>
</tbody>
</table>
Semantics of Access Graph Operations

- $P(G, M)$ is the set of paths in graph $G$ terminating on nodes in $M$. For graph $G_i$, $M_i$ is the set of all nodes in $G_i$.

- $S$ is the set of remainder graphs and $P(S, M_s)$ is the set of all paths in all remainder graphs in $S$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Access Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union</td>
<td>$G_3 = G_1 \cup G_2$</td>
</tr>
<tr>
<td></td>
<td>$P(G_3, M_3) \supseteq P(G_1, M_1) \cup P(G_2, M_2)$</td>
</tr>
<tr>
<td>Path Removal</td>
<td>$G_2 = G_1 \ominus \rho$</td>
</tr>
<tr>
<td></td>
<td>$P(G_2, M_2) \supseteq P(G_1, M_1) \setminus {\rho \rightarrow \sigma</td>
</tr>
<tr>
<td>Factorization</td>
<td>$S = G_1/(G_2, M)$</td>
</tr>
<tr>
<td></td>
<td>$P(S, M_s) = {\sigma</td>
</tr>
<tr>
<td>Extension</td>
<td>$G_2 = (G_1, M) # \emptyset$</td>
</tr>
<tr>
<td></td>
<td>$P(G_2, M_2) = \emptyset$</td>
</tr>
<tr>
<td></td>
<td>$P(G_2, M_2) \supseteq P(G_1, M_1) \cup {\rho \rightarrow \sigma</td>
</tr>
</tbody>
</table>

$\sigma$ represents remainder

$\rho'$ represents quotient
Access Graph Operations: Examples

Program

1. \( x = x.l \)

2. \( y = x.r.d \)

Access Graphs

<table>
<thead>
<tr>
<th>g1</th>
<th>g2</th>
<th>g3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( x \rightarrow r_2 \rightarrow l_1 \rightarrow r_2 )</td>
<td>( x \rightarrow l_1 \rightarrow r_2 )</td>
</tr>
</tbody>
</table>

Remainder Graphs

<table>
<thead>
<tr>
<th>rg1</th>
<th>rg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_2 \rightarrow l_1 \rightarrow r_2 )</td>
<td>( l_1 \rightarrow r_2 )</td>
</tr>
</tbody>
</table>

Union | Path Removal | Factorisation | Extension
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

May 2011
Uday Khedker
## Access Graph Operations: Examples

### Program

<table>
<thead>
<tr>
<th>Program</th>
<th>Access Graphs</th>
<th>Remainder Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( x = x.l )</td>
<td>( g_1 \rightarrow x )</td>
<td>( rg_1 \rightarrow r_2 )</td>
</tr>
<tr>
<td>2 ( y = x.r.d )</td>
<td>( g_4 \rightarrow x \rightarrow l_1 \rightarrow r_2 )</td>
<td>( rg_2 \rightarrow l_1 \rightarrow r_2 )</td>
</tr>
</tbody>
</table>

### Operations

<table>
<thead>
<tr>
<th>Union</th>
<th>Path Removal</th>
<th>Factorisation</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g_3 \uplus g_4 = g_4 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( g_2 \uplus g_4 = g_5 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( g_5 \uplus g_4 = g_5 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( g_5 \uplus g_6 = g_6 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Access Graph Operations: Examples**

<table>
<thead>
<tr>
<th>Program</th>
<th>Access Graphs</th>
<th>Remainder Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( x = x.l )</td>
<td>( g_1 \rightarrow x )</td>
<td>( rg_1 \rightarrow r_2 )</td>
</tr>
<tr>
<td>2 ( y = x.r.d )</td>
<td>( \begin{align*} g_4 &amp; \rightarrow x \rightarrow l_1 \rightarrow r_2 \ g_5 &amp; \rightarrow x \rightarrow l_1 \rightarrow r_2 \ g_6 &amp; \rightarrow x \rightarrow l_1 \rightarrow r_2 \end{align*} )</td>
<td>( \begin{align*} rg_2 &amp; \rightarrow l_1 \rightarrow r_2 \end{align*} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Union</th>
<th>Path Removal</th>
<th>Factorisation</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g_3 \cup g_4 = g_4 )</td>
<td>( g_6 \ominus x \rightarrow l = g_2 )</td>
<td>( g_6 \ominus x \rightarrow l = g_1 )</td>
<td></td>
</tr>
<tr>
<td>( g_2 \cup g_4 = g_5 )</td>
<td>( g_5 \ominus x = \mathcal{E}_G )</td>
<td>( g_4 \ominus x \rightarrow r = g_4 )</td>
<td></td>
</tr>
<tr>
<td>( g_5 \cup g_4 = g_5 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( g_5 \cup g_6 = g_6 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Access Graph Operations: Examples

<table>
<thead>
<tr>
<th>Program</th>
<th>Access Graphs</th>
<th>Remainder Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $x = x.l$</td>
<td>$g_1 \Rightarrow x$</td>
<td>$rg_1 \Rightarrow r_2$</td>
</tr>
<tr>
<td>2 $y = x.r.d$</td>
<td>$g_4 \Rightarrow x \rightarrow l_1 \rightarrow r_2$</td>
<td>$rg_2 \Rightarrow l_1 \rightarrow r_2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Union</th>
<th>Path Removal</th>
<th>Factorisation</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_3 \cup g_4 = g_4$</td>
<td>$g_6 \ominus x \rightarrow l = g_2$</td>
<td>$g_2 / (g_1, {x}) = {rg_1}$</td>
<td></td>
</tr>
<tr>
<td>$g_2 \cup g_4 = g_5$</td>
<td>$g_5 \ominus x = \mathcal{E}_G$</td>
<td>$g_5 / (g_1, {x}) = {rg_1, , \text{rg}_2}$</td>
<td></td>
</tr>
<tr>
<td>$g_5 \cup g_4 = g_5$</td>
<td>$g_4 \ominus x \rightarrow r = g_4$</td>
<td>$g_5 / (g_2, {r_2}) = {\epsilon_{RG}}$</td>
<td></td>
</tr>
<tr>
<td>$g_5 \cup g_6 = g_6$</td>
<td>$g_4 \ominus x \rightarrow l = g_1$</td>
<td>$g_4 / (g_2, {r_2}) = \emptyset$</td>
<td></td>
</tr>
</tbody>
</table>
Access Graph Operations: Examples

<table>
<thead>
<tr>
<th>Program</th>
<th>Access Graphs</th>
<th>Remainder Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (x = x.l)</td>
<td>(g_1) (\Rightarrow x)</td>
<td>(rg_1) (\Rightarrow r_2)</td>
</tr>
<tr>
<td>2 (y = x.r.d)</td>
<td>(g_4) (\Rightarrow x \rightarrow l_1 \rightarrow r_2)</td>
<td>(rg_2) (\Rightarrow l_1 \rightarrow r_2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Union</th>
<th>Path Removal</th>
<th>Factorisation</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>(g_3 \uplus g_4 = g_4)</td>
<td>(g_6 \uplus x \rightarrow l = g_2)</td>
<td>(g_2/ (g_1, {x}) = {rg_1})</td>
<td>((g_3, {l_1}) \not# {rg_1} = g_4)</td>
</tr>
<tr>
<td>(g_2 \uplus g_4 = g_5)</td>
<td>(g_5 \uplus x = \mathcal{E}_G)</td>
<td>(g_5/ (g_1, {x}) = {rg_1,) (rg_2})</td>
<td>((g_3, {x, l_1}) \not# {rg_1, rg_2} = g_6)</td>
</tr>
<tr>
<td>(g_5 \uplus g_4 = g_5)</td>
<td>(g_4 \uplus x \rightarrow r = g_4)</td>
<td>(g_5/ (g_2, {r_2}) = {\epsilon_{RG}})</td>
<td>((g_2, {r_2}) \not# {\epsilon_{RG}} = g_2)</td>
</tr>
<tr>
<td>(g_5 \uplus g_6 = g_6)</td>
<td>(g_4 \uplus x \rightarrow l = g_1)</td>
<td>(g_4/ (g_2, {r_2}) = \emptyset)</td>
<td>((g_2, {r_2}) \not# \emptyset = \mathcal{E}_G)</td>
</tr>
</tbody>
</table>
# Access Graph Operations: Examples

## Program Access Graphs

<table>
<thead>
<tr>
<th>Program</th>
<th>Access Graphs</th>
<th>Remainder Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = x.l$</td>
<td>$g_1 \xrightarrow{x} x$</td>
<td>$rg_1 \xrightarrow{r_2}$</td>
</tr>
<tr>
<td>$y = x.r.d$</td>
<td>$g_4 \xrightarrow{x} l_1 \xrightarrow{r_2}$</td>
<td>$rg_2 \xrightarrow{l_1} r_2$</td>
</tr>
</tbody>
</table>

## Operations

<table>
<thead>
<tr>
<th>Union</th>
<th>Path Removal</th>
<th>Factorisation</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_3 \cup g_4 = g_4$</td>
<td>$g_6 \oplus x \rightarrow l = g_2$</td>
<td>$g_2 / (g_1, {x}) = {rg_1}$</td>
<td>$(g_3, {l_1}) # {rg_1} = g_4$</td>
</tr>
<tr>
<td>$g_2 \cup g_4 = g_5$</td>
<td>$g_5 \oplus x = \epsilon_G$</td>
<td>$g_5 / (g_1, {x}) = {rg_1, \rg_2}$</td>
<td>$(g_3, {x, l_1}) # {rg_1, \rg_2} = g_6$</td>
</tr>
<tr>
<td>$g_5 \cup g_4 = g_5$</td>
<td>$g_4 \oplus x \rightarrow r = g_4$</td>
<td>$g_5 / (g_2, {r_2}) = {\epsilon_{RG}}$</td>
<td>$(g_2, {r_2}) # {\epsilon_{RG}} = g_2$</td>
</tr>
<tr>
<td>$g_5 \cup g_6 = g_6$</td>
<td>$g_4 \oplus x \rightarrow l = g_1$</td>
<td>$g_4 / (g_2, {r_2}) = \emptyset$</td>
<td>$(g_2, {r_2}) # \emptyset = \emptyset_G$</td>
</tr>
</tbody>
</table>

- **Remainder is empty**
- **Quotient is empty**
Data Flow Equations for Heap Liveness Analysis

Computing Liveness Access Graph for variable $v$ by incorporating the effect of statement $n$.

\[
ELIn_n(v) = (ELOut_n(v) \ominus ELKillPath_n(v)) \uplus ELGen_n(v)
\]

\[
ELOut_n(v) = \begin{cases} 
    makeGraph(v \rightarrow *) & n = \text{End}, \ v \in \text{Globals} \\
    \mathcal{E}_G \uplus \bigcup_{s \in \text{succ}(n)} ELIn_s(v) & \text{otherwise}
\end{cases}
\]

\[
ELGen_n(v) = ELConstGen_n(v) \uplus ELDepGen_n(v)
\]

(Note: This notation is slightly different from the notation in the book.)
# Flow Functions for Explicit Liveness Analysis

<table>
<thead>
<tr>
<th>Access Paths</th>
<th>Access Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong> $\alpha_x.d$</td>
<td><strong>Use</strong> $\alpha_x$</td>
</tr>
<tr>
<td>$ConstKill_n$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$ConstGen_n$</td>
<td>$\text{prefixes}(\rho_x)$</td>
</tr>
<tr>
<td>$DepGen_n(X)$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

$$
G_x = \text{makeGraph}(\rho_x) \quad G^B_x = \text{makeGraph}(\text{base}(\rho_x)) \\
G_y = \text{makeGraph}(\rho_y) \quad G^B_y = \text{makeGraph}(\text{base}(\rho_y))
$$

<table>
<thead>
<tr>
<th>$Use$ $\alpha_x.d$</th>
<th>$Use$ $\alpha_x$</th>
<th>$\alpha_x = \alpha_y$</th>
<th>$\alpha_x = \text{Null}$, $\alpha_x = \text{New}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ELKillPath_n(x)$</td>
<td>$\mathcal{E}$</td>
<td>$\mathcal{E}$</td>
<td>$\rho_x$</td>
</tr>
<tr>
<td>$ELKillPath_n(y)$</td>
<td>$\mathcal{E}$</td>
<td>$\mathcal{E}$</td>
<td>$\mathcal{E}$</td>
</tr>
<tr>
<td>$ELConstGen_n(x)$</td>
<td>$G_x$</td>
<td>$G^B_x$</td>
<td>$G^B_x$</td>
</tr>
<tr>
<td>$ELConstGen_n(y)$</td>
<td>$\mathcal{E}_G$</td>
<td>$\mathcal{E}_G$</td>
<td>$G^B_y$</td>
</tr>
<tr>
<td>$ELDepGen_n(x)(X)$</td>
<td>$\mathcal{E}_G$</td>
<td>$\mathcal{E}_G$</td>
<td>$\mathcal{E}_G$</td>
</tr>
<tr>
<td>$ELDepGen_n(y)(X)$</td>
<td>$\mathcal{E}_G$</td>
<td>$\mathcal{E}_G$</td>
<td>$(G_y, M_y) # (X/(G_x, M_x))$</td>
</tr>
</tbody>
</table>
### Flow Functions for Explicit Liveness Analysis

<table>
<thead>
<tr>
<th>Access Paths</th>
<th>Access Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{ConstKill}_n$</td>
<td>$\text{ConstGen}_n$</td>
</tr>
<tr>
<td>$\text{DepGen}_n(X)$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{ConstKill}_n$</td>
<td>$\text{prefixes}(\rho_x)$</td>
</tr>
<tr>
<td>$\text{ConstGen}_n$</td>
<td>$\text{prefixes}(\text{base}(\rho_x)) \cup \text{prefixes}(\text{base}(\rho_y))$</td>
</tr>
<tr>
<td>$\text{DepGen}_n(X)$</td>
<td>${ \rho_y \rightarrow \sigma \mid \rho_x \rightarrow \sigma \in X }$</td>
</tr>
</tbody>
</table>

$G_x = \text{makeGraph}(\rho_x)$

$G^B_x = \text{makeGraph}(\text{base}(\rho_x))$

$G^B_y = \text{makeGraph}(\text{base}(\rho_y))$

<table>
<thead>
<tr>
<th>$\text{ELKillPath}_n(x)$</th>
<th>$\mathcal{E}$</th>
<th>$\mathcal{E}$</th>
<th>$\rho_x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{ELKillPath}_n(y)$</td>
<td>$\mathcal{E}$</td>
<td>$\mathcal{E}$</td>
<td>$\rho_x$</td>
</tr>
<tr>
<td>$\text{ELConstGen}_n(x)$</td>
<td>$G_x$</td>
<td>$G^B_x$</td>
<td>$G^B_x$</td>
</tr>
<tr>
<td>$\text{ELConstGen}_n(y)$</td>
<td>$\mathcal{E}_G$</td>
<td>$\mathcal{E}_G$</td>
<td>$G^B_y$</td>
</tr>
<tr>
<td>$\text{ELDepGen}_n(x)(X)$</td>
<td>$\mathcal{E}_G$</td>
<td>$\mathcal{E}_G$</td>
<td>$\mathcal{E}_G$</td>
</tr>
<tr>
<td>$\text{ELDepGen}_n(y)(X)$</td>
<td>$\mathcal{E}_G$</td>
<td>$\mathcal{E}_G$</td>
<td>$(G_y, M_y) # (X / (G_x, M_x))$</td>
</tr>
</tbody>
</table>

The singleton set containing the last node corresponding to $\rho_x$.

The singleton set containing the last node corresponding to $\rho_y$.

$\alpha_x = \alpha_y$

$\alpha_x = \text{Null}$, $\alpha_x = \text{New}$
Liveness Analysis of Example Program: 1st Iteration

1. \( w = x \)

2. while \( (x.\text{data} < \text{max}) \)

3. \( x = x.\text{rptr} \)

4. \( y = x.\text{lptr} \)

5. \( z = \text{New class of } z \)

6. \( y = y.\text{lptr} \)

7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
Liveness Analysis of Example Program: 2nd Iteration

1. \( w = x \)

2. while \( (x.data < max) \)

3. \( x = x.rptr \)

4. \( y = x.lptr \)

5. \( z = \text{New class of } z \)

6. \( y = y.lptr \)

7. \( z.sum = x.data + y.data \)
Liveness Analysis of Example Program: 3rd Iteration

1. \( w = x \)
2. \( \text{while } (x.data < \text{max}) \)
3. \( x = x.rptr \)
4. \( y = x.lptr \)
5. \( z = \text{New class of } z \)
6. \( y = y.lptr \)
7. \( z.sum = x.data + y.data \)
Liveness Analysis of Example Program: 4th Iteration

1. \( w = x \)

2. \( \text{while} \ (x.\text{data} < \text{max}) \)

3. \( x = x.\text{rptr} \)

4. \( y = x.\text{lptr} \)

5. \( z = \text{New class of } z \)

6. \( y = y.\text{lptr} \)

7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
Which Access Paths Can be Nullified?

• Consider extensions of accessible paths for nullification.

Let $\rho$ be accessible at $p$ (i.e. available or anticipable) for each reference field $f$ of the object pointed to by $\rho$

if $\rho \rightarrow f$ is not live at $p$ then

Insert $\rho \rightarrow f = \text{null}$ at $p$ subject to profitability

• For simple access paths, $\rho$ is empty and $f$ is the root variable name.
Which Access Paths Can be Nullified?

- Consider extensions of accessible paths for nullification.

Let $\rho$ be accessible at $p$ (i.e. available or anticipable) for each reference field $f$ of the object pointed to by $\rho$ if $\rho \to f$ is not live at $p$ then

Insert $\rho \to f = \text{null}$ at $p$ subject to profitability

- For simple access paths, $\rho$ is empty and $f$ is the root variable name.
Which Access Paths Can be Nullified?

- Consider extensions of accessible paths for nullification.

  Let $\rho$ be accessible at $p$ (i.e., available or anticipable) for each reference field $f$ of the object pointed to by $\rho$.
  
  **If** $\rho \rightarrow f$ is not live at $p$ **then**
  
  Insert $\rho \rightarrow f = \text{null}$ at $p$ subject to profitability.

- For simple access paths, $\rho$ is empty and $f$ is the root variable name.
Which Access Paths Can be Nullified?

- Consider extensions of accessible paths for nullification.

Let $\rho$ be accessible at $p$ (i.e., available or anticipable)
for each reference field $f$ of the object pointed to by $\rho$
if $\rho \rightarrow f$ is not live at $p$ then

Insert $\rho \rightarrow f = \text{null}$ at $p$ subject to profitability

- For simple access paths, $\rho$ is empty and $f$ is the root variable name.
Availability and Anticipability Analyses

- $\rho$ is **available** at program point $p$ if the target of each prefix of $\rho$ is guaranteed to be created along every control flow path reaching $p$.

- $\rho$ is **anticipable** at program point $p$ if the target of each prefix of $\rho$ is guaranteed to be dereferenced along every control flow path starting at $p$. 
Availability and Anticipability Analyses

- $\rho$ is available at program point $p$ if the target of each prefix of $\rho$ is guaranteed to be created along every control flow path reaching $p$.
- $\rho$ is anticipable at program point $p$ if the target of each prefix of $\rho$ is guaranteed to be dereferenced along every control flow path starting at $p$.

Finiteness.

- An anticipable (available) access path must be anticipable (available) along every path. Thus unbounded paths arising out of loops cannot be anticipable (available).
- Due to “every control flow path nature”, computation of anticipable and available access paths uses $\cap$ as the confluence. Thus the sets are bounded.

$\Rightarrow$ No need of access graphs.
Availability Analysis of Example Program

1. \( w = x \)

2. \( \text{while} \ (x.\text{data} < \text{max}) \)

3. \( x = x.\text{rptr} \)

4. \( y = x.\text{lptr} \)

5. \( z = \text{New class of z} \)

6. \( y = y.\text{lptr} \)

7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
Anticipability Analysis of Example Program

1. \( w = x \)

2. \( \text{while (x.data < max)} \)

3. \( x = x.rptr \)

4. \( y = x.lptr \)

5. \( z = \text{New class_of_z} \)

6. \( y = y.lptr \)

7. \( z.sum = x.data + y.data \)

\[ \{x\} \]
\[ \{x\} \]
\[ \{x\} \]
\[ \{x\} \]
\[ \{x, x\rightarrow rptr\} \]
\[ \{x\} \]
\[ \{x, x\rightarrow lptr, x\rightarrow lptr, x\rightarrow lptr\} \]
\[ \{x, y\rightarrow lptr\} \]
\[ \{x, y, y\rightarrow lptr\} \]
\[ \{x, y, y\rightarrow lptr, z\} \]
\[ \{x, y, z\} \]
\[ \{x\} \]
\[ \{x\} \]
\[ \{x, x\rightarrow rptr\} \]
\[ \{x\} \]
\[ \{x, x\rightarrow lptr, x\rightarrow lptr, x\rightarrow lptr\} \]
\[ \{x\} \]
\[ \emptyset \]
Live and Accessible Paths

1. \( w = x \)

2. \( \text{while (x.data < max)} \)

3. \( x = x.rptr \)

4. \( y = x.lptr \)

5. \( z = \text{New class of z} \)

6. \( y = y.lptr \)

7. \( z.sum = x.data + y.data \)

May 2011
Creating null Assignments from Live and Accessible Paths

1. \( w = x \)

2. \( \text{while } (x.data < \text{max}) \)
   - \( x.rptr = x.lptr.rptr = \text{null} \)
   - \( x.lptr.lptr.lptr = \text{null} \)
   - \( x.lptr.lptr.rptr = \text{null} \)  
   - \( x.lptr = \text{null} \)

3. \( x = x.rptr \)

4. \( y = x.lptr \)

5. \( z = \text{New class of } z \)
   - \( z.lptr = z.rptr = \text{null} \)

6. \( y = y.lptr \)
   - \( y.lptr = y.rptr = \text{null} \)

7. \( z.sum = x.data + y.data \)

May 2011

Uday Khedker
The Resulting Program

1. \( w = x \)
2. \( w = null \)
3. \( x = x.rptr \)
4. \( x.rptr = x.lptr.rptr = null \)
5. \( x.lptr.lptr.lptr = null \)
6. \( x.lptr.lptr.rptr = null \)
7. \( y = x.lptr \)
8. \( x.lptr = y.rptr = null \)
9. \( y.lptr.lptr = y.lptr.rptr = null \)
10. \( z = New \ class\_of\_z \)
11. \( z.lptr = z.rptr = null \)
12. \( y = y.lptr \)
13. \( y.lptr = y.rptr = null \)
14. \( z.sum = x.data + y.data \)
15. \( x = y = z = null \)

May 2011
Uday Khedker
Non-Distributivity of Explicit Liveness Analysis

1. $x.n = \text{null}$

2. $x = x.n$

3. $x.n.n = \text{null}$

4. $x = x.r$

5. $x.n.r = \text{null}$

6. $x = x.n$

7. $z = x.n$

8. $z = x.r$
Non-Distributivity of Explicit Liveness Analysis

1. \( x.n = \text{null} \)

2. \( x = x.n \)

3. \( x.n.n = \text{null} \)

4. \( x = x.r \)

5. \( x.n.r = \text{null} \)

6. \( x = x.n \)

7. \( z = x.n \)

8. \( z = x.r \)
Non-Distributivity of Explicit Liveness Analysis

1. \( x.n = \text{null} \)

2. \( x = x.n \)

3. \( x.n.n = \text{null} \)

4. \( x = x.r \)

5. \( x.n.r = \text{null} \)

6. \( x = x.n \)

7. \( z = x.n \)

8. \( z = x.r \)

May 2011

Uday Khedker
Non-Distributivity of Explicit Liveness Analysis

1. \( x.n = \text{null} \)

2. \( x = x.n \)

3. \( x.n.n = \text{null} \)

4. \( x = x.r \)

5. \( x.n.r = \text{null} \)

6. \( x = x.n \)

7. \( z = x.n \)

8. \( z = x.r \)

May 2011

Uday Khedker
Non-Distributivity of Explicit Liveness Analysis

1. $x.n = \text{null}$
2. $x = x.n$
3. $x.n.n = \text{null}$
4. $x = x.r$
5. $x.n.r = \text{null}$
6. $x = x.n$
7. $z = x.n$
8. $z = x.r$
Non-Distributivity of Explicit Liveness Analysis

1. $x.n = \text{null}$

2. $x = x.n$

3. $x.n.n = \text{null}$

4. $x = x.r$

5. $x.n.r = \text{null}$

6. $x = x.n$

7. $z = x.n$

8. $z = x.r$
Non-Distributivity of Explicit Liveness Analysis

1. $x.n = \text{null}$

2. $x = x.n$

3. $x.n.n = \text{null}$

4. $x = x.r$

5. $x.n.r = \text{null}$

6. $x = x.n$

7. $z = x.n$

8. $z = x.r$
Non-Distributivity of Explicit Liveness Analysis

1. $x.n = \text{null}$
2. $x = x.n$
3. $x.n.n = \text{null}$
4. $x = x.r$
5. $x.n.r = \text{null}$
6. $x = x.n$
7. $z = x.n$
8. $z = x.r$
Non-Distributivity of Explicit Liveness Analysis

1. $x.n = \text{null}$
2. $x = x.n$
3. $x.n.n = \text{null}$
4. $x = x.r$
5. $x.n.r = \text{null}$
6. $x = x.n$
7. $z = x.n$
8. $z = x.r$
Non-Distributivity of Explicit Liveness Analysis

1. $x.n = \text{null}$

2. $x = x.n$

3. $x.n.n = \text{null}$

4. $x.n.r = \text{null}$

ELOut$_1(x)$

$f_1(ELIn_2(x) \cup ELIn_4(x))$

May 2011  Uday Khedker
Non-Distributivity of Explicit Liveness Analysis

1. $x.n = \text{null}$
2. $x = x.n$
3. $x.n.n = \text{null}$
4. $x.n.r = \text{null}$
5. $ELOut_1(x)$
6. $x = x.n$
7. $z = x.n$
8. $z = x.r$

Remove $x \rightarrow n \rightarrow \star$ due to the assignment in node 1.

$f_1 \left( ELIn_2(x) \cup ELIn_4(x) \right)$
Non-Distributivity of Explicit Liveness Analysis

1. $x.n = \text{null}$
2. $x = x.n$
3. $x.n.n = \text{null}$
4. $x.n.r = \text{null}$
5. $x.n.r = \text{null}$
6. $x = x.n$
7. $z = x.n$
8. $z = x.r$

Remove $x \rightarrow n \rightarrow \ast$ due to the assignment in node 1.

$\text{ELOut}_1(x)$

May 2011 Uday Khedker
Non-Distributivity of Explicit Liveness Analysis

1. $x.n = \text{null}$

2. $x = x.n$

3. $x.n.n = \text{null}$

4. $x.n.r = \text{null}$

5. remove $x \rightarrow n \rightarrow *$ due to the assignment in node 1

6. $x = x.n$

7. $z = x.n$

8. $z = x.r$

May 2011

Uday Khedker
Non-Distributivity of Explicit Liveness Analysis

Access path $x \rightarrow r \rightarrow n \rightarrow r$ (shown in blue color) is a spurious access path that arises due to $\cup$ and is not removed by the assignment in node 1.

$ELOut_1(x)$

$f_1 \left( ELIn_2(x) \cup ELIn_4(x) \right)$

remove $x \rightarrow n \rightarrow \ast$ due to the assignment in node 1.

Access path $x \rightarrow r \rightarrow n \rightarrow r$ (shown in blue color) is a spurious access path that arises due to $\cup$ and is not removed by the assignment in node 1.
Issues Not Covered in These Slides

- Precision of information
  - Cyclic Data Structures
  - Eliminating Redundant null Assignments

- Properties of Data Flow Analysis:
  Monotonicity, Boundedness, Complexity

- Interprocedural Analysis

- Extensions for C/C++
Part 7

Conclusions
Conclusions

- Data flow analysis is a powerful program analysis technique
- Requires us to design appropriate
  - Set of values with reasonable approximations
    ⇒ Acceptable partial order and merge operation
  - Monotonic functions which are closed under composition
Conclusions

• Data flow analysis can be used for discovering complex semantics

• Unbounded information can summarized using interesting insights

  ▶ Example: Heap Analysis

  Heap manipulations consist of repeating patterns which bear a close resemblance to program structure

  Analysis of heap data is possible despite the fact that the mappings between access expressions and l-values keep changing
BTW, What is Static Analysis of Heap?

Static

Dynamic
BTW, What is Static Analysis of Heap?

Static:
- Abstract, Bounded, Single Instance

Dynamic:
- Concrete, Unbounded, Infinitely Many

Static:
- Program Code

Dynamic:
- Program Execution

May 2011
BTW, What is Static Analysis of Heap?

- **Static**
  - Program Code

- **Dynamic**
  - Program Execution
  - Heap Memory
  - Heap Memory
  - Heap Memory

**Framework:**
- Abstract, Bounded, Single Instance
- Concrete, Unbounded, Infinitely Many

May 2011
Uday Khedker
BTW, What is Static Analysis of Heap?

- Abstract, Bounded, Single Instance
- Concrete, Unbounded, Infinitely Many

Static

- Program Code
- Summary Heap Data

Dynamic

- Program Execution
- Heap Memory
- Heap Memory
- ?
BTW, What is Static Analysis of Heap?

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

Static

Program Code

Summary Heap Data

Dynamic

Profiling

Program Execution

Heap Memory

Heap Memory

Heap Memory

Heap Memory

Heap Memory

Heap Memory

Heap Memory
BTW, What is Static Analysis of Heap?

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

Static

Program Code

Static Analysis

Summary Heap Data

Dynamic

Program Execution

Heap Memory

?