General Data Flow Framekworks

Uday P. Khedker

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

May 2011

Part 1

About These Slides

Copyright

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

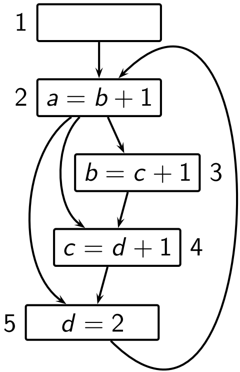
Part 2

Modelling General Flows

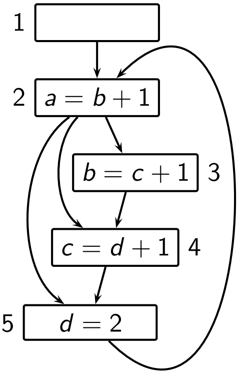
Part 3

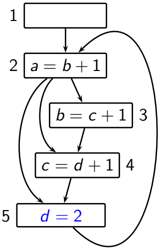
Precise Modelling of General Flows

Complexity of Constant Propagation?



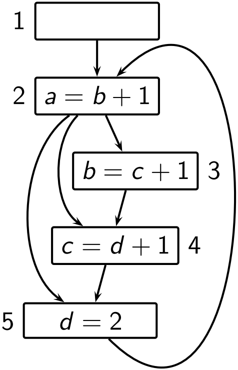
Complexity of Constant Propagation?

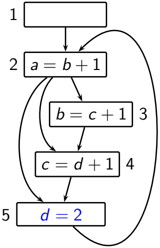




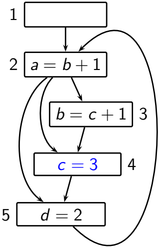
Iteration #1

Complexity of Constant Propagation?



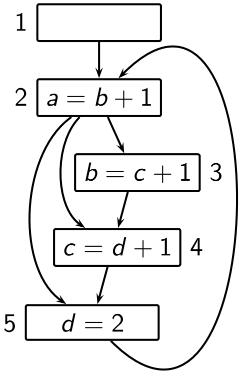


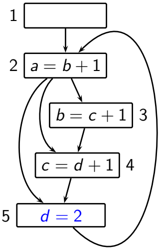
Iteration #1



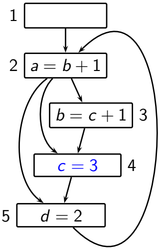
Iteration #2

Complexity of Constant Propagation?

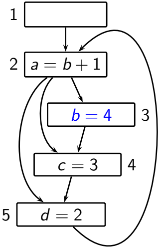




Iteration #1

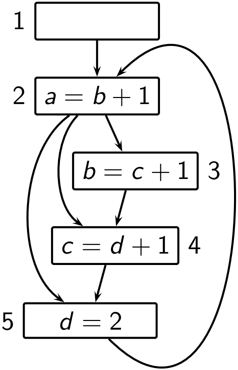


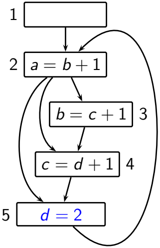
Iteration #2



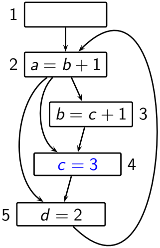
Iteration #3

Complexity of Constant Propagation?

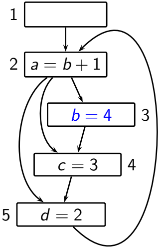




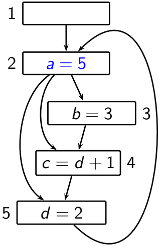
Iteration #1



Iteration #2

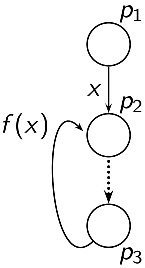


Iteration #3



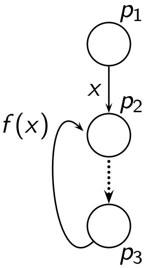
Iteration #4

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	$f(x)$
$p_1, p_2, p_3, p_2, p_3, p_2$	$f(f(x)) = f^2(x)$
$p_1, p_2, p_3, p_2, p_3, p_2, p_3, p_2$	$f(f(f(x))) = f^3(x)$
...	...

Loop Closures of Flow Functions

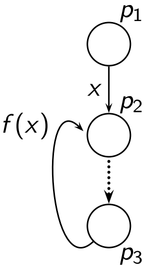


Paths Terminating at p_2	Data Flow Value
p_1, p_2	x
p_1, p_2, p_3, p_2	$f(x)$
$p_1, p_2, p_3, p_2, p_3, p_2$	$f(f(x)) = f^2(x)$
$p_1, p_2, p_3, p_2, p_3, p_2, p_3, p_2$	$f(f(f(x))) = f^3(x)$
$\ldots$	$\ldots$

- 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

$$
\begin{aligned}
f^*(x) &= x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap \ldots \\
f^2(x) &= f\,(\text{Gen} \cup (x - \text{Kill})) \\
&= \text{Gen} \cup ((\text{Gen} \cup (x - \text{Kill})) - \text{Kill}) \\
&= \text{Gen} \cup ((\text{Gen} - \text{Kill}) \cup (x - \text{Kill})) \\
&= \text{Gen} \cup (\text{Gen} - \text{Kill}) \cup (x - \text{Kill}) \\
&= \text{Gen} \cup (x - \text{Kill}) \quad = \quad f(x) \\
f^*(x) &= x \sqcap f(x)
\end{aligned}
$$

Loop Closures in Bit Vector Frameworks

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

$$
\begin{array}{rcl}
f^*(x) & = & x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap \ldots \\
f^2(x) & = & f\,(\text{Gen} \cup (x - \text{Kill})) \\
 & = & \text{Gen} \cup ((\text{Gen} \cup (x - \text{Kill})) - \text{Kill}) \\
 & = & \text{Gen} \cup ((\text{Gen} - \text{Kill}) \cup (x - \text{Kill})) \\
 & = & \text{Gen} \cup (\text{Gen} - \text{Kill}) \cup (x - \text{Kill}) \\
 & = & \text{Gen} \cup (x - \text{Kill}) \quad = \quad f(x) \\
f^*(x) & = & x \sqcap f(x)
\end{array}
$$

- *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

$$\begin{aligned}
f^*(x) &= x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap \ldots \\
f^2(x) &= f\,(\text{Gen} \cup (x - \text{Kill})) \\
&= \text{Gen} \cup ((\text{Gen} \cup (x - \text{Kill})) - \text{Kill}) \\
&= \text{Gen} \cup ((\text{Gen} - \text{Kill}) \cup (x - \text{Kill})) \\
&= \text{Gen} \cup (\text{Gen} - \text{Kill}) \cup (x - \text{Kill}) \\
&= \text{Gen} \cup (x - \text{Kill}) \;\; = \;\; f(x) \\
f^*(x) &= x \sqcap f(x)
\end{aligned}$$

- *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 (eg. 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

$f : L \mapsto L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

Separability

$f : L \mapsto L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

Separable	**Non-Separable**
Example: All bit vector frameworks	Example: Constant Propagation

Separability

$f : L \mapsto L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

Separable

$$\langle\, \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m \,\rangle$$
$$\Downarrow$$
$$f$$
$$\Downarrow$$
$$\langle\, \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m \,\rangle$$

Example: All bit vector frameworks

Non-Separable

$$\langle\, \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m \,\rangle$$
$$\Downarrow$$
$$f$$
$$\Downarrow$$
$$\langle\, \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m \,\rangle$$

Example: Constant Propagation

Separability

$f : L \mapsto L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

Separable	Non-Separable
$\langle\ \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m\ \rangle$	$\langle\ \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m\ \rangle$
$\widehat{h}_2$	$\Downarrow$ f $\Downarrow$
$\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$	$\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$
Example: All bit vector frameworks	Example: Constant Propagation

Separability

$f : L \mapsto L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

Separable

$\langle\ \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m\ \rangle$

$\widehat{h}_2$

$\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$

$\widehat{h} : \widehat{L} \mapsto \widehat{L}$

Example: All bit vector frameworks

Non-Separable

$\langle\ \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m\ \rangle$

$\Downarrow$

f

$\Downarrow$

$\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$

Example: Constant Propagation

Separability

$f : L \mapsto L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

Separable

$\langle\ \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m\ \rangle$

$\widehat{h}_2$

$\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$

$\widehat{h} : \widehat{L} \mapsto \widehat{L}$

Example: All bit vector frameworks

Non-Separable

$\langle\ \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m\ \rangle$

$\widehat{h}_2$

$\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$

Example: Constant Propagation

Separability

$f : L \mapsto L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

Separable

$\langle\ \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m\ \rangle$

$\widehat{h}_2$

$\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$

$\widehat{h} : \widehat{L} \mapsto \widehat{L}$

Example: All bit vector frameworks

Non-Separable

$\langle\ \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m\ \rangle$

$\widehat{h}_2$

$\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$

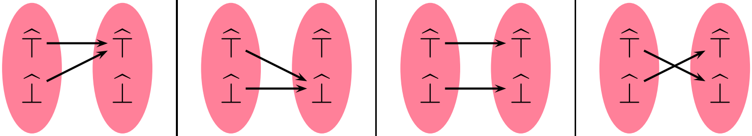
$\widehat{h} : L \mapsto \widehat{L}$

Example: Constant Propagation

Separability of Bit Vector Frameworks

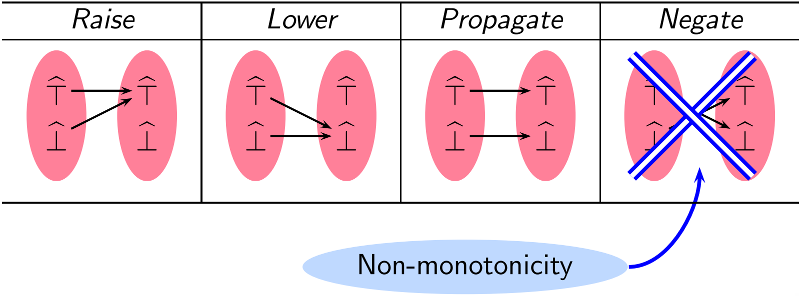
- $\widehat{L}$ is $\{0,1\}$, L is $\{0,1\}^m$
- $\widehat{\sqcap}$ is either boolean AND or boolean OR
- $\widehat{\top}$ and $\widehat{\bot}$ are 0 or 1 depending on $\widehat{\sqcap}$.
- $\widehat{h}$ is a *bit function* and could be one of the following:

Raise	*Lower*	*Propagate*	*Negate*

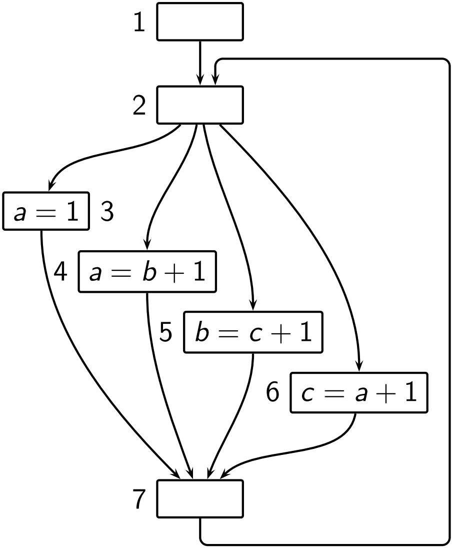


Separability of Bit Vector Frameworks

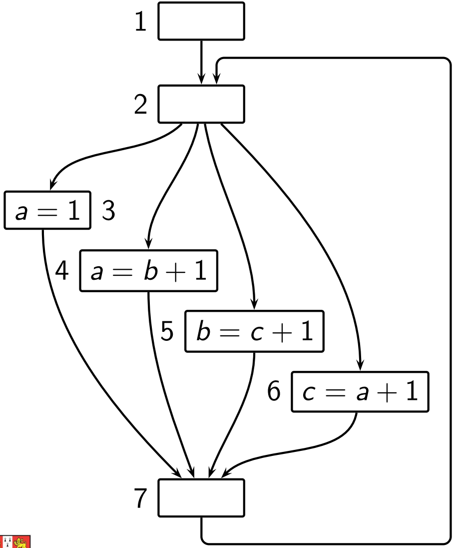
- $\widehat{L}$ is $\{0,1\}$, L is $\{0,1\}^m$
- $\widehat{\sqcap}$ is either boolean AND or boolean OR
- $\widehat{\top}$ and $\widehat{\bot}$ are 0 or 1 depending on $\widehat{\sqcap}$.
- $\widehat{h}$ is a *bit function* and could be one of the following:



Boundedness of Constant Propagation



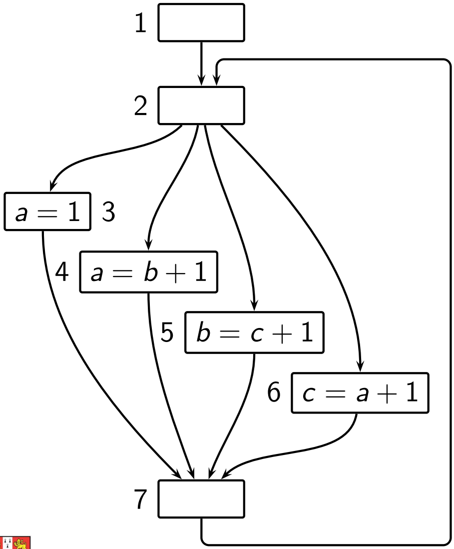
Boundedness of Constant Propagation



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

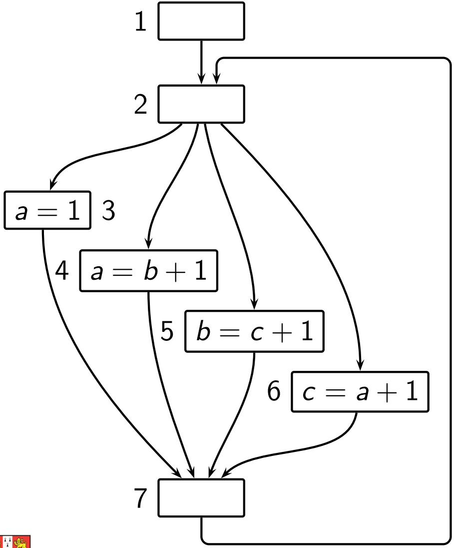


Summary flow function:
(data flow value at node 7)

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

$$f^0(\top) = \langle \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$$
$$f^1(\top) = \langle 1, \widehat{\top}, \widehat{\top} \rangle$$

Boundedness of Constant Propagation



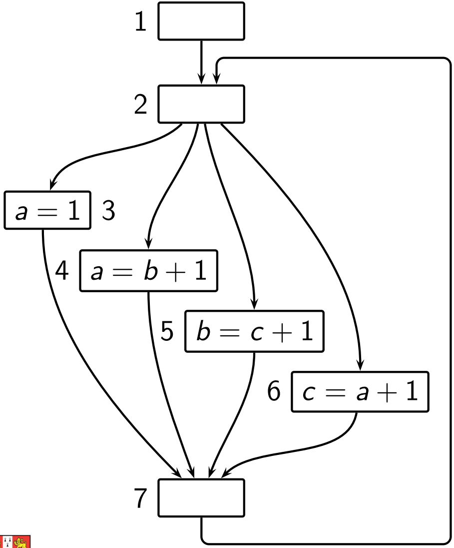
Summary flow function:

(data flow value at node 7)

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

$$\begin{aligned} f^0(\top) &= \langle \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle \\ f^1(\top) &= \langle 1, \widehat{\top}, \widehat{\top} \rangle \\ f^2(\top) &= \langle 1, \widehat{\top}, 2 \rangle \end{aligned}$$

Boundedness of Constant Propagation

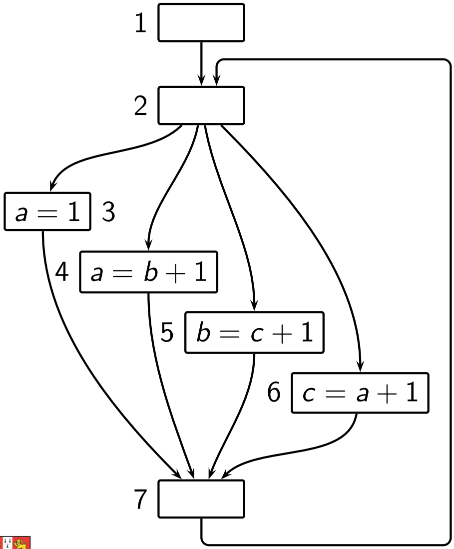


Summary flow function:
(data flow value at node 7)

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

$$\begin{aligned} f^0(\top) &= \langle \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle \\ f^1(\top) &= \langle 1, \widehat{\top}, \widehat{\top} \rangle \\ f^2(\top) &= \langle 1, \widehat{\top}, 2 \rangle \\ f^3(\top) &= \langle 1, 3, 2 \rangle \end{aligned}$$

Boundedness of Constant Propagation

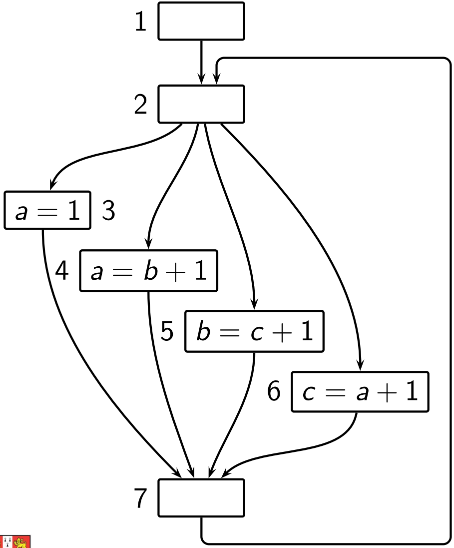


Summary flow function:
(data flow value at node 7)

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

$$\begin{aligned}
f^0(\top) &= \langle \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle \\
f^1(\top) &= \langle 1, \widehat{\top}, \widehat{\top} \rangle \\
f^2(\top) &= \langle 1, \widehat{\top}, 2 \rangle \\
f^3(\top) &= \langle 1, 3, 2 \rangle \\
f^4(\top) &= \langle \widehat{\bot}, 3, 2 \rangle
\end{aligned}$$

Boundedness of Constant Propagation

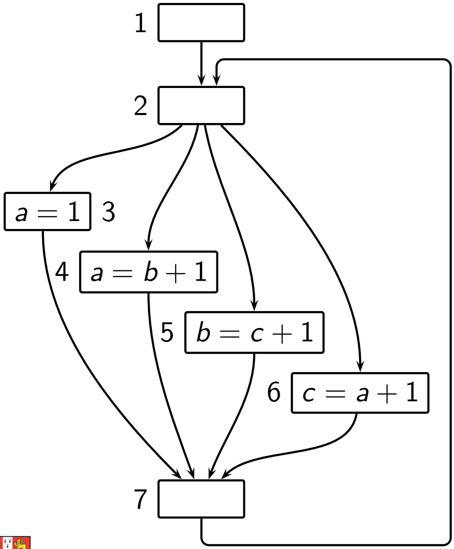


Summary flow function:
(data flow value at node 7)

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

$$\begin{aligned}
f^0(\top) &= \langle \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle \\
f^1(\top) &= \langle 1, \widehat{\top}, \widehat{\top} \rangle \\
f^2(\top) &= \langle 1, \widehat{\top}, 2 \rangle \\
f^3(\top) &= \langle 1, 3, 2 \rangle \\
f^4(\top) &= \langle \widehat{\bot}, 3, 2 \rangle \\
f^5(\top) &= \langle \widehat{\bot}, 3, \widehat{\bot} \rangle
\end{aligned}$$

Boundedness of Constant Propagation



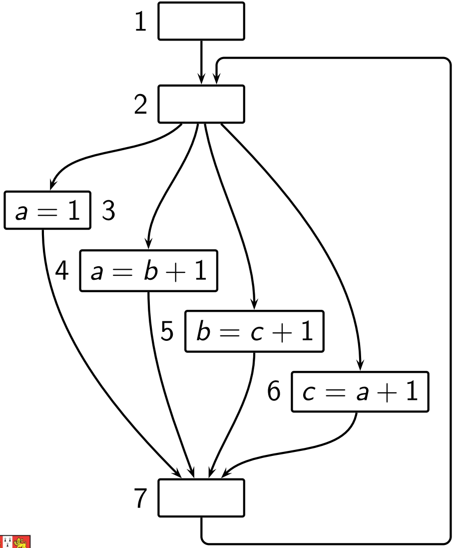
Summary flow function:

(data flow value at node 7)

$$
\begin{aligned}
f(\langle v_a, v_b, v_c \rangle) \;=\;& \langle\; 1 \sqcap (v_b + 1), \\
& \quad (v_c + 1), \\
& \quad (v_a + 1) \\
& \rangle
\end{aligned}
$$

$$
\begin{aligned}
f^0(\top) &= \langle \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle \\
f^1(\top) &= \langle 1, \widehat{\top}, \widehat{\top} \rangle \\
f^2(\top) &= \langle 1, \widehat{\top}, 2 \rangle \\
f^3(\top) &= \langle 1, 3, 2 \rangle \\
f^4(\top) &= \langle \widehat{\bot}, 3, 2 \rangle \\
f^5(\top) &= \langle \widehat{\bot}, 3, \widehat{\bot} \rangle \\
f^6(\top) &= \langle \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle
\end{aligned}
$$

Boundedness of Constant Propagation

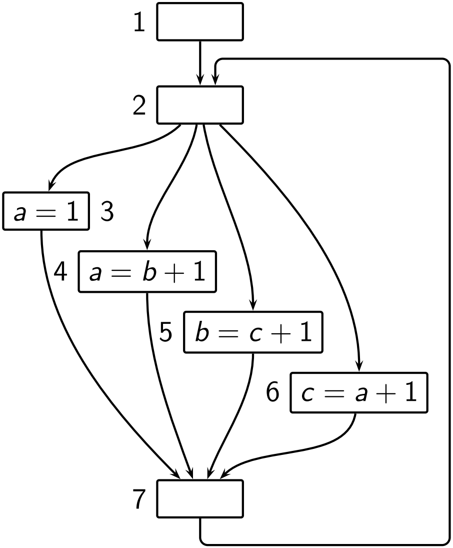


Summary flow function:
(data flow value at node 7)

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

$$\begin{aligned}
f^0(\top) &= \langle \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle \\
f^1(\top) &= \langle 1, \widehat{\top}, \widehat{\top} \rangle \\
f^2(\top) &= \langle 1, \widehat{\top}, 2 \rangle \\
f^3(\top) &= \langle 1, 3, 2 \rangle \\
f^4(\top) &= \langle \widehat{\bot}, 3, 2 \rangle \\
f^5(\top) &= \langle \widehat{\bot}, 3, \widehat{\bot} \rangle \\
f^6(\top) &= \langle \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle \\
f^7(\top) &= \langle \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle
\end{aligned}$$

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

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

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: $2 \times |\mathbb{V}\text{ar}|$

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: $2 \times |\mathbb{V}\text{ar}|$
- Boundedness parameter k is $(2 \times |\mathbb{V}\text{ar}|) + 1$

Modelling Flow Functions for General Flows

- General flow functions can be written as

$$f_n(X) \quad = \quad (X - \mathsf{Kill}_n(X)) \cup \mathsf{Gen}_n(X)$$

where Gen and Kill have constant and dependent parts

$$\begin{aligned} \mathsf{Gen}_n(X) &= ConstGen_n \cup DepGen_n(X) \\ \mathsf{Kill}_n(X) &= ConstKill_n \cup DepKill_n(X) \end{aligned}$$

Modelling Flow Functions for General Flows

- General flow functions can be written as

$$f_n(X) \quad = \quad (X - \text{Kill}_n(X)) \cup \text{Gen}_n(X)$$

where Gen and Kill have constant and dependent parts

$$\begin{aligned} \text{Gen}_n(X) &= \mathit{ConstGen}_n \cup \mathit{DepGen}_n(X) \\ \text{Kill}_n(X) &= \mathit{ConstKill}_n \cup \mathit{DepKill}_n(X) \end{aligned}$$

- 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

$$f_n(X) \quad = \quad (X - \mathsf{Kill}_n(X)) \cup \mathsf{Gen}_n(X)$$

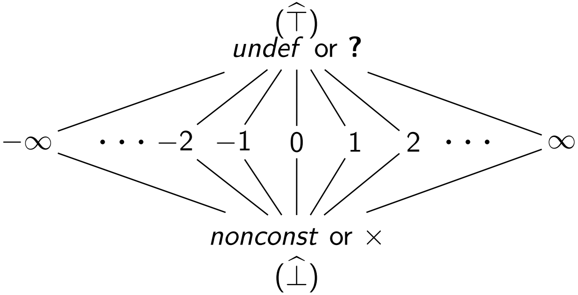
where Gen and Kill have constant and dependent parts

$$\begin{aligned} \mathsf{Gen}_n(X) &= ConstGen_n \cup DepGen_n(X) \\ \mathsf{Kill}_n(X) &= ConstKill_n \cup DepKill_n(X) \end{aligned}$$

- 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

$$DepGen_n(X) = DepKill_n(X) = \emptyset$$

Component Lattice for Integer Constant Propagation



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

$\widehat{\sqcap}$	$\langle v, \mathbf{?} \rangle$	$\langle v, \times \rangle$	$\langle v, c_1 \rangle$
$\langle v, \mathbf{?} \rangle$	$\langle v, \mathbf{?} \rangle$	$\langle v, \times \rangle$	$\langle v, c_1 \rangle$
$\langle v, \times \rangle$	$\langle v, \times \rangle$	$\langle v, \times \rangle$	$\langle v, \times \rangle$
$\langle v, c_2 \rangle$	$\langle v, c_2 \rangle$	$\langle v, \times \rangle$	If $c_1 = c_2$ then $\langle v, c_1 \rangle$ else $\langle v, \times \rangle$

Flow Functions for Constant Propagation

- Flow function for $r = a_1 * a_2$

$mult$	$\langle a_1, \mathbf{?} \rangle$	$\langle a_1, \times \rangle$	$\langle a_1, c_1 \rangle$
$\langle a_2, \mathbf{?} \rangle$	$\langle r, \mathbf{?} \rangle$	$\langle r, \times \rangle$	$\langle r, \mathbf{?} \rangle$
$\langle a_2, \times \rangle$	$\langle r, \times \rangle$	$\langle r, \times \rangle$	$\langle r, \times \rangle$
$\langle a_2, c_2 \rangle$	$\langle r, \mathbf{?} \rangle$	$\langle r, \times \rangle$	$\langle r, (c_1 * c_2) \rangle$

Defining Data Flow Equations for Constant Propagation

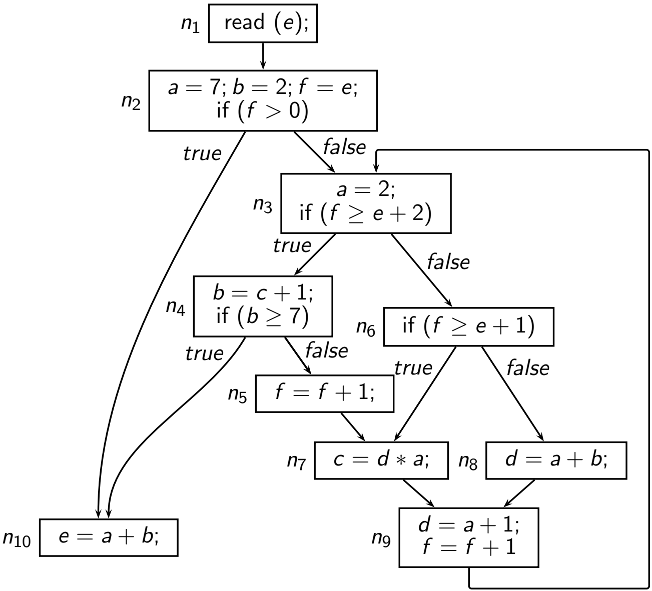
	$ConstGen_n$	$DepGen_n(X)$	$ConstKill_n$	$DepKill_n(X)$
$v = c$, $c \in \mathbb{C}\text{onst}$	$\{\langle v, c\rangle\}$	$\emptyset$	$\emptyset$	$\{\langle v, d\rangle \mid \langle v, d\rangle \in X\}$
$v = e$, $e \in \mathbb{E}\text{xpr}$	$\emptyset$	$\{\langle v, eval(e,X)\rangle\}$	$\emptyset$	$\{\langle v, d\rangle \mid \langle v, d\rangle \in X\}$
$read(v)$	$\{\langle v, \times\rangle\}$	$\emptyset$	$\emptyset$	$\{\langle v, d\rangle \mid \langle v, d\rangle \in X\}$
$other$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Defining Data Flow Equations for Constant Propagation

	$ConstGen_n$	$DepGen_n(X)$	$ConstKill_n$	$DepKill_n(X)$
$v = c$, $c \in \mathbb{C}\text{onst}$	$\{\langle v, c\rangle\}$	$\emptyset$	$\emptyset$	$\{\langle v, d\rangle \mid \langle v, d\rangle \in X\}$
$v = e$, $e \in \mathbb{E}\text{xpr}$	$\emptyset$	$\{\langle v, eval(e,X)\rangle\}$	$\emptyset$	$\{\langle v, d\rangle \mid \langle v, d\rangle \in X\}$
$read(v)$	$\{\langle v, \times\rangle\}$	$\emptyset$	$\emptyset$	$\{\langle v, d\rangle \mid \langle v, d\rangle \in X\}$
$other$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

$eval(a_1\ op\ a_2, X)$			
	$\langle a_1, \textbf{?}\rangle \in X$	$\langle a_1, \times\rangle \in X$	$\langle a_1, c_1\rangle \in X$
$\langle a_2, \textbf{?}\rangle \in X$	**?**	$\times$	**?**
$\langle a_2, \times\rangle \in X$	$\times$	$\times$	$\times$
$\langle a_2, c_2\rangle \in X$	**?**	$\times$	$c_1\ op\ c_2$

Example Program for Constant Propagation



Result of Constant Propagation

	Iteration #1	Changes in iteration #2	Changes in iteration #3	Changes in iteration #4
In_{n_1}	$\hat{\top}, \hat{\top}, \hat{\top}, \hat{\top}, \hat{\top}, \hat{\top}$			
Out_{n_1}	$\hat{\top}, \hat{\top}, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\top}$			
In_{n_2}	$\hat{\top}, \hat{\top}, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\top}$			
Out_{n_2}	$7, 2, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\bot}$			
In_{n_3}	$7, 2, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\bot}$	$\hat{\bot}, 2, \hat{\top}, 3, \hat{\bot}, \hat{\bot}$	$\hat{\bot}, 2, 6, 3, \hat{\bot}, \hat{\bot}$	$\hat{\bot}, \hat{\bot}, 6, 3, \hat{\bot}, \hat{\bot}$
Out_{n_3}	$2, 2, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\bot}$	$2, 2, \hat{\top}, 3, \hat{\bot}, \hat{\bot}$	$2, 2, 6, 3, \hat{\bot}, \hat{\bot}$	$2, \hat{\bot}, 6, 3, \hat{\bot}, \hat{\bot}$
In_{n_4}	$2, 2, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\bot}$	$2, 2, \hat{\top}, 3, \hat{\bot}, \hat{\bot}$	$2, 2, 6, 3, \hat{\bot}, \hat{\bot}$	$2, \hat{\bot}, 6, 3, \hat{\bot}, \hat{\bot}$
Out_{n_4}	$2, \hat{\top}, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\bot}$	$2, \hat{\top}, \hat{\top}, 3, \hat{\bot}, \hat{\bot}$	$2, 7, 6, 3, \hat{\bot}, \hat{\bot}$	
In_{n_5}	$2, \hat{\top}, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\bot}$	$2, \hat{\top}, \hat{\top}, 3, \hat{\bot}, \hat{\bot}$	$2, 7, 6, 3, \hat{\bot}, \hat{\bot}$	
Out_{n_5}	$2, \hat{\top}, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\bot}$	$2, \hat{\top}, \hat{\top}, 3, \hat{\bot}, \hat{\bot}$	$2, 7, 6, 3, \hat{\bot}, \hat{\bot}$	
In_{n_6}	$2, 2, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\bot}$	$2, 2, \hat{\top}, 3, \hat{\bot}, \hat{\bot}$	$2, 2, 6, 3, \hat{\bot}, \hat{\bot}$	$2, \hat{\bot}, 6, 3, \hat{\bot}, \hat{\bot}$
Out_{n_6}	$2, 2, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\bot}$	$2, 2, \hat{\top}, 3, \hat{\bot}, \hat{\bot}$	$2, 2, 6, 3, \hat{\bot}, \hat{\bot}$	$2, \hat{\bot}, 6, 3, \hat{\bot}, \hat{\bot}$
In_{n_7}	$2, 2, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\bot}$	$2, 2, \hat{\top}, 3, \hat{\bot}, \hat{\bot}$	$2, \hat{\bot}, 6, 3, \hat{\bot}, \hat{\bot}$	
Out_{n_7}	$2, 2, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\bot}$	$2, 2, 6, 3, \hat{\bot}, \hat{\bot}$	$2, \hat{\bot}, 6, 3, \hat{\bot}, \hat{\bot}$	
In_{n_8}	$2, 2, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\bot}$	$2, 2, \hat{\top}, 3, \hat{\bot}, \hat{\bot}$	$2, 2, 6, 3, \hat{\bot}, \hat{\bot}$	$2, \hat{\bot}, 6, 3, \hat{\bot}, \hat{\bot}$
Out_{n_8}	$2, 2, \hat{\top}, 4, \hat{\bot}, \hat{\bot}$	$2, 2, \hat{\top}, 4, \hat{\bot}, \hat{\bot}$	$2, 2, 6, 4, \hat{\bot}, \hat{\bot}$	$2, \hat{\bot}, 6, \hat{\bot}, \hat{\bot}, \hat{\bot}$
In_{n_9}	$2, 2, \hat{\top}, 4, \hat{\bot}, \hat{\bot}$	$2, 2, 6, \hat{\bot}, \hat{\bot}, \hat{\bot}$	$2, \hat{\bot}, 6, \hat{\bot}, \hat{\bot}, \hat{\bot}$	
Out_{n_9}	$2, 2, \hat{\top}, 3, \hat{\bot}, \hat{\bot}$	$2, 2, 6, 3, \hat{\bot}, \hat{\bot}$	$2, \hat{\bot}, 6, 3, \hat{\bot}, \hat{\bot}$	
$In_{n_{10}}$	$\hat{\bot}, 2, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\bot}$	$\hat{\bot}, 2, \hat{\top}, 3, \hat{\bot}, \hat{\bot}$	$\hat{\bot}, \hat{\bot}, 6, 3, \hat{\bot}, \hat{\bot}$	
$Out_{n_{10}}$	$\hat{\bot}, 2, \hat{\top}, \hat{\top}, \hat{\bot}, \hat{\bot}$	$\hat{\bot}, 2, \hat{\top}, 3, \hat{\bot}, \hat{\bot}$	$\hat{\bot}, \hat{\bot}, 6, 3, \hat{\bot}, \hat{\bot}$	

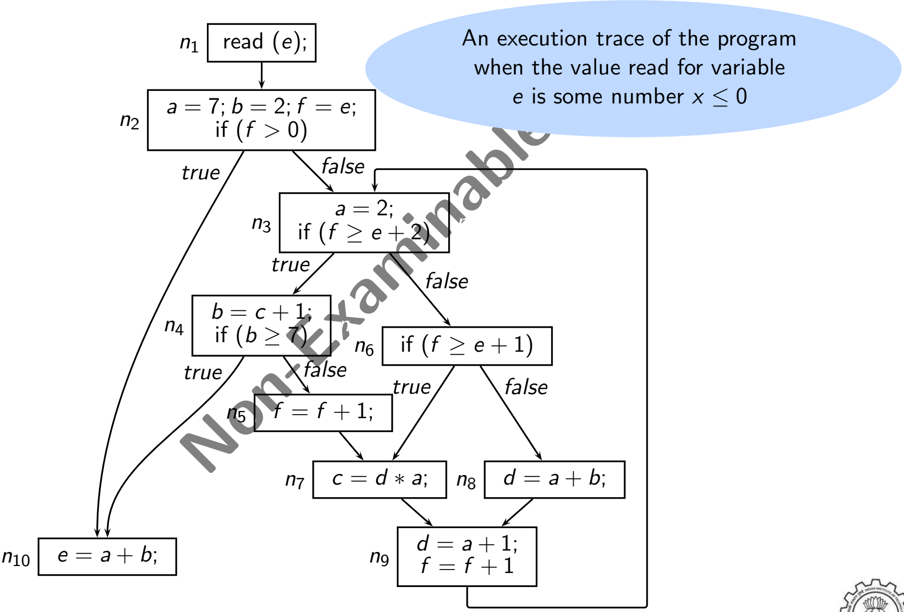
Monotonicity of Constant Propagation

- Flow function $f_n(X) = (X - \text{Kill}_n(X)) \cup \text{Gen}_n(X)$ where

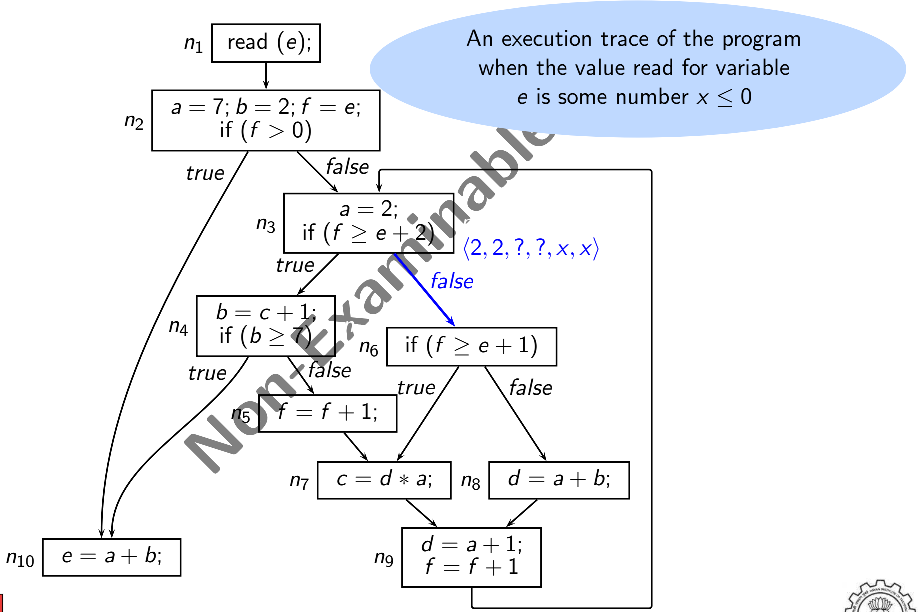
$$\begin{aligned} \text{Gen}_n(X) &= \mathit{ConstGen}_n \cup \mathit{DepGen}_n(X) \\ \text{Kill}_n(X) &= \mathit{ConstKill}_n \cup \mathit{DepKill}_n(X) \end{aligned}$$

- $\mathit{ConstGen}_n$ and $\mathit{ConstKill}_n$ are trivially monotonic
- To show $X_1 \sqsubseteq X_2 \Rightarrow \mathit{DepGen}_n(X_1) \sqsubseteq \mathit{DepGen}_n(X_2)$ we need to show that $X_1 \sqsubseteq X_2 \Rightarrow \mathit{eval}(e, X_1) \sqsubseteq \mathit{eval}(e, X_2)$. This follows from definition of $\mathit{eval}(e, X)$.
- To show $X_1 \sqsubseteq X_2 \Rightarrow (X_1 - \mathit{DepKill}_n(X_1)) \sqsubseteq (X_2 - \mathit{DepKill}_n(X_2))$ observe that $\mathit{DepKill}_n$ removes the pair corresponding to the variable modified in statement n. Data flow values of other variables remain unaffected.

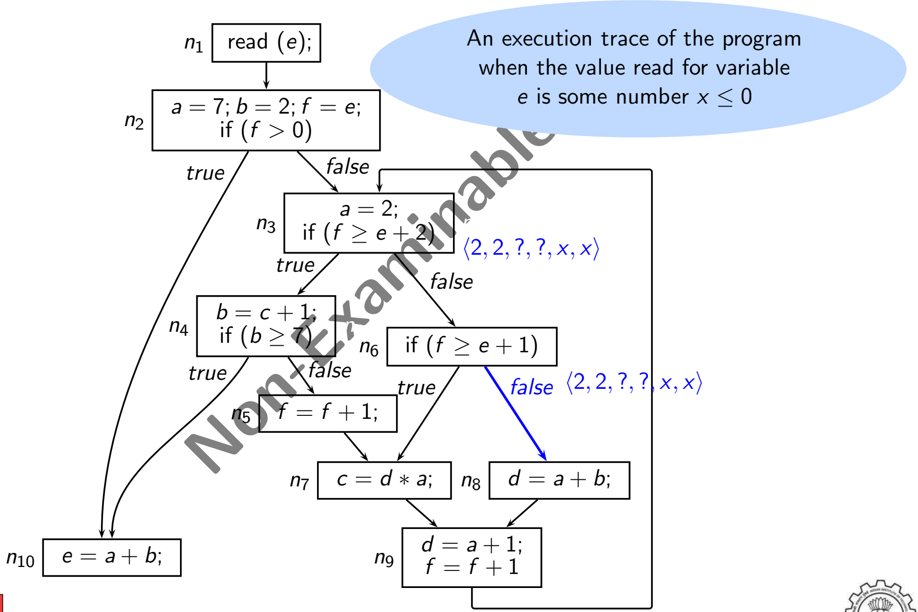
Conditional Constant Propagation



Conditional Constant Propagation

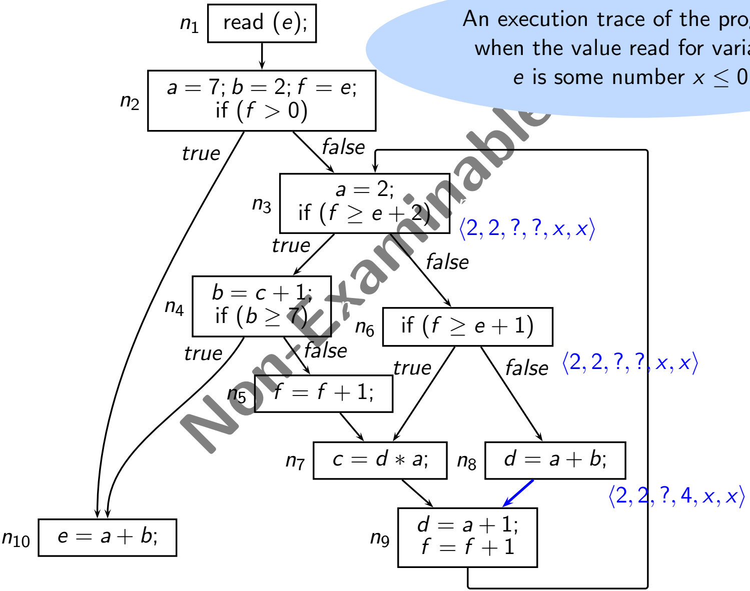


Conditional Constant Propagation



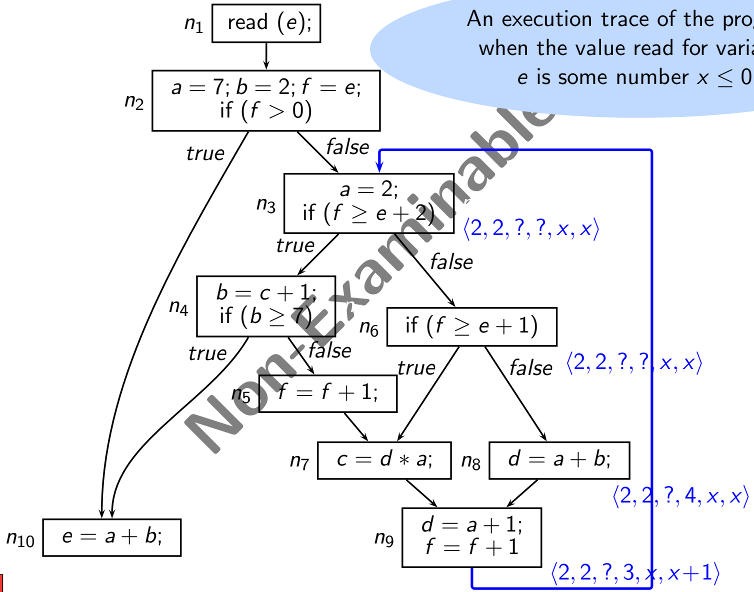
Conditional Constant Propagation

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

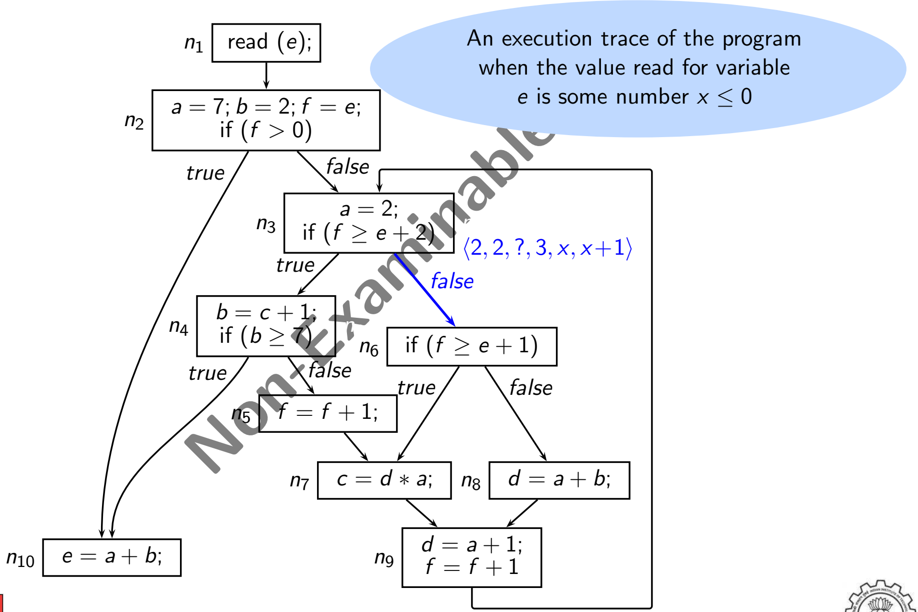


Conditional Constant Propagation

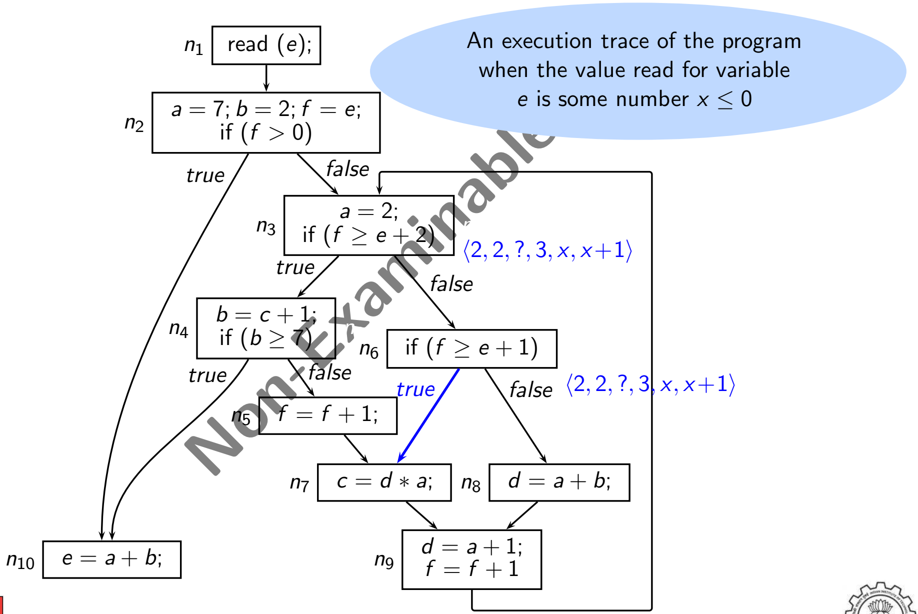
An execution trace of the program when the value read for variable e is some number $x \leq 0$



Conditional Constant Propagation

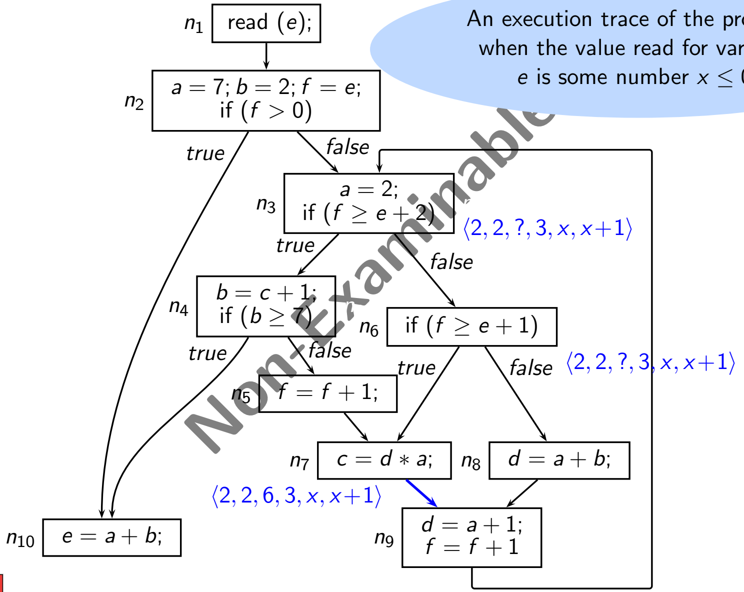


Conditional Constant Propagation

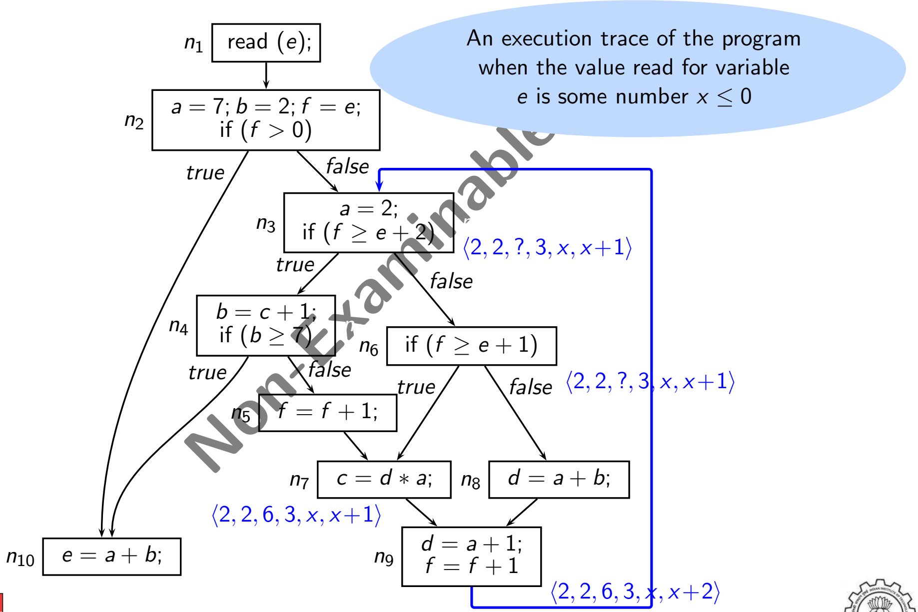


Conditional Constant Propagation

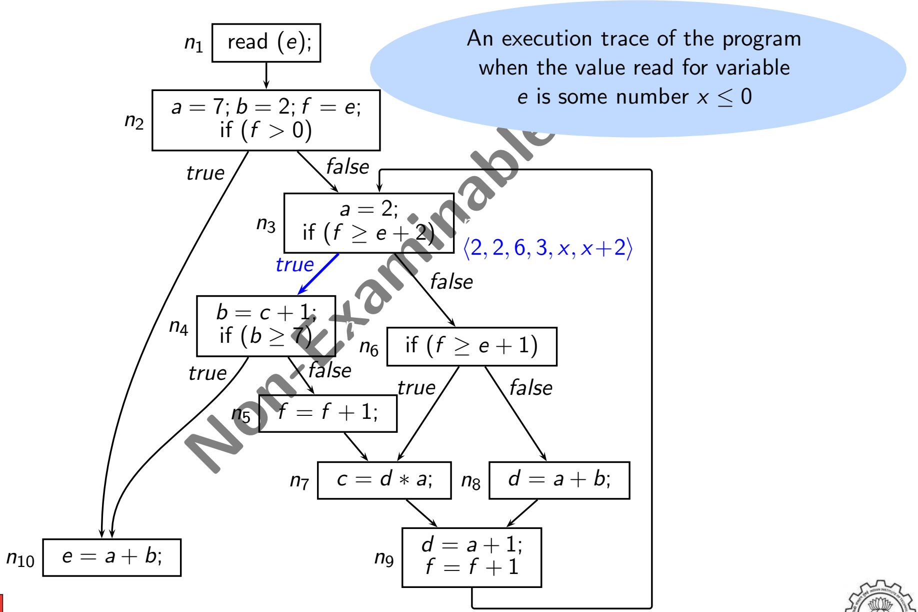
An execution trace of the program when the value read for variable e is some number $x \leq 0$



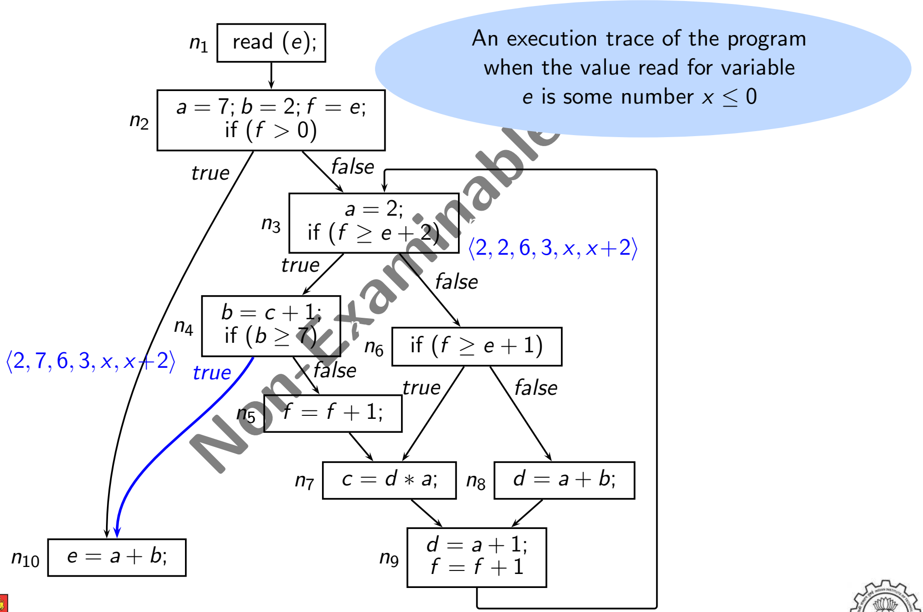
Conditional Constant Propagation



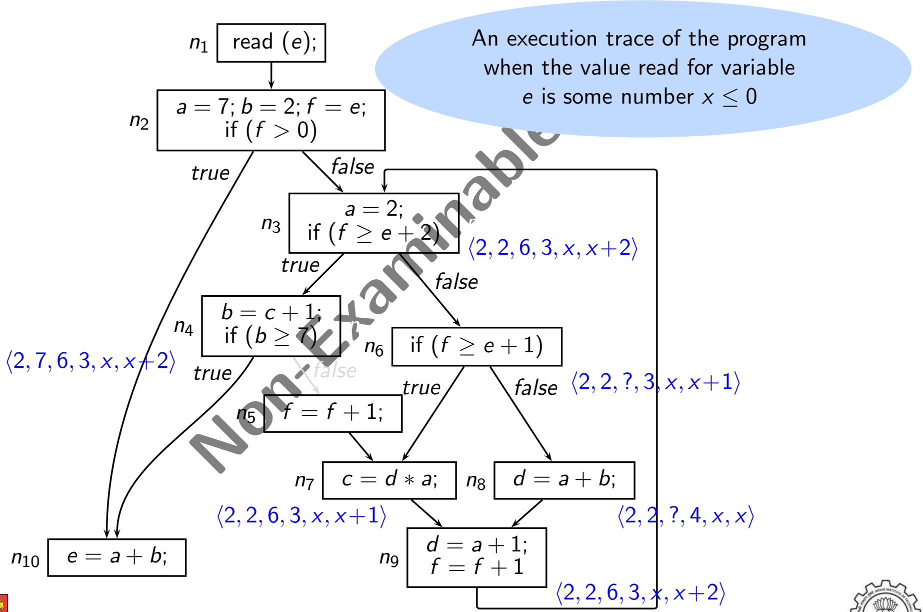
Conditional Constant Propagation



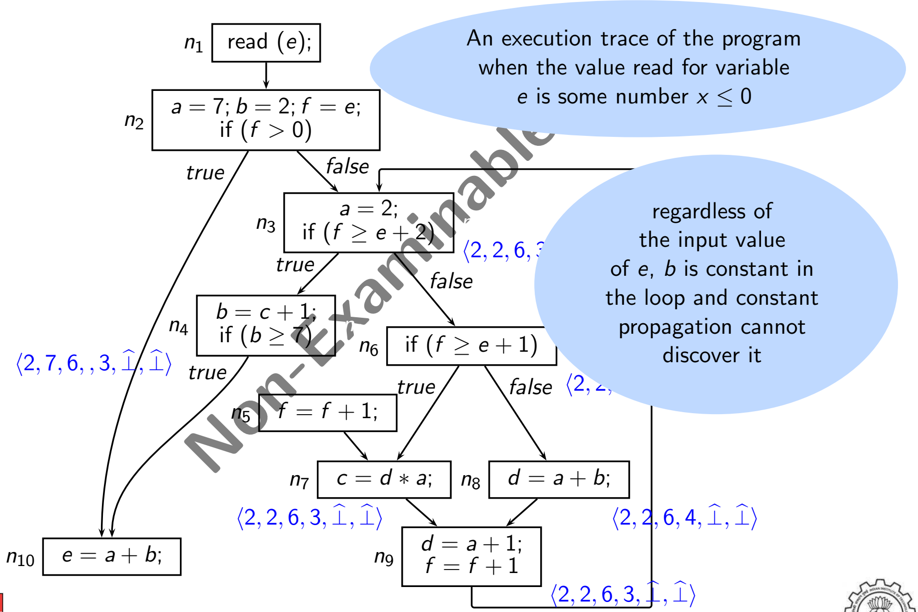
Conditional Constant Propagation



Conditional Constant Propagation



Conditional Constant Propagation



Lattice for Conditional Constant Propagation

$$\begin{array}{c} \textit{notReachable} \\ | \\ \textit{reachable} \end{array} \quad \times \quad \widehat{L} \quad \times \quad \widehat{L} \quad \times \quad \cdots \quad \times \quad \widehat{L}$$

- Let $\langle s, X \rangle$ denote an augmented data flow value where $s \in \{\textit{reachable}, \textit{notReachable}\}$ and $X \in L$.
- If we can maintain the invariant $s = \textit{notReachable} \Rightarrow X = \top$, then the meet can be defined as

$$\langle s_1, X_1 \rangle \ \sqcap_c \ \langle s_2, X_2 \rangle = \langle s_1 \sqcap_c s_2, X_1 \sqcap X_2 \rangle$$

Data Flow Equations for Conditional Constant Propagation

$$In_n = \begin{cases} \langle reachable, BI\rangle & n \text{ is } Start \\ \displaystyle\prod_{p \in pred(n)}{}_c \; g_{p\rightarrow n}(Out_p) & \text{otherwise} \end{cases}$$

$$Out_n = \begin{cases} \langle reachable, f_n(X)\rangle & In_n = \langle reachable, X\rangle \\ \langle notReachable, \top\rangle & \text{otherwise} \end{cases}$$

$$g_{m\rightarrow n}(s, X) = \begin{cases} \langle notReachable, \top\rangle & evalCond(m, X) \neq undefined \text{ and} \\ & evalCond(m, X) \neq label(m \rightarrow n) \\ \langle s, X\rangle & \text{otherwise} \end{cases}$$

Conditional Constant Propagation

	Iteration #1	Changes in iteration #2	Changes in iteration #3
In_{n_1}	$R, \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$		
Out_{n_1}	$R, \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\top} \rangle$		
In_{n_2}	$R, \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$		
Out_{n_2}	$R, \langle 7, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$		
In_{n_3}	$R, \langle 7, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
Out_{n_3}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
In_{n_4}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
Out_{n_4}	$R, \langle 2, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, \widehat{\top}, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 7, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
In_{n_5}	$R, \langle 2, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, \widehat{\top}, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$N, \top = \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$
Out_{n_5}	$R, \langle 2, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, \widehat{\top}, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$N, \top = \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$
In_{n_6}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
Out_{n_6}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
In_{n_7}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
Out_{n_7}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
In_{n_8}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
Out_{n_8}	$R, \langle 2, 2, \widehat{\top}, 4, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, \widehat{\top}, 4, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 4, \widehat{\bot}, \widehat{\bot} \rangle$
In_{n_9}	$R, \langle 2, 2, \widehat{\top}, 4, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle$
Out_{n_9}	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
$In_{n_{10}}$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, \widehat{\bot}, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
$Out_{n_{10}}$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, \widehat{\bot}, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$

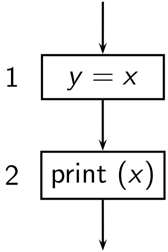
Non-Examinable

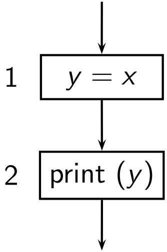
Part 4

Faint Variables Analysis

Faint Variables Analysis

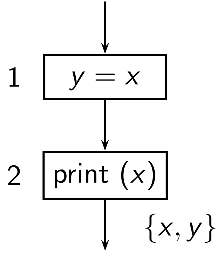
A variable is faint if it is dead or is used in computing faint variables.

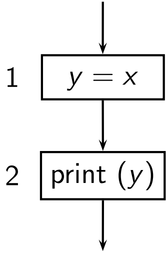




Faint Variables Analysis

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

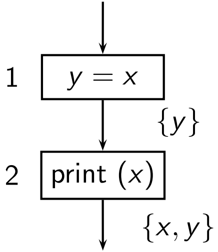


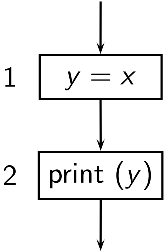


$\text{Gen}_2 = \emptyset$
$\text{Kill}_2 = \{x\}$

Faint Variables Analysis

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



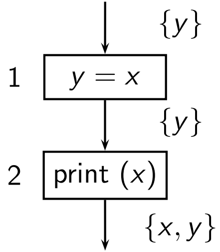


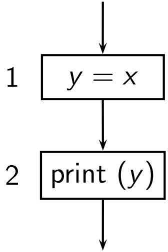
$\text{Gen}_2 = \emptyset$ $\quad$ $\text{Gen}_1 = \{y\}$

$\text{Kill}_2 = \{x\}$ $\quad$ $\text{Kill}_1 = \emptyset$

Faint Variables Analysis

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



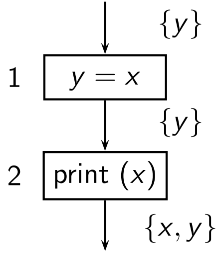


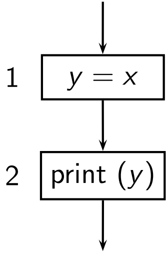
$\text{Gen}_2 = \emptyset$ $\quad$ $\text{Gen}_1 = \{y\}$

$\text{Kill}_2 = \{x\}$ $\quad$ $\text{Kill}_1 = \emptyset$

Faint Variables Analysis

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



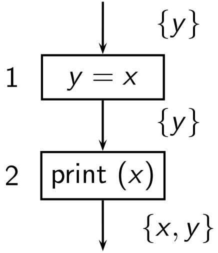


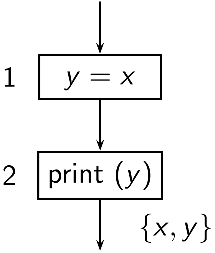
$\text{Gen}_2 = \emptyset$ $\text{Gen}_1 = \{y\}$
$\text{Kill}_2 = \{x\}$ $\text{Kill}_1 = \emptyset$

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.





$\text{Gen}_2 = \emptyset$ $\quad$ $\text{Gen}_1 = \{y\}$

$\text{Kill}_2 = \{x\}$ $\quad$ $\text{Kill}_1 = \emptyset$

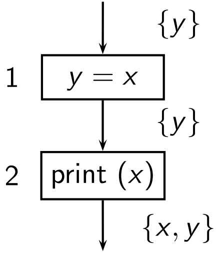
$\text{Gen}_2 = \emptyset$

$\text{Kill}_2 = \{y\}$

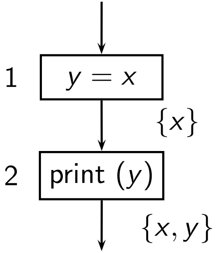
Faintness of x is killed by the print statement (i.e. x becomes live)

Faint Variables Analysis

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



$\text{Gen}_2 = \emptyset$ $\text{Gen}_1 = \{y\}$
$\text{Kill}_2 = \{x\}$ $\text{Kill}_1 = \emptyset$

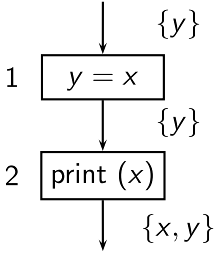


$\text{Gen}_2 = \emptyset$ $\text{Gen}_1 = \{y\}$
$\text{Kill}_2 = \{y\}$ $\text{Kill}_1 = \{x\}$

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

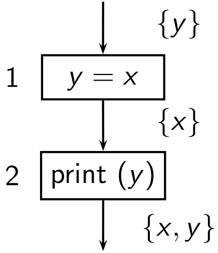
Faint Variables Analysis

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



$\text{Gen}_2 = \emptyset$ $\text{Gen}_1 = \{y\}$
$\text{Kill}_2 = \{x\}$ $\text{Kill}_1 = \emptyset$

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



$\text{Gen}_2 = \emptyset$ $\text{Gen}_1 = \{y\}$
$\text{Kill}_2 = \{y\}$ $\text{Kill}_1 = \{x\}$

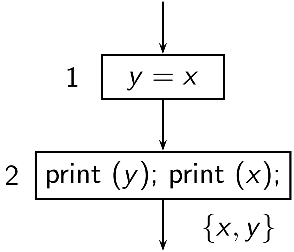
Faintness of x is killed by the assignment to y (i.e. x becomes live)

Faint Variables Analysis

1 $y = x$

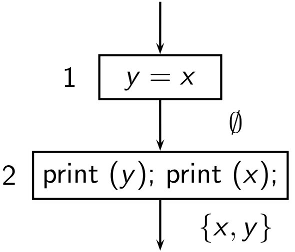
2 print (y); print (x);

Faint Variables Analysis



$\text{Gen}_2 = \emptyset$
$\text{Kill}_2 = \{x, y\}$

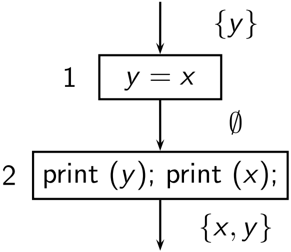
Faint Variables Analysis



$\text{Gen}_2 = \emptyset$ $\quad\quad$ $\text{Gen}_1 = \{y\}$

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

Faint Variables Analysis



$\text{Gen}_2 = \emptyset$ $\text{Gen}_1 = \{y\}$

$\text{Kill}_2 = \{x, y\}$ $\text{Kill}_1 = \{x\}$

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

$$
\begin{aligned}
In_n &= f_n(Out_n) \\
Out_n &= \begin{cases} BI & n \text{ is } End \\ \displaystyle\bigcap_{s \in succ(n)} In_s & \text{otherwise} \end{cases}
\end{aligned}
$$

where,

$$
\begin{aligned}
f_n(X) = \; & (X - (ConstKill_n \cup DepKill_n(X))) \\
& \cup (ConstGen_n \cup DepGen_n(X))
\end{aligned}
$$

and BI contains all local variables

Flow Function Components for Faint Variables Analysis

	Statement		
	$x = e,\ e \in \mathbb{E}\text{xpr}$	$read(x)$ (assigning value from input)	$use(x)$ (not in assignment)
$ConstGen_n$	$x \notin Opd(e) \Rightarrow \{x\}$ $x \in Opd(e) \Rightarrow \emptyset$	$\{x\}$	$\emptyset$
$ConstKill_n$	$\emptyset$	$\emptyset$	$\{x\}$
$DepGen_n(X)$	$\emptyset$	$\emptyset$	$\emptyset$
$DepKill_n(X)$	$x \notin X \Rightarrow Opd(e) \cap \mathbb{V}\text{ar}$ $x \in X \Rightarrow \emptyset$	$\emptyset$	$\emptyset$

Note: For statement $x = e$, $f_n(X)$ is an identity function if $x \in Opd(e)$

Faint Variable Analysis

- What is $\widehat{L}$ 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

$$
\begin{aligned}
&(X_1 \cap X_2) - DepKill_n(X_1 \cap X_2) \\
= \;\; &(X_1 - DepKill_n(X_1)) \;\cap\; (X_2 - DepKill_n(X_2))
\end{aligned}
$$

Distributivity of Faint Variables Analysis

To show that

$$\begin{aligned}&(X_1 \cap X_2) - DepKill_n(X_1 \cap X_2)\\ =\ \ &(X_1 - DepKill_n(X_1)) \ \cap\ (X_2 - DepKill_n(X_2))\end{aligned}$$

- 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.

$$\begin{aligned}&(X_1 \cap X_2) - DepKill_n(X_1 \cap X_2)\\ &= (X_1 \cap X_2) - (Opd(e) \cap \mathbb{V}\text{ar})\\ &= (X_1 - (Opd(e) \cap \mathbb{V}\text{ar})) \ \cap\ (X_2 - (Opd(e) \cap \mathbb{V}\text{ar}))\\ &= (X_1 - DepKill_n(X_1)) \ \cap\ (X_2 - DepKill_n(X_2))\end{aligned}$$

What if x is in X_1 but not in X_2?

Distributivity of Faint Variables Analysis

To show that

$$(X_1 \cap X_2) - DepKill_n(X_1 \cap X_2)$$
$$= \ (X_1 - DepKill_n(X_1)) \ \cap \ (X_2 - DepKill_n(X_2))$$

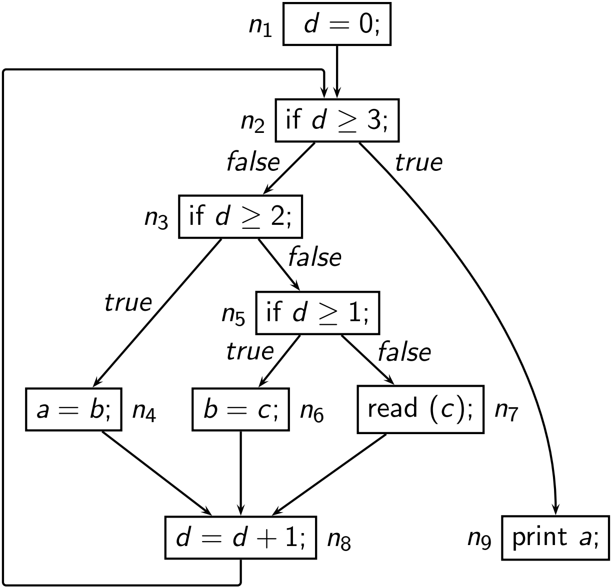
- If n is an assignment statement $x = e$, and $x \notin X_1 \cap X_2$. Assume that x is neither in X_1 nor in X_2.

$$\begin{aligned}
&(X_1 \cap X_2) - DepKill_n(X_1 \cap X_2) \\
&= (X_1 \cap X_2) - (Opd(e) \cap \mathbb{V}\text{ar}) \\
&= (X_1 - (Opd(e) \cap \mathbb{V}\text{ar})) \ \cap \ (X_2 - (Opd(e) \cap \mathbb{V}\text{ar})) \\
&= (X_1 - DepKill_n(X_1)) \ \cap \ (X_2 - DepKill_n(X_2))
\end{aligned}$$

  What if x is in X_1 but not in X_2?

- In all other cases, $DepKill_n(X) = \emptyset$.

Example Program for Faint Variables Analysis



Result of Faint Variables Analysis

Node	Iteration #1		Changes in Iteration #2		Changes in Iteration #3		Changes in Iteration #4	
	Out_n	In_n	Out_n	In_n	Out_n	In_n	Out_n	In_n
n_9	$\{a, b, c, d\}$	$\{b, c, d\}$						
n_8	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{b, c\}$	$\{b, c\}$	$\{c\}$	$\{c\}$	$\emptyset$	$\emptyset$
n_7	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{b, c\}$	$\{b, c\}$	$\{c\}$	$\{c\}$	$\emptyset$	
n_6	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{b, c\}$	$\{b, c\}$	$\{c\}$	$\emptyset$	$\emptyset$	
n_5	$\{a, b, c, d\}$	$\{a, b, c\}$	$\{b, c\}$	$\{b, c\}$	$\emptyset$	$\emptyset$		
n_4	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{b, c\}$	$\{a, c\}$	$\{c\}$		$\emptyset$	$\emptyset$
n_3	$\{a, b, c\}$	$\{a, b, c\}$	$\{c\}$	$\{c\}$	$\emptyset$	$\emptyset$		
n_2	$\{b, c\}$	$\{b, c\}$	$\{c\}$	$\{c\}$	$\emptyset$	$\emptyset$	$\emptyset$	
n_1	$\{b, c\}$	$\{b, c, d\}$	$\{c\}$	$\{c, d\}$	$\emptyset$	$\{d\}$		

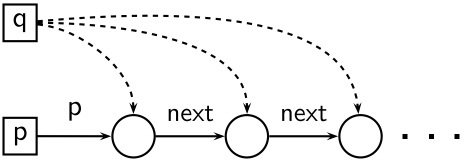
Part 5

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



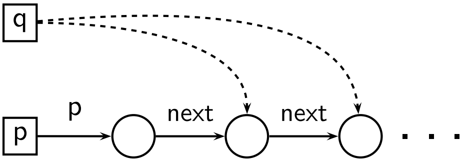
Memory graph at statement 5

- 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↣next;
4.  while (. . . )
5.  p↣data = r1;
6.  print (q↣data);
7.  p↣data = r2;
8.  r4 = p↣data + r3;
```

Program



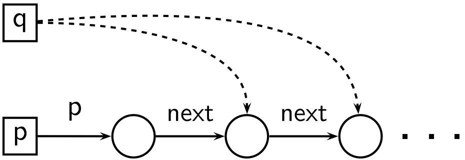
Memory graph at statement 5

- Is p↣data live at the exit of line 5? Can we delete line 5?

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



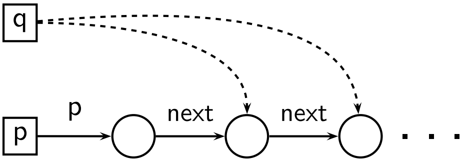
Memory graph at statement 5

- Is p⟼data live at the exit of line 5? Can we delete line 5?
- No, if p and q can be possibly aliased.

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

$$
\boxed{a = 5}
$$

$$
\downarrow
$$

$$
\boxed{x = \&a}
$$

$$
\downarrow
$$

$$
\boxed{b = *x}
$$

Original Program

Code Optimization In Presence of Pointers

$$\begin{array}{c} \boxed{a = 5} \\ \downarrow \\ \boxed{x = \&a} \\ \downarrow \\ \boxed{b = *x} \end{array}$$

Original Program

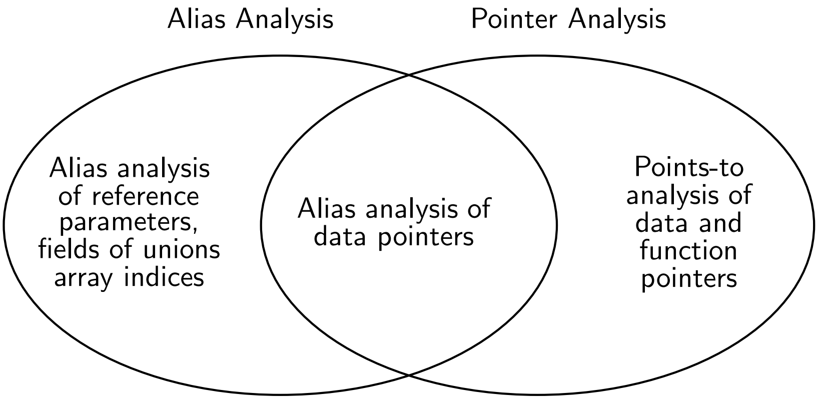
$$\begin{array}{c} \boxed{a = 5} \\ \downarrow \\ \boxed{x = \&a} \\ \downarrow \\ \boxed{b = *x} \end{array}$$

Constant Propagation
without aliasing

Code Optimization In Presence of Pointers

$\boxed{a = 5} \rightarrow \boxed{x = \&a} \rightarrow \boxed{b = *x}$	$\boxed{a = 5} \rightarrow \boxed{x = \&a} \rightarrow \boxed{b = *x}$	$\boxed{a = 5} \rightarrow \boxed{x = \&a} \rightarrow \boxed{b = 5}$
Original Program	Constant Propagation without aliasing	Constant Propagation with aliasing

The World of Pointer Analysis



The Mathematics of Pointer Analysis

In the most general situation

- Alias analysis is undecidable.
  Landi-Ryder [POPL 1991], Landi [LOPLAS 1992],
  Ramalingam [TOPLAS 1994]
- 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.

  $\Rightarrow$ Results in many questionable perceptions.

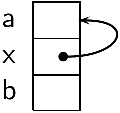
Alias Information Vs. Points-To Information

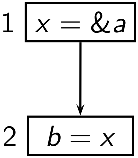
1 $\boxed{x = \&a}$

$\downarrow$

2 $\boxed{b = x}$

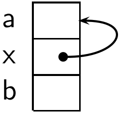
Alias Information Vs. Points-To Information





"*x Points-To a*"
denoted $x \rightarrowtail a$

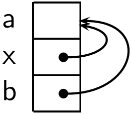
Alias Information Vs. Points-To Information



1 $\boxed{x = \&a}$

"*x Points-To a*"
denoted $x \rightarrowtail a$

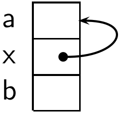
$\downarrow$



2 $\boxed{b = x}$

"*x* and *b* are *Aliases*"
denoted $x \stackrel{\circ}{=} b$

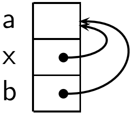
Alias Information Vs. Points-To Information



1 $\boxed{x = \&a}$

"x *Points-To* a"
denoted $x \rightarrowtail a$

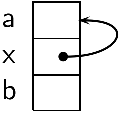
2 $\boxed{b = x}$



"x and b are *Aliases*"
denoted $x \stackrel{\circ}{=} b$

Symmetric
and
Reflexive

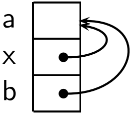
Alias Information Vs. Points-To Information



1 $\boxed{x = \&a}$

"x *Points-To* a"
denoted $x \rightarrowtail a$

Neither
Symmetric
Nor Reflexive

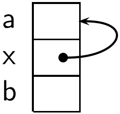


2 $\boxed{b = x}$

"x and b are *Aliases*"
denoted $x \stackrel{\circ}{=} b$

Symmetric
and
Reflexive

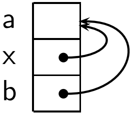
Alias Information Vs. Points-To Information



1 $\boxed{x = \&a}$

"x *Points-To* a"
denoted $x \rightarrowtail a$

Neither Symmetric Nor Reflexive



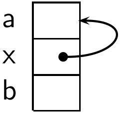
2 $\boxed{b = x}$

"x and b are *Aliases*"
denoted $x \doteq b$

Symmetric and Reflexive

- What about transitivity?

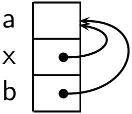
Alias Information Vs. Points-To Information



1 $\boxed{x = \&a}$

"x *Points-To* a"
denoted $x \rightarrowtail a$

Neither Symmetric Nor Reflexive



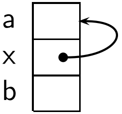
2 $\boxed{b = x}$

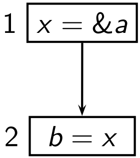
"x and b are *Aliases*"
denoted $x \stackrel{\circ}{=} b$

Symmetric and Reflexive

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

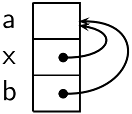
Alias Information Vs. Points-To Information





"*x Points-To a*"
denoted $x \rightarrowtail a$

Neither Symmetric Nor Reflexive

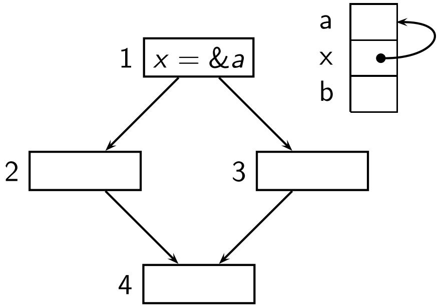


"*x* and *b* are *Aliases*"
denoted $x \stackrel{\circ}{=} b$

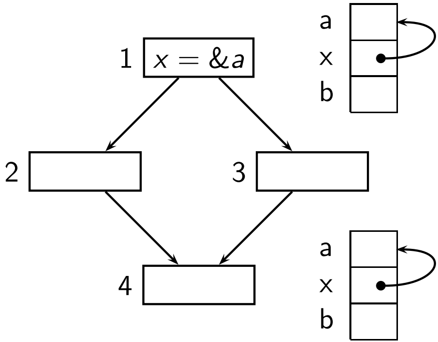
Symmetric and Reflexive

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

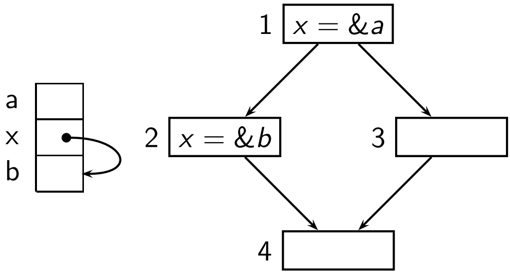
Must Points-To Information



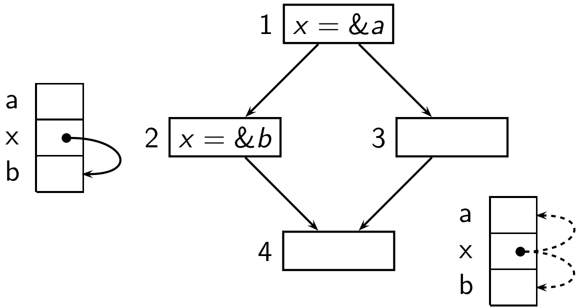
Must Points-To Information



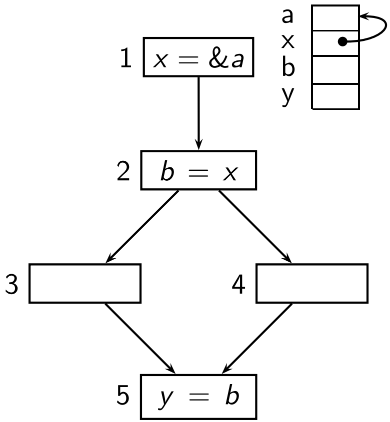
May Points-To Information



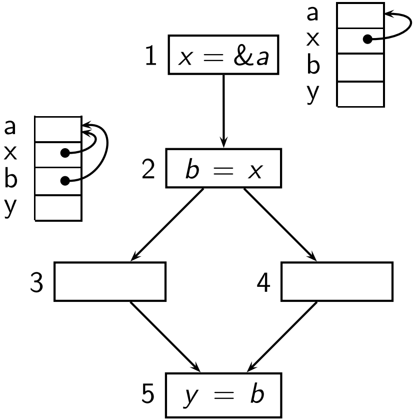
May Points-To Information



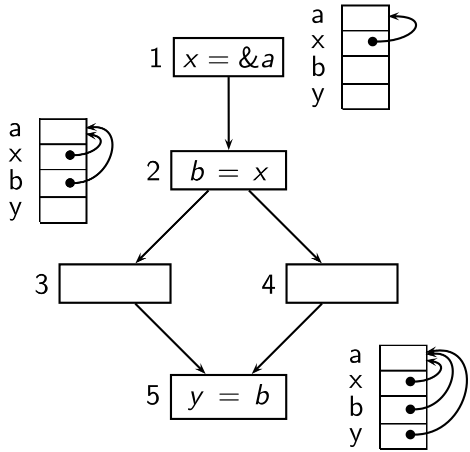
Must Alias Information



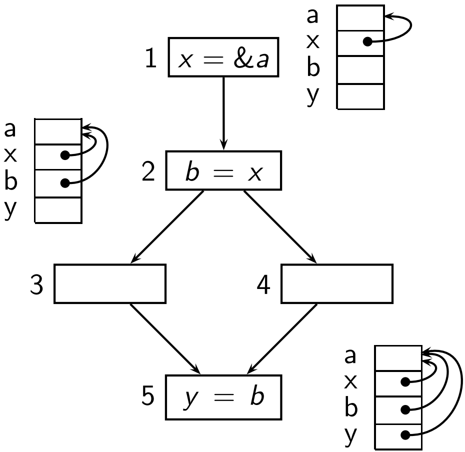
Must Alias Information



Must Alias Information

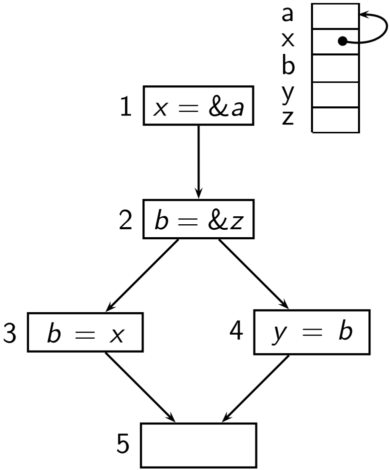


Must Alias Information

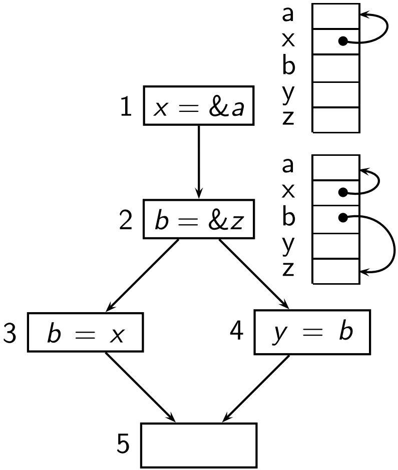


$x \stackrel{\circ}{=} b$ and $b \stackrel{\circ}{=} y \Rightarrow x \stackrel{\circ}{=} y$

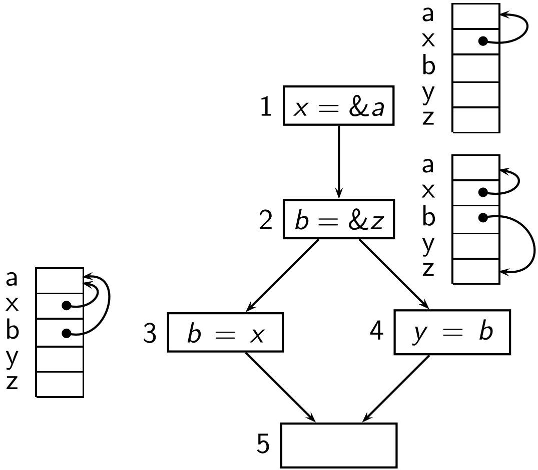
May Alias Information



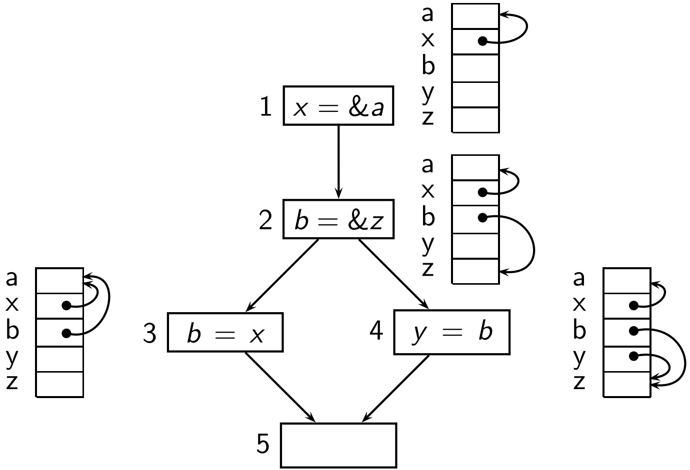
May Alias Information



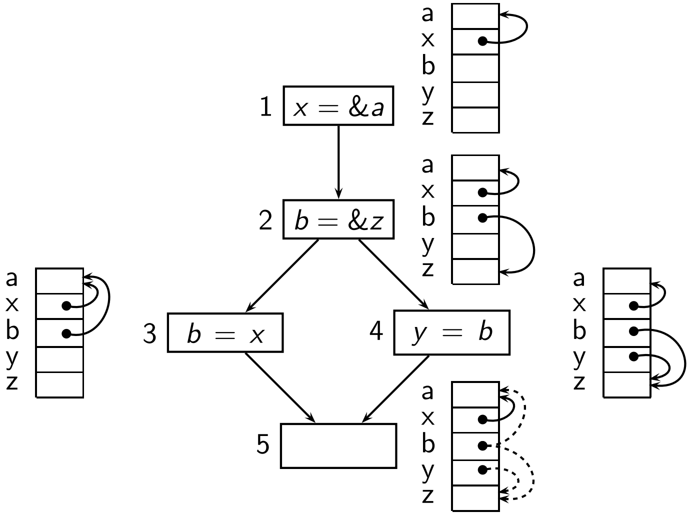
May Alias Information



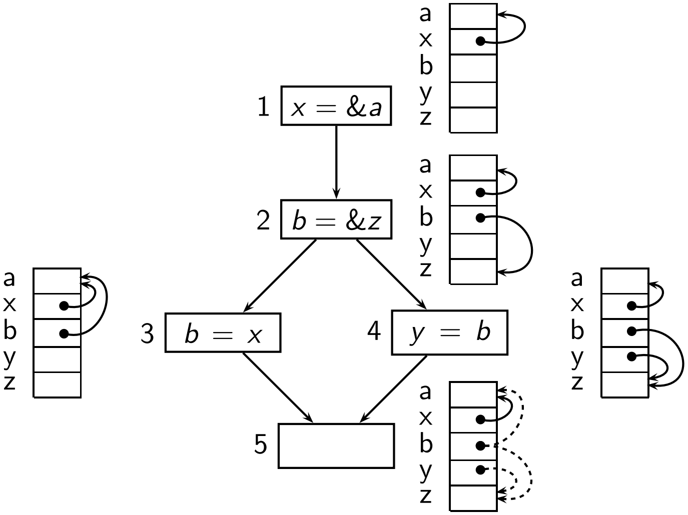
May Alias Information



May Alias Information

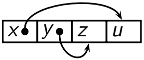
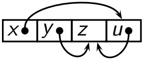
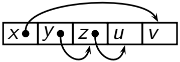
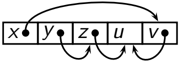


May Alias Information

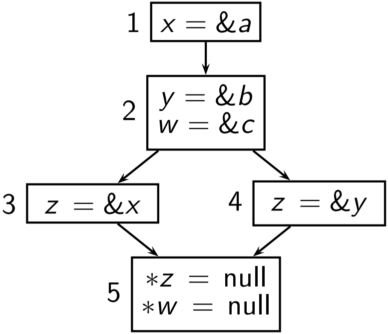


$$x \doteq b \text{ and } b \doteq y \not\Rightarrow x \doteq y$$

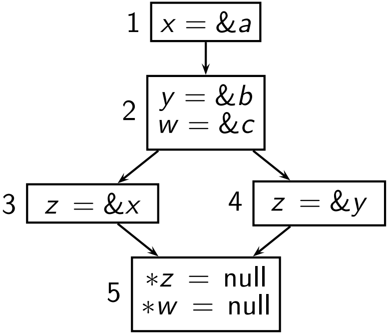
A Comparison of Points-To and Alias Relations

Asgn.	Memory		Points-to		Aliases	
$*x = y$	Before		Existing	$x \rightarrowtail u$ $y \rightarrowtail z$	Existing	$*x \stackrel{\circ}{=} u$ $*y \stackrel{\circ}{=} z$
	After		New	$u \rightarrowtail z$	New Direct	$*x \stackrel{\circ}{=} y$ $y \stackrel{\circ}{=} u$
					New Indirect	$*u \stackrel{\circ}{=} z$ $**x \stackrel{\circ}{=} z$
$*x = *y$	Before		Existing	$x \rightarrowtail v$ $y \rightarrowtail z$ $z \rightarrowtail u$	Existing	$*x \stackrel{\circ}{=} v$ $*y \stackrel{\circ}{=} z$ $*z \stackrel{\circ}{=} u$ $**y \stackrel{\circ}{=} u$
	After		New	$v \rightarrowtail u$	New Direct	$*x \stackrel{\circ}{=} *y$ $*x \stackrel{\circ}{=} z$ $v \stackrel{\circ}{=} z$ $v \stackrel{\circ}{=} *y$
					New Indirect	$**x \stackrel{\circ}{=} u$ $*v \stackrel{\circ}{=} u$

Strong and Weak Updates

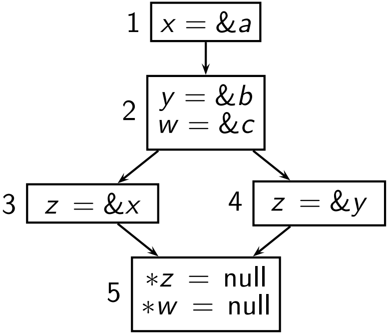


Strong and Weak Updates



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

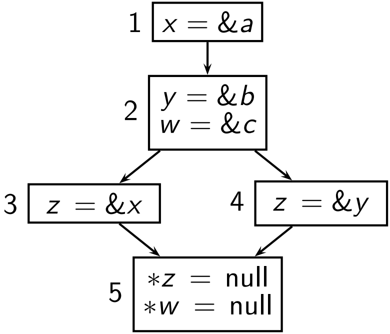
Strong and Weak Updates



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

Left Locations		
lhs_n	$ConstLeftL_n$	$DepLeftL_n(X)$
x	$\{x\}$	$\emptyset$
$*x$	$\emptyset$	$\{y \mid (x \rightarrowtail y) \in X\}$

- Right Locations

Right Locations		
rhs_n	$ConstRightL_n$	$DepRightL_n(X)$
x	$\emptyset$	$\{y \mid (x \rightarrowtail y) \in X\}$
$*x$	$\emptyset$	$\{z \mid \{x \rightarrowtail y, y \rightarrowtail z\} \subseteq X\}$
$\&x$	$\{x\}$	$\emptyset$

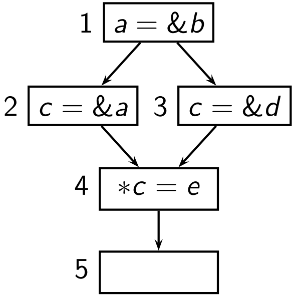
Gen and Kill Components

$$
\begin{aligned}
ConstGen_n &= \{x \rightarrowtail y \mid x \in ConstLeftL_n, y \in ConstRightL_n\} \\
DepGen_n(X) &= \{x \rightarrowtail y \mid (x \in ConstLeftL_n, y \in DepRightL_n(X)), \text{ or} \\
&\qquad\qquad (x \in DepLeftL_n(X), y \in ConstRightL_n), \text{ or} \\
&\qquad\qquad (x \in DepLeftL_n(X), y \in DepRightL_n(X))\} \\
ConstKill_n &= \{x \rightarrowtail y \mid x \in ConstLeftL_n\} \\
DepKill_n(X) &= \{x \rightarrowtail y \mid x \in DepLeftL_n(X)\}
\end{aligned}
$$

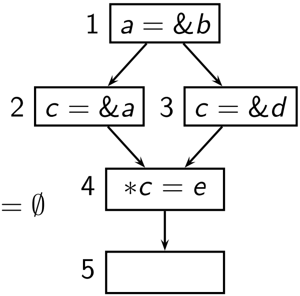
DepKill(X) in May and Must Points-To Analysis

- 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

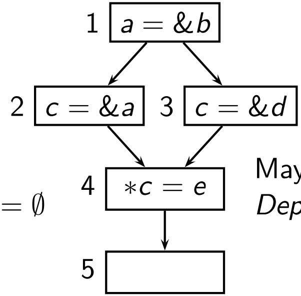


$DepKill(X)$ in May and Must Points-To Analysis



$\mathrm{MustIn}_4 = \{a \rightarrowtail b\}$
$DepLeftL_4(MustIn_4) = \emptyset$

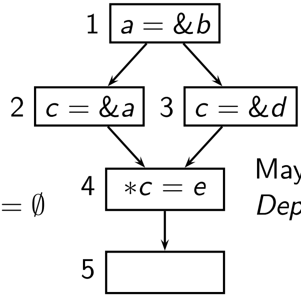
$DepKill(X)$ in May and Must Points-To Analysis



$\text{MustIn}_4 = \{a \rightarrowtail b\}$
$DepLeftL_4(MustIn_4) = \emptyset$

$\text{MayIn}_4 = \{a \rightarrowtail b, c \rightarrowtail a, c \rightarrowtail d\}$
$DepLeftL_4(MayIn_4) = \{a, d\}$

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



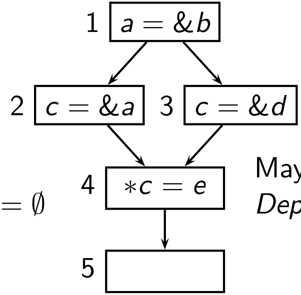
$\text{MustIn}_4 = \{a \rightarrowtail b\}$
$DepLeftL_4(MustIn_4) = \emptyset$

$\text{MayIn}_4 = \{a \rightarrowtail b, c \rightarrowtail a, c \rightarrowtail d\}$
$DepLeftL_4(MayIn_4) = \{a, d\}$

- $a \rightarrowtail b$ at block 5 along path $1, 3, 4, 5$ but not along path $1, 2, 4, 5$.

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

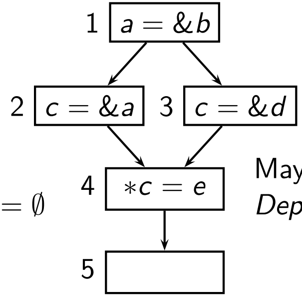
$\text{MustIn}_4 = \{a \rightarrowtail b\}$
$DepLeftL_4(MustIn_4) = \emptyset$



$\text{MayIn}_4 = \{a \rightarrowtail b, c \rightarrowtail a, c \rightarrowtail d\}$
$DepLeftL_4(MayIn_4) = \{a, d\}$

- $a \rightarrowtail b$ at block 5 along path $1, 3, 4, 5$ but not along path $1, 2, 4, 5$.
- $a \rightarrowtail b \in MayIn_5$ but $a \rightarrowtail b \notin MustIn_5$

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

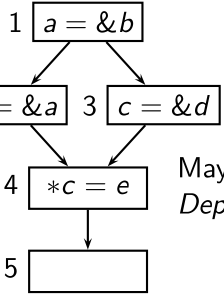


$\text{MustIn}_4 = \{a \rightarrowtail b\}$
$DepLeftL_4(MustIn_4) = \emptyset$

$\text{MayIn}_4 = \{a \rightarrowtail b, c \rightarrowtail a, c \rightarrowtail d\}$
$DepLeftL_4(MayIn_4) = \{a, d\}$

- $a \rightarrowtail b$ at block 5 along path $1, 3, 4, 5$ but not along path $1, 2, 4, 5$.
- $a \rightarrowtail b \in MayIn_5$ but $a \rightarrowtail b \notin MustIn_5$
- If $DepKill_n$ for $MayOut_4$ is defined in terms of $MayIn_4$ then $a \rightarrowtail b \notin MayOut_4$ because a is in $DepLeftL_4(MayIn_4)$

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



$\text{MustIn}_4 = \{a \rightarrowtail b\}$
$DepLeftL_4(MustIn_4) = \emptyset$

$\text{MayIn}_4 = \{a \rightarrowtail b, c \rightarrowtail a, c \rightarrowtail d\}$
$DepLeftL_4(MayIn_4) = \{a, d\}$

- $a \rightarrowtail b$ at block 5 along path $1, 3, 4, 5$ but not along path $1, 2, 4, 5$.
- $a \rightarrowtail b \in MayIn_5$ but $a \rightarrowtail b \notin MustIn_5$
- If $DepKill_n$ for $MayOut_4$ is defined in terms of $MayIn_4$ then $a \rightarrowtail b \notin MayOut_4$ because a is in $DepLeftL_4(MayIn_4)$
- If $DepKill_4$ for $MustOut_4$ is defined in terms of $MustIn_4$ then $a \rightarrowtail b \in MustOut_4$ because a is not in $DepLeftL_4(MustIn_4)$

Data Flow Equations for Points-To Analysis

$$
\begin{aligned}
MayIn_n &= \begin{cases} BI & n \text{ is } Start \\ \bigcup\limits_{p \in pred(n)} MayOut_n & \text{otherwise} \end{cases} \\
MayOut_n &= f_n(MayIn_n, MustIn_n) \\
MustIn_n &= \begin{cases} BI & n \text{ is } Start \\ \bigcap\limits_{p \in pred(n)} MustOut_n & \text{otherwise} \end{cases} \\
MustOut_n &= f_n(MustIn_n, MayIn_n)
\end{aligned}
$$

$$
f_n(X_1, X_2) = (X_1 - \mathrm{Kill}_n(X_2)) \cup \mathrm{Gen}_n(X_1)
$$

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 alised 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 alised 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 alised 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

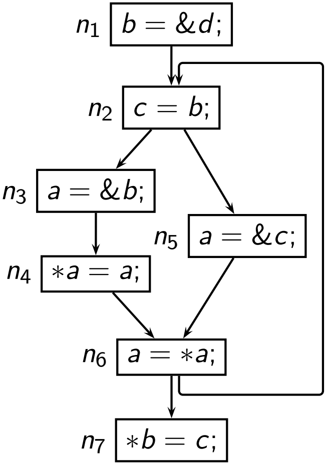
- May Alias: Every pointer variable is aliased to every pointer variable.
- Must Alias: Every pointer variable is alised only to itself.
- May Points-To: Every pointer variable points to every location.
- Must Points-To: No pointer variable points to any location.
- Both May and Must analyses need not 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 alised 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 $\perp$ element of the lattice.

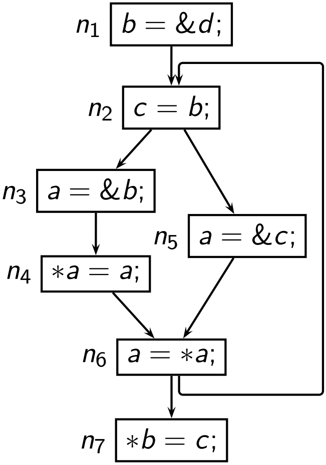
Example Program for Points-To Analysis



- Variables and points-to sets:

$$
\begin{aligned}
\mathbb{V}\text{ar} &= \{a, b, c, d\} \\
\mathbb{U} &= \{\ a \rightarrowtail a,\ a \rightarrowtail b,\ a \rightarrowtail c,\ a \rightarrowtail d, \\
&\qquad b \rightarrowtail a,\ b \rightarrowtail b,\ b \rightarrowtail d,\ b \rightarrowtail d, \\
&\qquad c \rightarrowtail a,\ c \rightarrowtail b,\ c \rightarrowtail c,\ c \rightarrowtail d, \\
&\qquad d \rightarrowtail a,\ d \rightarrowtail b,\ d \rightarrowtail c,\ d \rightarrowtail d\ \}
\end{aligned}
$$

Example Program for Points-To Analysis

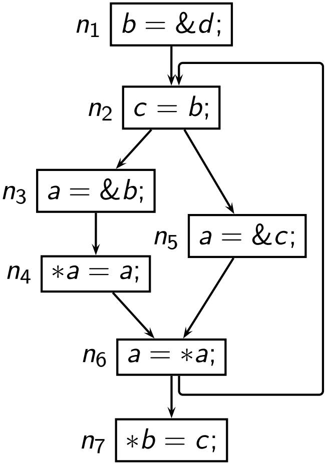


- Variables and points-to sets:

$$
\begin{array}{rcl}
\mathbb{V}\text{ar} & = & \{a, b, c, d\} \\
\mathbb{U} & = & \{\ a{\rightarrowtail}a,\ a{\rightarrowtail}b,\ a{\rightarrowtail}c,\ a{\rightarrowtail}d, \\
 & & \ \ b{\rightarrowtail}a,\ b{\rightarrowtail}b,\ b{\rightarrowtail}d,\ b{\rightarrowtail}d, \\
 & & \ \ c{\rightarrowtail}a,\ c{\rightarrowtail}b,\ c{\rightarrowtail}c,\ c{\rightarrowtail}d, \\
 & & \ \ d{\rightarrowtail}a,\ d{\rightarrowtail}b,\ d{\rightarrowtail}c,\ d{\rightarrowtail}d\ \}
\end{array}
$$

- $L_{may} = \langle 2^{\mathbb{U}}, \supseteq \rangle,\ \top_{may} = \emptyset, \bot_{may} = \mathbb{U}$

Example Program for Points-To Analysis



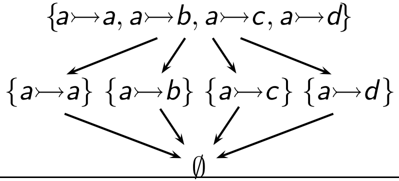
- Variables and points-to sets:

$$\mathbb{V}\text{ar} = \{a, b, c, d\}$$

$$\begin{aligned}\mathbb{U} = \{\ & a \rightarrowtail a,\ a \rightarrowtail b,\ a \rightarrowtail c,\ a \rightarrowtail d,\\ & b \rightarrowtail a,\ b \rightarrowtail b,\ b \rightarrowtail d,\ b \rightarrowtail d,\\ & c \rightarrowtail a,\ c \rightarrowtail b,\ c \rightarrowtail c,\ c \rightarrowtail d,\\ & d \rightarrowtail a,\ d \rightarrowtail b,\ d \rightarrowtail c,\ d \rightarrowtail d\ \}\end{aligned}$$

- $L_{may} = \langle 2^{\mathbb{U}}, \supseteq \rangle,\ \top_{may} = \emptyset, \bot_{may} = \mathbb{U}$

- $L_{must} = \widehat{L}_a \times \widehat{L}_b \times \widehat{L}_c \times \widehat{L}_d$
  The component lattice $\widehat{L}_a$ is:



Result of Pointer Analysis

	Iteration #1	Changes in Iteration #2	Changes in Iteration #3
$MayIn_{n_1}$	$\emptyset$		
$MustIn_{n_1}$	$\emptyset$		
$MayOut_{n_1}$	$\{b \rightarrowtail d\}$		
$MustOut_{n_1}$	$\{b \rightarrowtail d\}$		
$MayIn_{n_2}$	$\{b \rightarrowtail d\}$	$\{a \rightarrowtail b, a \rightarrowtail d, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail d\}$	$\{a \rightarrowtail b, a \rightarrowtail d, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail b, c \rightarrowtail d\}$
$MustIn_{n_2}$	$\{b \rightarrowtail d\}$	$\emptyset$	
$MayOut_{n_2}$	$\{b \rightarrowtail d, c \rightarrowtail d\}$	$\{a \rightarrowtail b, a \rightarrowtail d, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail b, c \rightarrowtail d\}$	
$MustOut_{n_2}$	$\{b \rightarrowtail d, c \rightarrowtail d\}$	$\emptyset$	
$MayIn_{n_3}$	$\{b \rightarrowtail d, c \rightarrowtail d\}$	$\{a \rightarrowtail b, a \rightarrowtail d, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail b, c \rightarrowtail d\}$	
$MustIn_{n_3}$	$\{b \rightarrowtail d, c \rightarrowtail d\}$	$\emptyset$	
$MayOut_{n_3}$	$\{a \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail d\}$	$\{a \rightarrowtail b, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail b, c \rightarrowtail d\}$	
$MustOut_{n_3}$	$\{a \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail d\}$	$\{a \rightarrowtail b\}$	

Result of Pointer Analysis

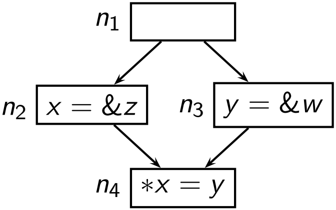
	Iteration #1	Changes in Iteration #2	Changes in Iteration #3
$MayIn_{n_4}$	$\{a \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail d\}$	$\{a \rightarrowtail b, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail b, c \rightarrowtail d\}$	
$MustIn_{n_4}$	$\{a \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail d\}$	$\{a \rightarrowtail b\}$	
$MayOut_{n_4}$	$\{a \rightarrowtail b, b \rightarrowtail b, c \rightarrowtail d\}$	$\{a \rightarrowtail b, b \rightarrowtail b, c \rightarrowtail b, c \rightarrowtail d\}$	
$MustOut_{n_4}$	$\{a \rightarrowtail b, b \rightarrowtail b, c \rightarrowtail d\}$	$\{a \rightarrowtail b, b \rightarrowtail b\}$	
$MayIn_{n_5}$	$\{b \rightarrowtail d, c \rightarrowtail d\}$	$\{a \rightarrowtail b, a \rightarrowtail d, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail b, c \rightarrowtail d\}$	
$MustIn_{n_5}$	$\{b \rightarrowtail d, c \rightarrowtail d\}$	$\emptyset$	
$MayOut_{n_5}$	$\{a \rightarrowtail c, b \rightarrowtail d, c \rightarrowtail d\}$	$\{a \rightarrowtail c, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail b, c \rightarrowtail d\}$	
$MustOut_{n_5}$	$\{a \rightarrowtail c, b \rightarrowtail d, c \rightarrowtail d\}$	$\{a \rightarrowtail c\}$	

Result of Pointer Analysis

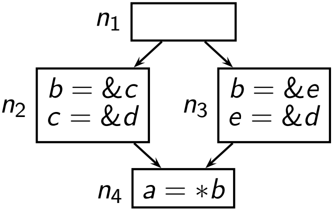
	Iteration #1	Changes in Iteration #2	Changes in Iteration #3
$MayIn_{n_6}$	$\{a \rightarrowtail b, a \rightarrowtail c, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail d\}$	$\{a \rightarrowtail b, a \rightarrowtail c, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail b, c \rightarrowtail d\}$	
$MustIn_{n_6}$	$\{c \rightarrowtail d\}$	$\emptyset$	
$MayOut_{n_6}$	$\{a \rightarrowtail b, a \rightarrowtail d, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail d\}$	$\{a \rightarrowtail b, a \rightarrowtail d, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail b, c \rightarrowtail d\}$	
$MustOut_{n_6}$	$\{c \rightarrowtail d\}$	$\emptyset$	
$MayIn_{n_7}$	$\{a \rightarrowtail b, a \rightarrowtail d, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail d\}$	$\{a \rightarrowtail b, a \rightarrowtail d, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail b, c \rightarrowtail d\}$	
$MustIn_{n_7}$	$\{c \rightarrowtail d\}$	$\emptyset$	
$MayOut_{n_7}$	$\{a \rightarrowtail b, a \rightarrowtail d, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail d, d \rightarrowtail d\}$	$\{a \rightarrowtail b, a \rightarrowtail d, b \rightarrowtail b, b \rightarrowtail d, c \rightarrowtail b, c \rightarrowtail d, d \rightarrowtail b, d \rightarrowtail d\}$	
$MustOut_{n_7}$	$\{c \rightarrowtail d\}$	$\emptyset$	

Non-Distributivity of Points-To Analysis

May Points-To

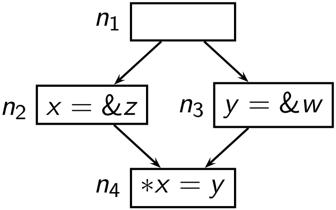


Must Points-To



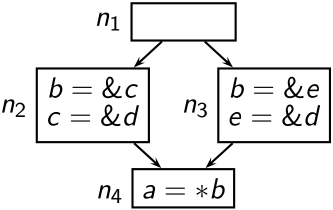
Non-Distributivity of Points-To Analysis

May Points-To



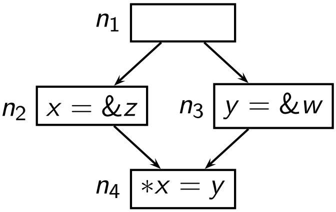
$z \rightarrowtail w$ is spurious

Must Points-To



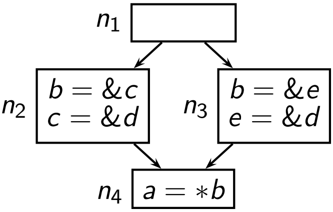
Non-Distributivity of Points-To Analysis

May Points-To



$z \rightarrowtail w$ is spurious

Must Points-To



$a \rightarrowtail d$ is missing

Part 6

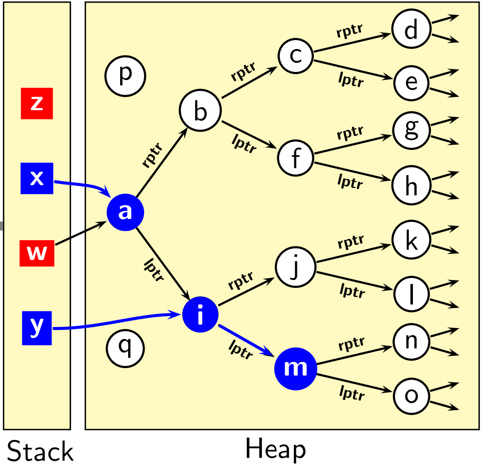
Heap Reference Analysis

Motivating Example for Heap Liveness Analysis

If the while loop is not executed even once.

```
w = x            // x points to m_a
while  (x.data < max)
        x = x.rptr
y = x.lptr

z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
```

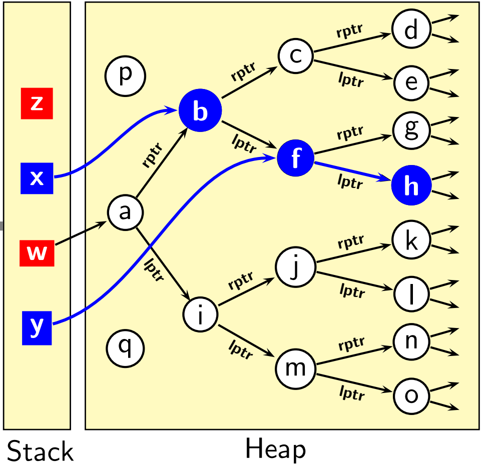


Motivating Example for Heap Liveness Analysis

If the while loop is executed once.

```
w = x          // x points to m_a
while  (x.data < max)
        x = x.rptr
y = x.lptr

z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
```

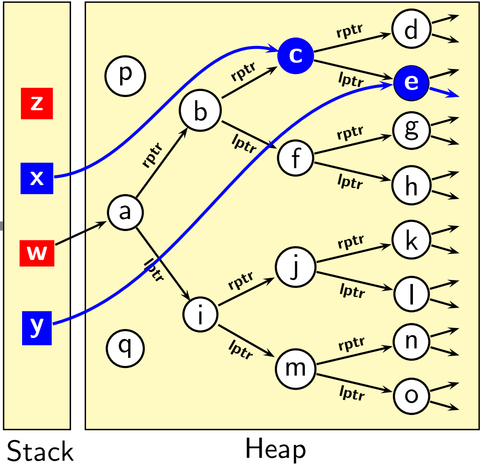


Motivating Example for Heap Liveness Analysis

If the while loop is executed twice.

```
w = x            // x points to m_a
while  (x.data < max)
        x = x.rptr
y = x.lptr

z = New  class_of_z
y = y.lptr
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?

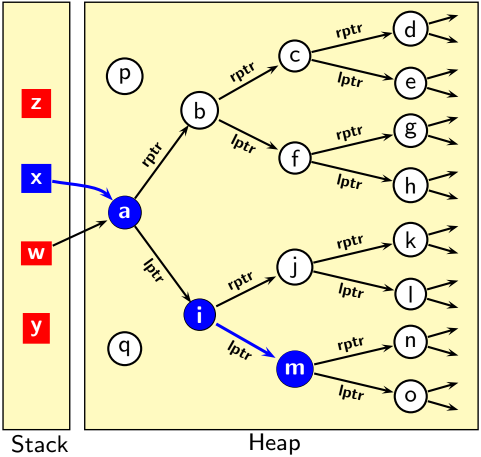
Our Solution

	Statement	Null assignment
		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

Our Solution

```
y = z = null
w = x
w = null
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
y = x.lptr
x.lptr = y.rptr = null
y.lptr.lptr = y.lptr.rptr = null
z = New class_of_z
z.lptr = z.rptr = null
y = y.lptr
y.lptr = y.rptr = null
z.sum = x.data + y.data
x = y = z = null
```

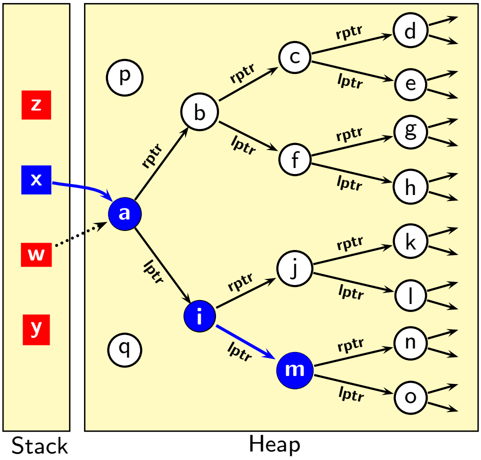
While loop is not executed even once



Our Solution

```
y = z = null
w = x
w = null
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
y = x.lptr
x.lptr = y.rptr = null
y.lptr.lptr = y.lptr.rptr = null
z = New class_of_z
z.lptr = z.rptr = null
y = y.lptr
y.lptr = y.rptr = null
z.sum = x.data + y.data
x = y = z = null
```

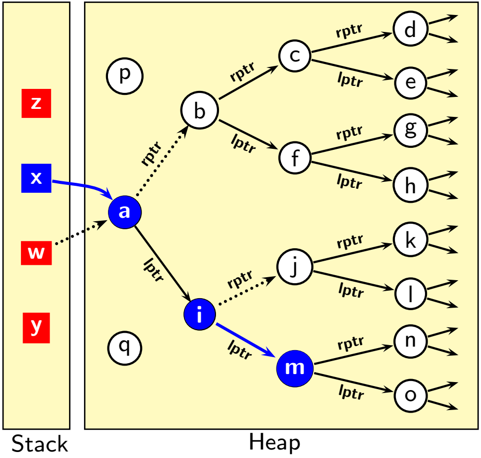
While loop is not executed even once



Our Solution

```
  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
```

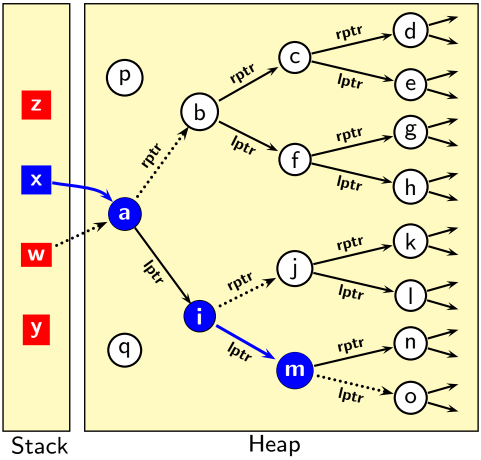
While loop is not executed even once



Our Solution

```
    y = z = 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
```

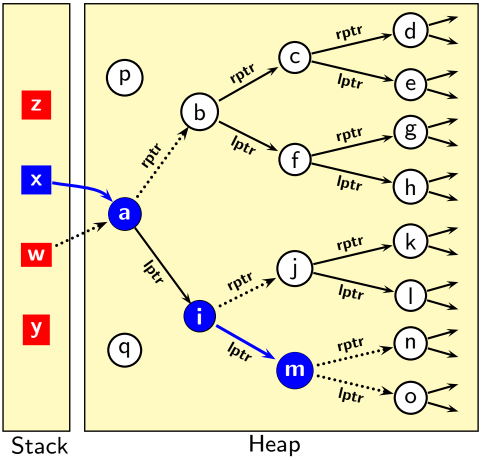
While loop is not executed even once



Our Solution

```
    y = z = 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
```

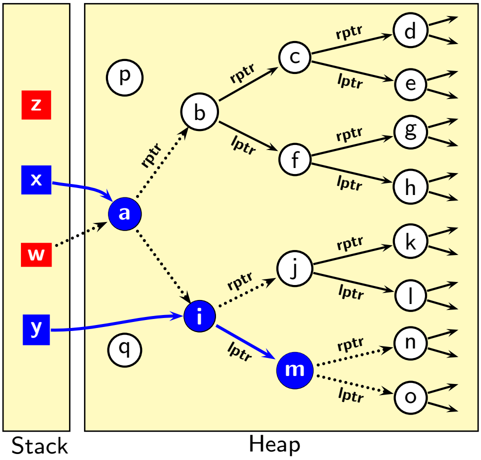
While loop is not executed even once



Our Solution

```
  y = z = 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
```

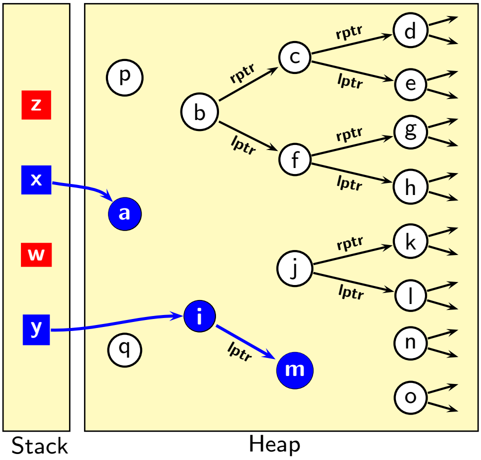
While loop is not executed even once



Our Solution

```
     y = z = 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
```

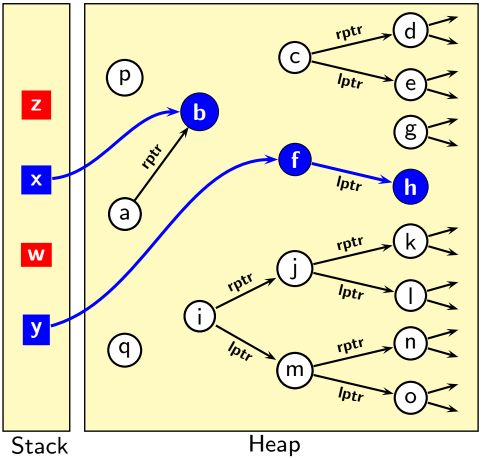
While loop is not executed even once



Our Solution

```
y = z = null
w = x
w = null
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
y = x.lptr
x.lptr = y.rptr = null
y.lptr.lptr = y.lptr.rptr = null
z = New class_of_z
z.lptr = z.rptr = null
y = y.lptr
y.lptr = y.rptr = null
z.sum = x.data + y.data
x = y = z = null
```

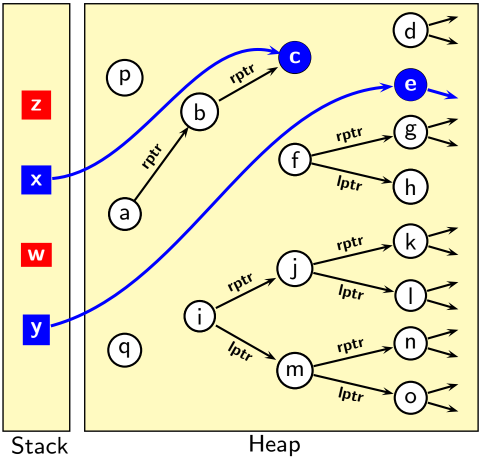
While loop is executed once



Our Solution

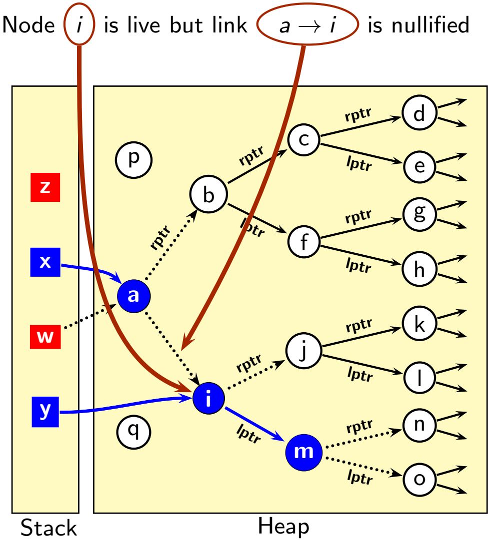
```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr        }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = 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 executed twice



Some Observations

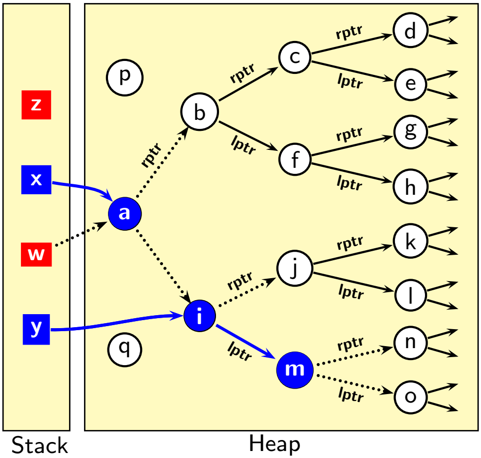
```
  y = z = null
1 w = x
  w = null
2 while (x.data < max)
  {      x.lptr = null
3        x = x.rptr        }
  x.rptr = x.lptr.rptr = null
  x.lptr.lptr.lptr = null
  x.lptr.lptr.rptr = null
4 y = x.lptr
  x.lptr = y.rptr = null
  y.lptr.lptr = y.lptr.rptr = null
5 z = New class_of_z
  z.lptr = z.rptr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



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
```

New access expressions are created.
Can they cause exceptions?



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

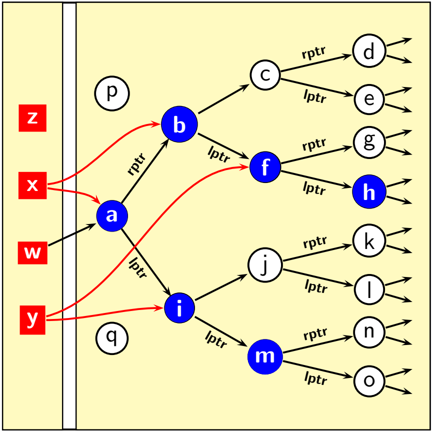
  - *Alias Analysis*. Identify all possible ways of accessing a link
  - *Liveness Analysis*. For each program point, identify "dead" links (i.e. links which are not accessed after that program point)
  - *Availability and Anticipability Analyses*. Dead links should be reachable for making null assignment.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



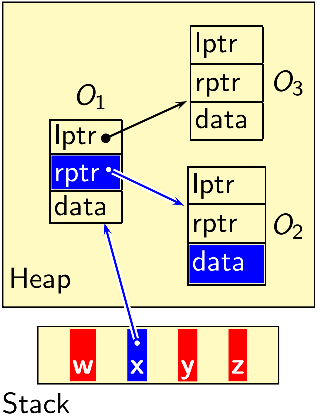
- Root variables : x, y, z
- Field names : rptr, lptr
- Access path : x➔rptr➔lptr
  Semantically, sequence of "links"
- Frontier : name of the last link
- Live access path : If the link corresponding to its frontier is used in future

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:

Example	Objects read	Live access paths
sum = x.rptr.data	x, O_1, O_2	$x, x\rightarrow$rptr
if (x.rptr.data < sum)	x, O_1, O_2	$x, x\rightarrow$rptr

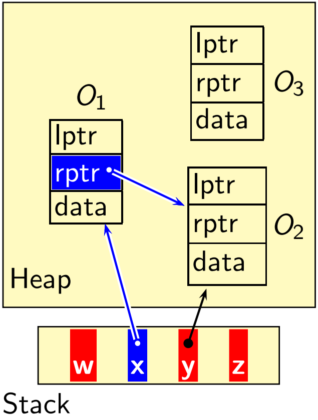


What Makes a Link Live?

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:

Example	Objects read	Live access paths
$y = x$.rptr	x, O_1	x

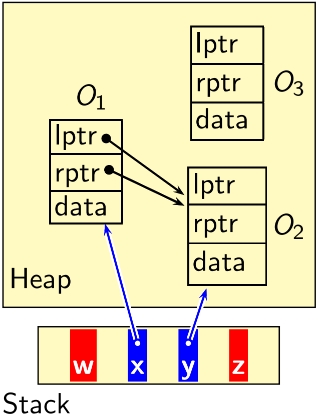


What Makes a Link Live?

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:

Example	Objects read	Live access paths
$y = x.\text{rptr}$	x, O_1	x
$x.\text{lptr} = y$	x, O_1, y	x

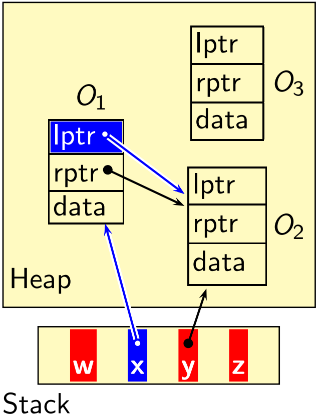


What Makes a Link Live?

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:

Example	Objects read	Live access paths
if (x.lptr == null)	x, O_1	$x, x{\rightarrow}$lptr

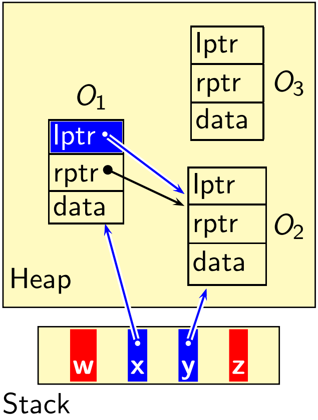


What Makes a Link Live?

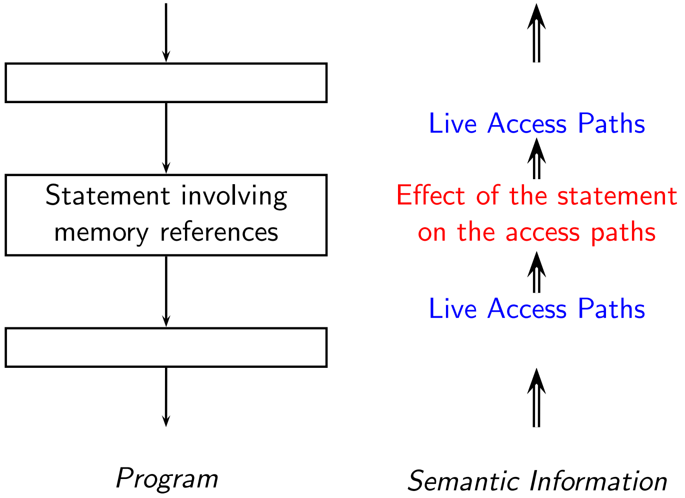
Assuming that a statement is the last statement in the program, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

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

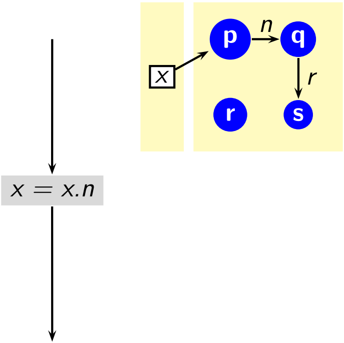
Example	Objects read	Live access paths
if (x.lptr == null)	x, O_1	$x, x \rightarrow$lptr
if (y == x.lptr)	x, O_1, y	$x, x \rightarrow$lptr$, y$



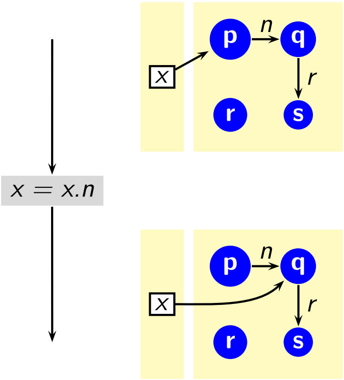
Liveness Analysis



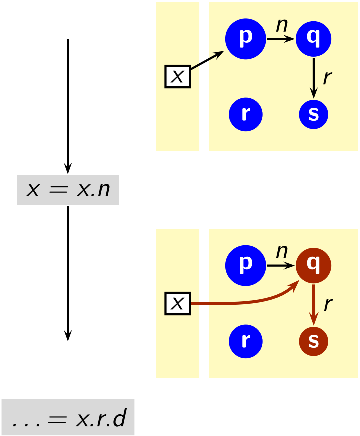
Key Idea #2 : Transfer of Access Paths



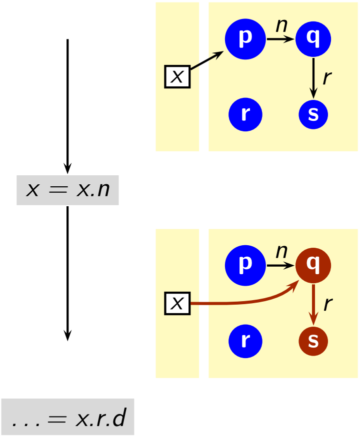
Key Idea #2 : Transfer of Access Paths



Key Idea #2 : Transfer of Access Paths

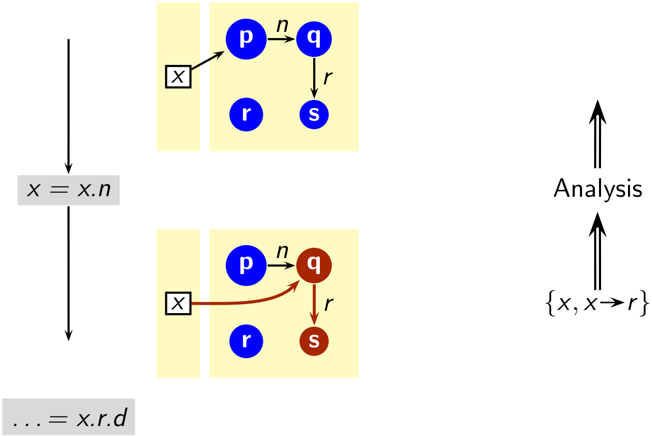


Key Idea #2 : Transfer of Access Paths

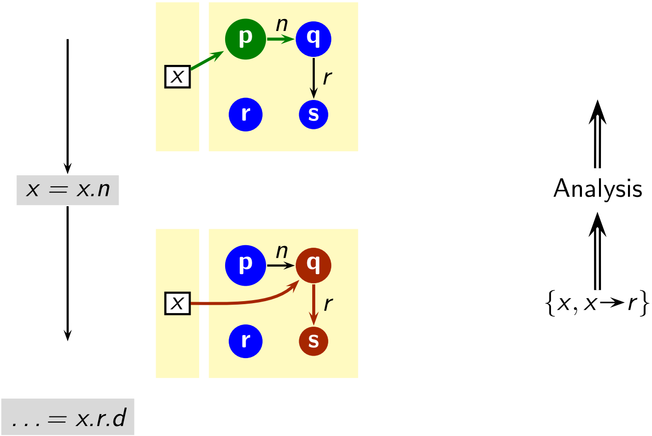


$\{x, x \rightarrow r\}$

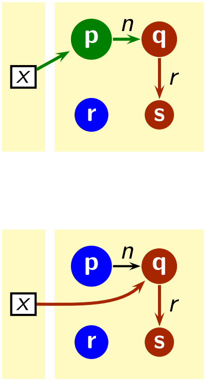
Key Idea #2 : Transfer of Access Paths



Key Idea #2 : Transfer of Access Paths



Key Idea #2 : Transfer of Access Paths



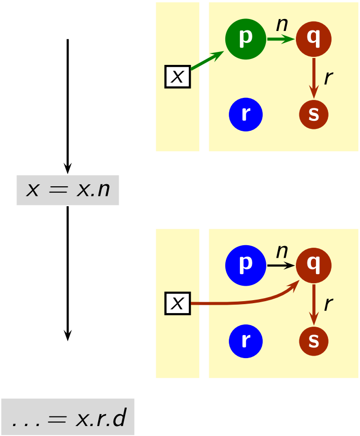
$x = x.n$

Analysis

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

$\ldots = x.r.d$

Key Idea #2 : Transfer of Access Paths



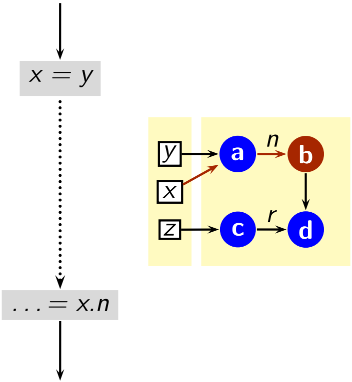
Generated	
Constant	$\{x\}$
Dependent	$\{x \rightarrow n, x \rightarrow n \rightarrow r\}$
Killed	
Constant	$\{x, x \rightarrow r\}$
Dependent	$\emptyset$

Analysis

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

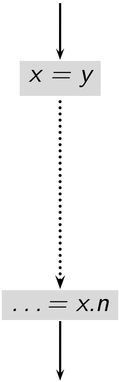
x after the assinment is same as the $x \rightarrow n$ before the assignment

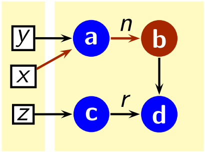
Key Idea #3 : Liveness Closure Under Link Aliasing



x and y are node aliases

Key Idea #3 : Liveness Closure Under Link Aliasing

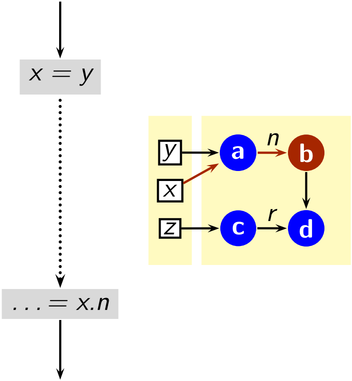




x and y are node aliases

$x.n$ and $y.n$ are link aliases

Key Idea #3 : Liveness Closure Under Link Aliasing

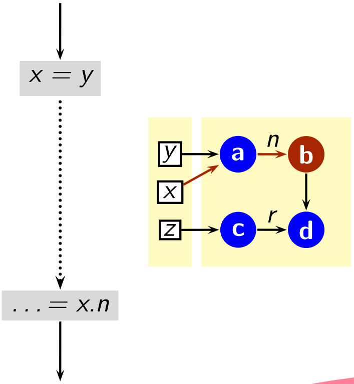


x and y are node aliases

$x.n$ 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



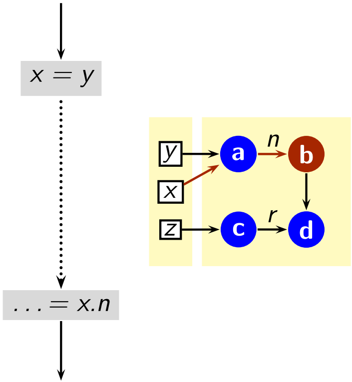
x and y are node aliases

$x.n$ and $y.n$ are link aliases

$x \rightarrow n$ is live $\Rightarrow$ $y \rightarrow n$ is live

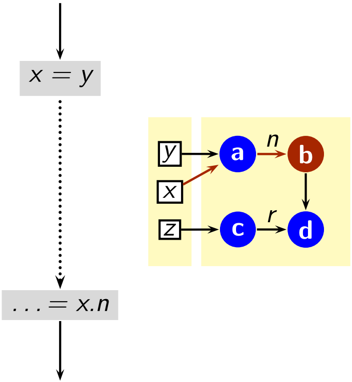
Nullifying $y \rightarrow n$ will have the side effect of nullifying $x \rightarrow n$

Explicit and Implicit Liveness



$x \rightarrow n$ is live $\Rightarrow$ $y \rightarrow n$ is live

Explicit and Implicit Liveness



$x \rightarrow n$ 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

$$
\begin{aligned}
\mathit{base}(\rho_x) &= \text{longest proper prefix of } \rho_x \\
\mathit{prefixes}(\rho_x) &= \{\rho'_x \mid \rho'_x \text{ is a prefixe of } \rho_x\} \\
\mathit{summary}(S) &= \{\rho_x \rightarrow * \mid \rho_x \in S\}
\end{aligned}
$$

ρ_x	frontier(ρ_x)	$\mathit{base}(\rho_x)$	$\mathit{prefixes}(\rho_x)$	$\mathit{summary}(\{\rho_x\})$
$x \rightarrow n \rightarrow r$	r	$x \rightarrow n$	$\{x, x \rightarrow n, x \rightarrow n \rightarrow r\}$	$\{x \rightarrow n \rightarrow r \rightarrow *\}$
$x \rightarrow r \rightarrow n$	n	$x \rightarrow r$	$\{x, x \rightarrow r, x \rightarrow r \rightarrow n\}$	$\{x \rightarrow r \rightarrow n \rightarrow *\}$
$x \rightarrow n$	n	x	$\{x, x \rightarrow n\}$	$\{x \rightarrow n \rightarrow *\}$
$x \rightarrow r$	r	x	$\{x, x \rightarrow r\}$	$\{x \rightarrow r \rightarrow *\}$
x	x	$\mathcal{E}$	$\{x\}$	$\{x \rightarrow *\}$

empty access path

0 or more occurrences
of any field name

Notation for Defining Flow Functions for Explicit Liveness

$$\begin{array}{rcl} \mathit{base}(\rho_x) & = & \text{longest proper prefix of } \rho_x \\ \mathit{prefixes}(\rho_x) & = & \{\rho'_x \mid \rho'_x \text{ is a prefixe of } \rho_x\} \\ \mathit{summary}(S) & = & \{\rho_x \rightarrow * \mid \rho_x \in S\} \end{array}$$

ρ_x	frontier(ρ_x)	$\mathit{base}(\rho_x)$	$\mathit{prefixes}(\rho_x)$	$\mathit{summary}(\{\rho_x\})$
$x \rightarrow n \rightarrow r$	r	$x \rightarrow n$	$\{x, x \rightarrow n, x \rightarrow n \rightarrow r\}$	$\{x \rightarrow n \rightarrow r \rightarrow *\}$
$x \rightarrow r \rightarrow n$	n	$x \rightarrow r$	$\{x, x \rightarrow r, x \rightarrow r \rightarrow n\}$	$\{x \rightarrow r \rightarrow n \rightarrow *\}$
$x \rightarrow n$	n	x	$\{x, x \rightarrow n\}$	$\{x \rightarrow n \rightarrow *\}$
$x \rightarrow r$	r	x	$\{x, x \rightarrow r\}$	$\{x \rightarrow r \rightarrow *\}$
x	x	$\mathcal{E}$	$\{x\}$	$\{x \rightarrow *\}$

Flow Functions for Explicit Liveness Analysis

access expression

corresponding access path

Statement	*ConstKill*	*DepKill*(X)	*ConstGen*	*DepGen*(X)
Use α_y	$\emptyset$	$\emptyset$	*prefixes*(*base*(ρ_y))	$\emptyset$
Use $\alpha_y.d$	$\emptyset$	$\emptyset$	*prefixes*(ρ_y)	$\emptyset$
$\alpha_x = new$	$\{\rho_x \rightarrow *\}$	$\emptyset$	*prefixes*(*base*(ρ_x))	$\emptyset$
$\alpha_x = Null$	$\{\rho_x \rightarrow *\}$	$\emptyset$	*prefixes*(*base*(ρ_x))	$\emptyset$
$\alpha_x = \alpha_y$	$\{\rho_x \rightarrow *\}$	$\emptyset$	*prefixes*(*base*(ρ_x)) $\cup$ *prefixes*(*base*(ρ_y))	$\{\rho_y \rightarrow \sigma \mid \rho_x \rightarrow \sigma \in X\}$
End	$\emptyset$	$\emptyset$	*summary*(*Globals*)	$\emptyset$
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Flow Functions for Explicit Liveness Analysis

Statement	$ConstKill$	$DepKill(X)$	$ConstGen$	$DepGen(X)$
$Use\ \alpha_y$	$\emptyset$	$\emptyset$	$prefixes(base(\rho_y))$	$\emptyset$
$Use\ \alpha_y.d$	$\emptyset$	$\emptyset$	$prefixes(\rho_y)$	$\emptyset$
$\alpha_x = new$	$\{\rho_x \rightarrow *\}$	$\emptyset$	$prefixes(base(\rho_x))$	$\emptyset$
$\alpha_x = Null$	$\{\rho_x \rightarrow *\}$	$\emptyset$	$prefixes(base(\rho_x))$	$\emptyset$
$\alpha_x = \alpha_y$	$\{\rho_x \rightarrow *\}$	$\emptyset$	$prefixes(base(\rho_x)) \cup prefixes(base(\rho_y))$	$\{\rho_y \rightarrow \sigma \mid \rho_x \rightarrow \sigma \in X\}$
End	$\emptyset$	$\emptyset$	$summary(Globals)$	$\emptyset$
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Transfer

Flow Functions for Explicit Liveness Analysis

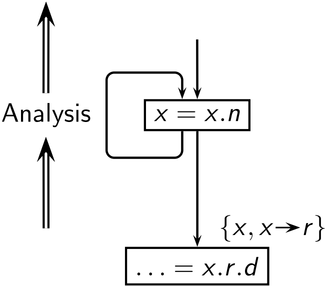
Statement	*ConstKill*	*DepKill*(X)	*ConstGen*	*DepGen*(X)
Use α_y	$\emptyset$	$\emptyset$	$prefixes(base(\rho_y))$	$\emptyset$
Use $\alpha_y.d$	$\emptyset$	$\emptyset$	$prefixes(\rho_y)$	$\emptyset$
$\alpha_x = new$	$\{\rho_x \rightarrow *\}$	$\emptyset$	$prefixes(base(\rho_x))$	$\emptyset$
$\alpha_x = Null$	$\{\rho_x \rightarrow *\}$	$\emptyset$	$prefixes(base(\rho_x))$	$\emptyset$
$\alpha_x = \alpha_y$	$\{\rho_x \rightarrow *\}$	$\emptyset$	$prefixes(base(\rho_x)) \cup prefixes(base(\rho_y))$	$\{\rho_y \rightarrow \sigma \mid \rho_x \rightarrow \sigma \in X\}$
End	$\emptyset$	$\emptyset$	$summary(Globals)$	$\emptyset$
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

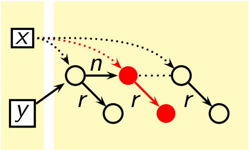
End of procedure

Flow Functions for Handling Procedure Calls in Computing Explicit Liveness

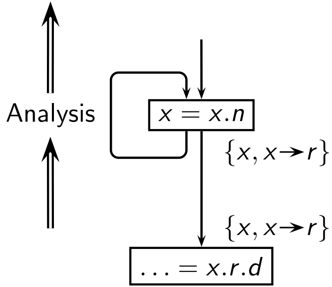
Statement	*ConstKill*	*DepKill*(X)	*ConstGen*	*DepGen*(X)
$\alpha_x = f(\alpha_y)$	$\{\rho_x \rightarrow *\}$	$\emptyset$	$prefixes(base(\rho_x)) \cup$ $prefixes(base(\rho_y)) \cup$ $summary(\{\rho_y\} \cup Globals)$	$\emptyset$
return α_y	$\emptyset$	$\emptyset$	$prefixes(base(\rho_y)) \cup$ $summary(\{\rho_y\})$	$\emptyset$

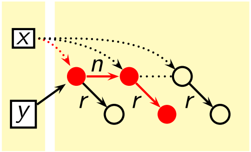
Computing Explicit Liveness Using Sets of Access Paths

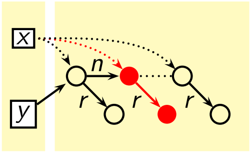




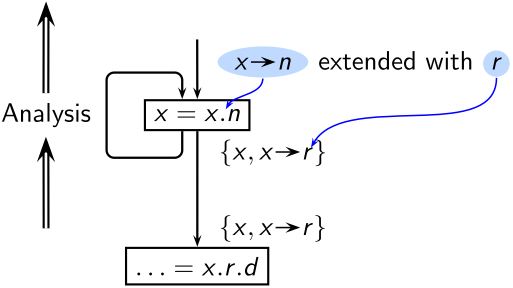
Computing Explicit Liveness Using Sets of Access Paths

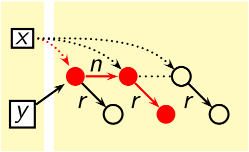


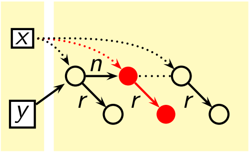




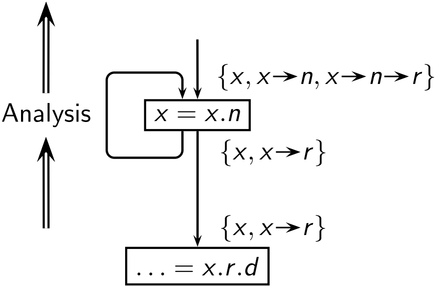
Computing Explicit Liveness Using Sets of Access Paths

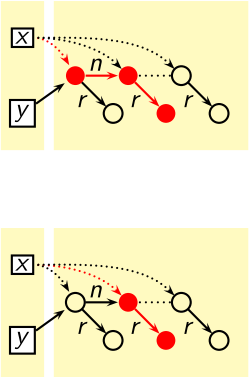




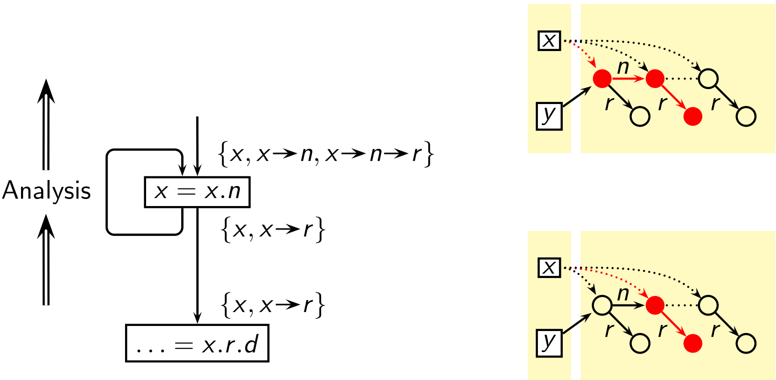


Computing Explicit Liveness Using Sets of Access Paths



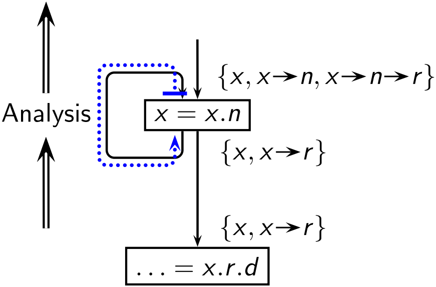


Computing Explicit Liveness Using Sets of Access Paths



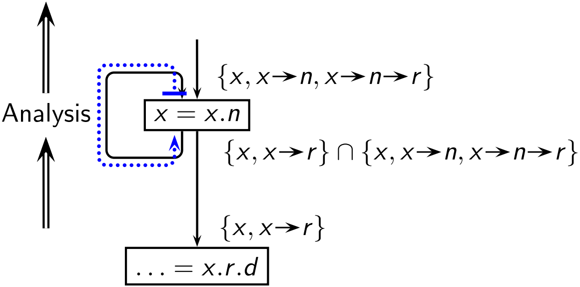
Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An *All Paths* problem



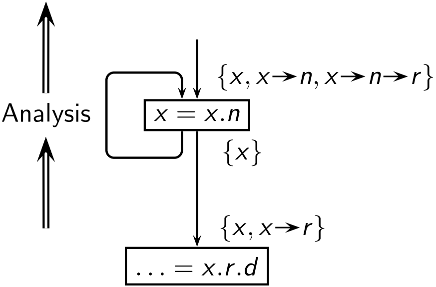
Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An *All Paths* problem



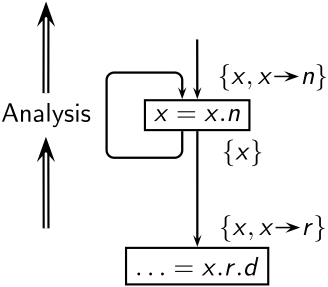
Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An *All Paths* problem



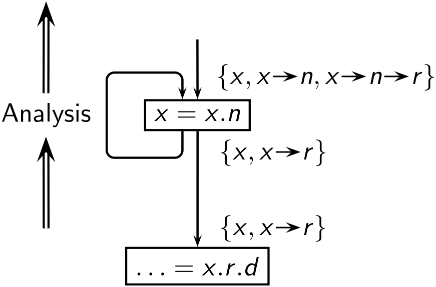
Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An *All Paths* problem



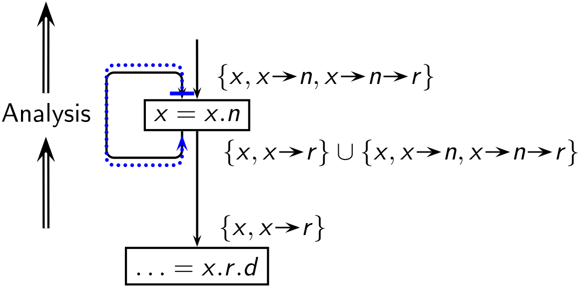
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



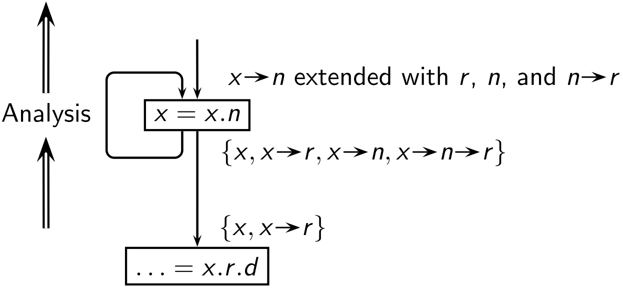
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



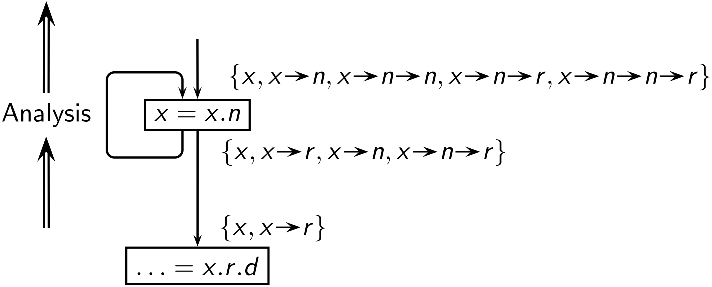
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



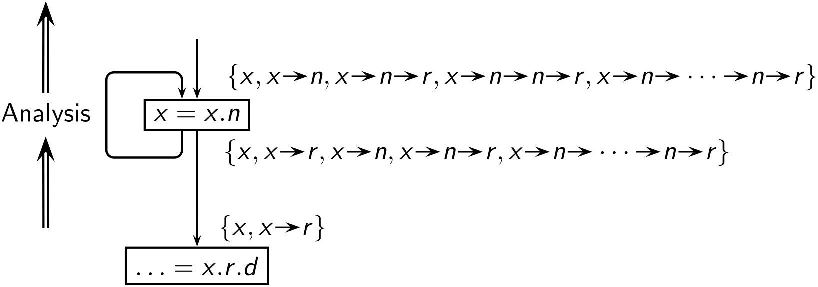
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



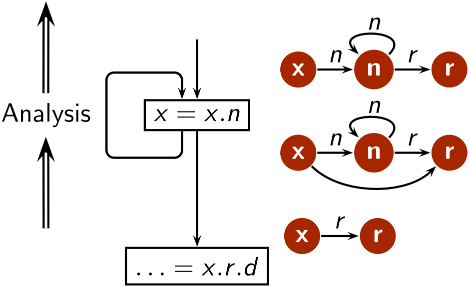
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



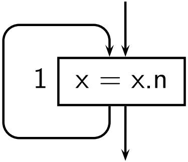
Infinite Number of Unbounded Access Paths

Key Idea #5: Using Graphs as Data Flow Values



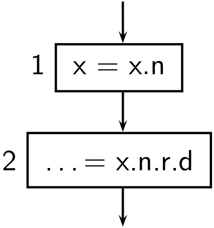
Finite Number of Bounded Structures

Key Idea #6 : Include Program Point in Graphs



$\{x, x \rightarrow n, x \rightarrow n \rightarrow n, x \rightarrow n \rightarrow n \rightarrow n, \ldots\}$

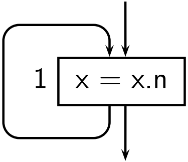
Different occurrences of n's in an access path are
Indistinguishable



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

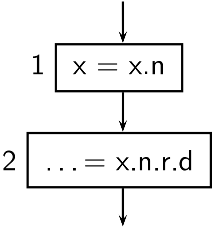
Different occurrences of n's in an access path are
Distinct

Key Idea #6 : Include Program Point in Graphs



$\{x, x{\rightarrow}n, x{\rightarrow}n{\rightarrow}n, x{\rightarrow}n{\rightarrow}n{\rightarrow}n, \ldots\}$

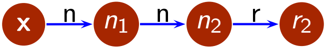
Different occurrences of n's in an access path are **Indistinguishable**



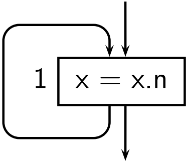
$\{x, x{\rightarrow}n, x{\rightarrow}n{\rightarrow}n, x{\rightarrow}n{\rightarrow}n{\rightarrow}r\}$

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

Access Graph :



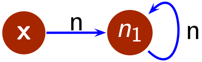
Key Idea #6 : Include Program Point in Graphs

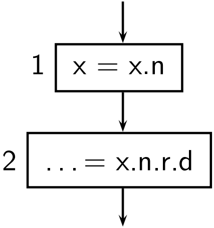


$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow n, \ldots\}$

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

Access Graph :

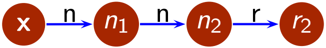




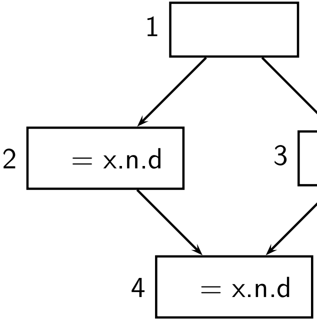
$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow r\}$

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

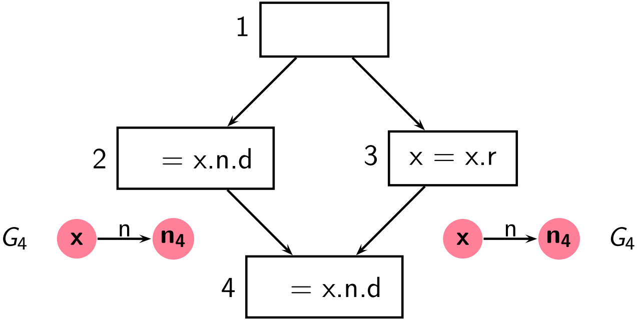
Access Graph :



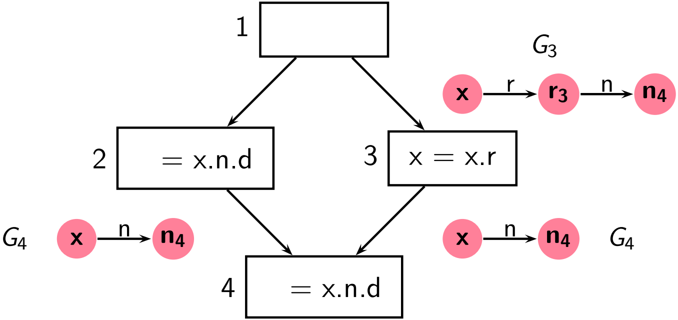
Inclusion of Program Point Facilitates Summarization



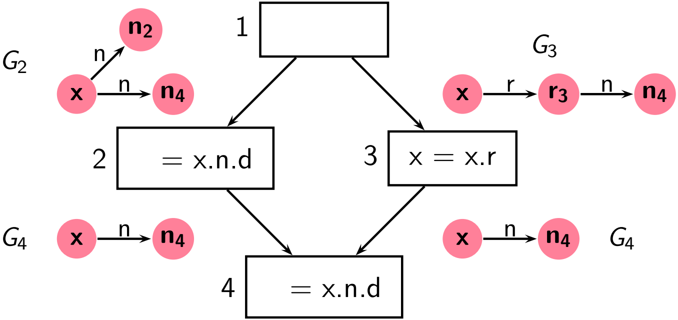
Inclusion of Program Point Facilitates Summarization



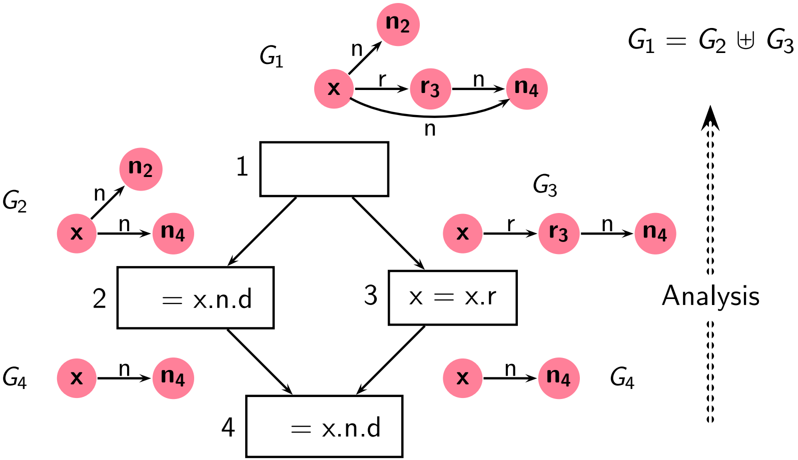
Inclusion of Program Point Facilitates Summarization



Inclusion of Program Point Facilitates Summarization

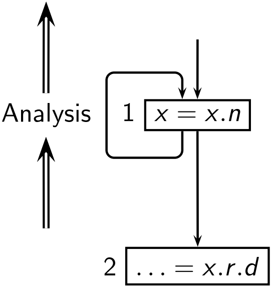


Inclusion of Program Point Facilitates Summarization



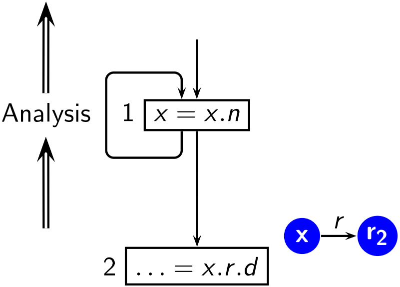
Inclusion of Program Point Facilitates Summarization

Iteration #1

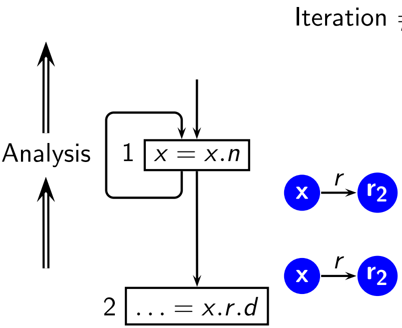


Inclusion of Program Point Facilitates Summarization

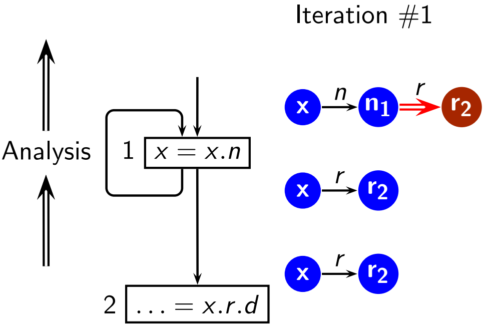
Iteration #1



Inclusion of Program Point Facilitates Summarization

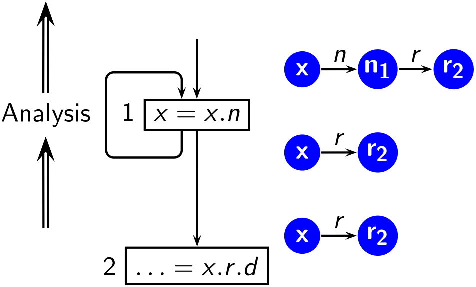


Inclusion of Program Point Facilitates Summarization

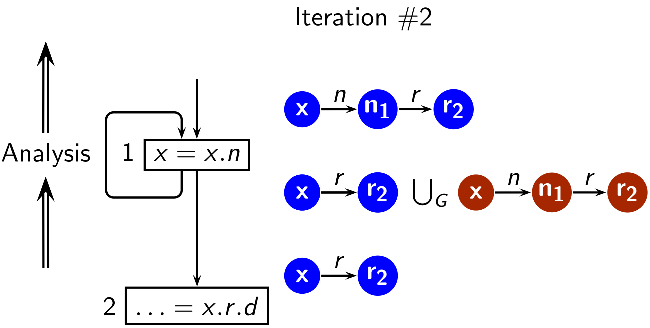


Inclusion of Program Point Facilitates Summarization

Iteration #1

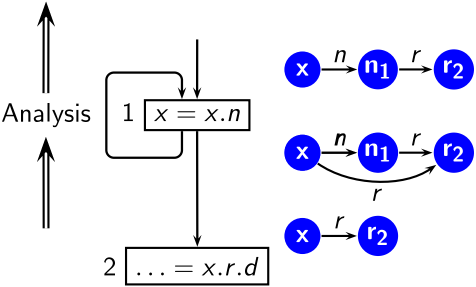


Inclusion of Program Point Facilitates Summarization



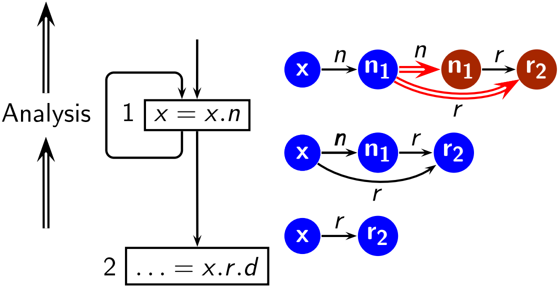
Inclusion of Program Point Facilitates Summarization

Iteration #2

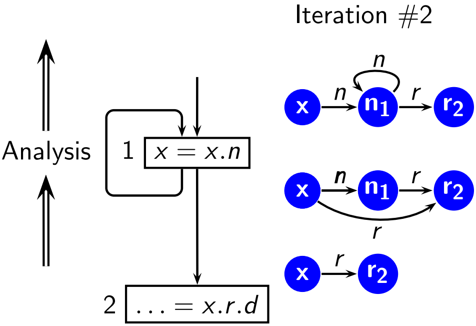


Inclusion of Program Point Facilitates Summarization

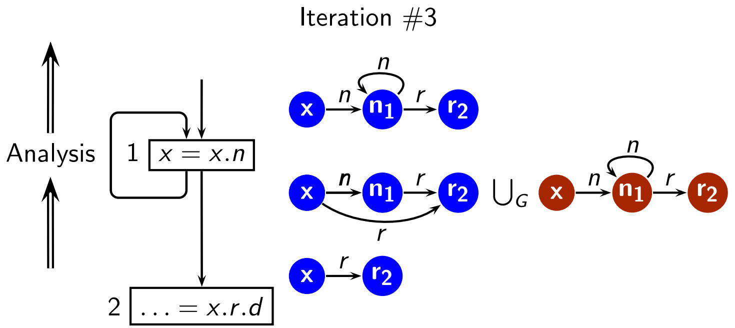
Iteration #2



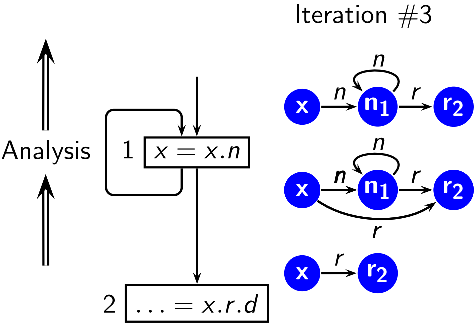
Inclusion of Program Point Facilitates Summarization



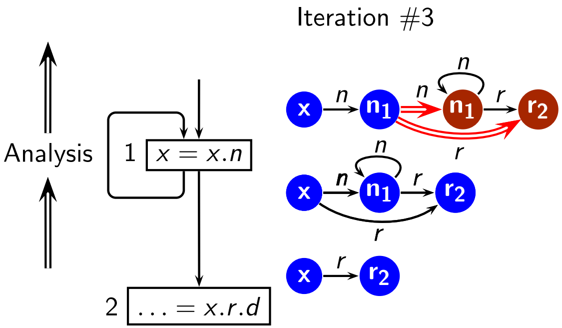
Inclusion of Program Point Facilitates Summarization



Inclusion of Program Point Facilitates Summarization

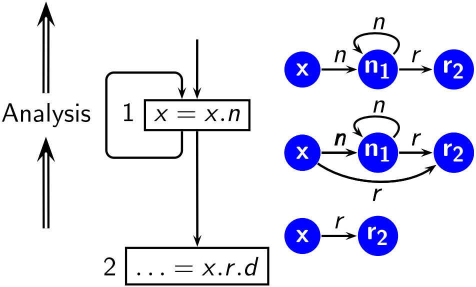


Inclusion of Program Point Facilitates Summarization



Inclusion of Program Point Facilitates Summarization

Iteration #3



Access Graph and Memory Graph

Program Fragment

$$\boxed{x.l = y.r} \quad 1$$

$\downarrow$

$$\boxed{\text{if } (x.l.n == y.r.n)} \quad 2$$

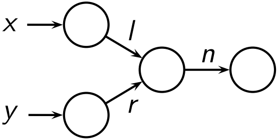
Access Graph and Memory Graph

Program Fragment

$\boxed{x.l = y.r}$ 1

$\boxed{\text{if } (x.l.n == y.r.n)}$ 2

Memory Graph



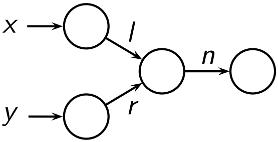
Access Graph and Memory Graph

Program Fragment

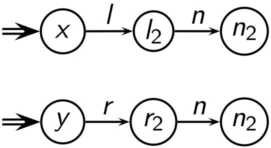
$\boxed{x.l = y.r}$ 1

$\boxed{\text{if } (x.l.n == y.r.n)}$ 2

Memory Graph

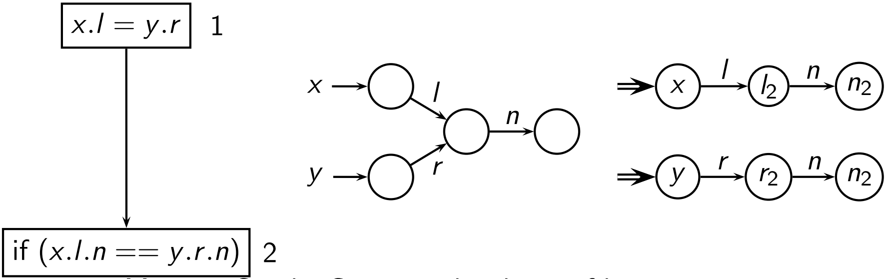


Access Graphs



Access Graph and Memory Graph

Program Fragment | Memory Graph | Access Graphs



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

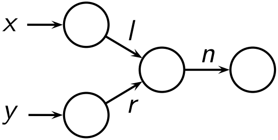
Access Graph and Memory Graph

Program Fragment

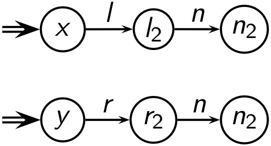
$\boxed{x.l = y.r}$ 1

$\boxed{\text{if } (x.l.n == y.r.n)}$ 2

Memory Graph



Access Graphs



- 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
- $\uplus$ 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*. $G \uplus G'$
- *Path Removal*.
  $G \ominus \rho$ removes those access paths in G which have ρ 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.

Non-Examinable

Operation		Access Paths
Union	$G_3 = G_1 \uplus G_2$	$P(G_3, M_3) \supseteq P(G_1, M_1) \cup P(G_2, M_2)$
Path Removal	$G_2 = G_1 \ominus \rho$	$P(G_2, M_2) \supseteq P(G_1, M_1) - \{\rho \rightarrow \sigma \mid \rho \rightarrow \sigma \in P(G_1, M_1)\}$
Factorization	$S = G_1/(G_2, M)$	$P(S, M_s) = \{\sigma \mid \rho' \rightarrow \sigma \in P(G_1, M_1),\ \rho' \in P(G_2, M)\}$
Extension	$G_2 = (G_1, M) \# \emptyset$	$P(G_2, M_2) = \emptyset$
	$G_2 = (G_1, M) \# S$	$P(G_2, M_2) \supseteq P(G_1, M_1) \cup \{\rho \rightarrow \sigma \mid \rho \in P(G_1, M),\ \sigma \in P(S, M_s)\}$

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.

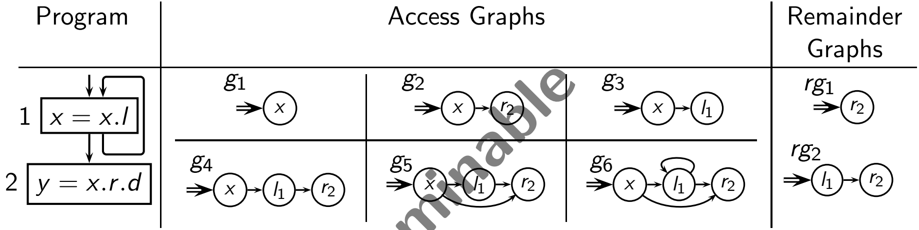
Operation		Access Paths
Union	$G_3 = G_1 \uplus G_2$	$P(G_3, M_3) \supseteq P(G_1, M_1) \cup P(G_2, M_2)$
Path Removal	$G_2 = G_1 \ominus \rho$	$P(G_2, M_2) \supseteq P(G_1, M_1) - \{\rho \rightarrow \sigma \mid \rho \rightarrow \sigma \in P(G_1, M_1)\}$
Factorization	$S = G_1/(G_2, M)$	$P(S, M_s) = \{\sigma \mid \rho' \rightarrow \sigma \in P(G_1, M_1),\ \rho' \in P(G_2, M)\}$
Extension	$G_2 = (G_1, M) \# \emptyset$	$P(G_2, M_2) = \emptyset$
	$G_2 = (G_1, M) \# S$	$P(G_2, M_2) \supseteq P(G_1, M_1) \cup \{\rho \rightarrow \sigma \mid \rho \in P(G_1, M),\ \sigma \in P(S, M_s)\}$

σ represents remainder

ρ' represents quotient

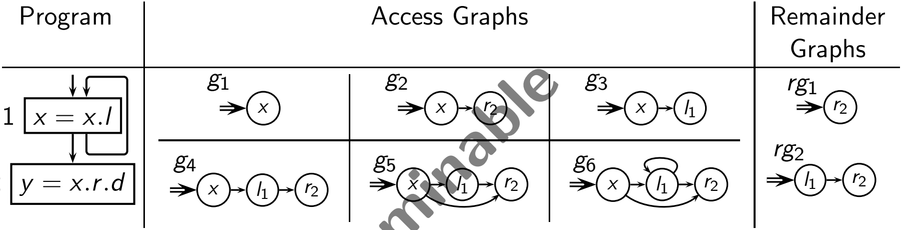
Non-Examinable

Access Graph Operations: Examples



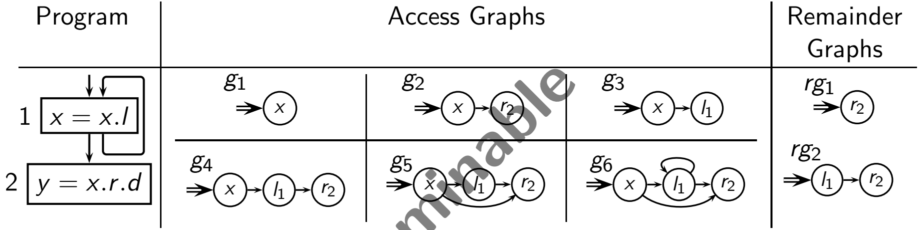
Union	Path Removal	Factorisation	Extension

Access Graph Operations: Examples



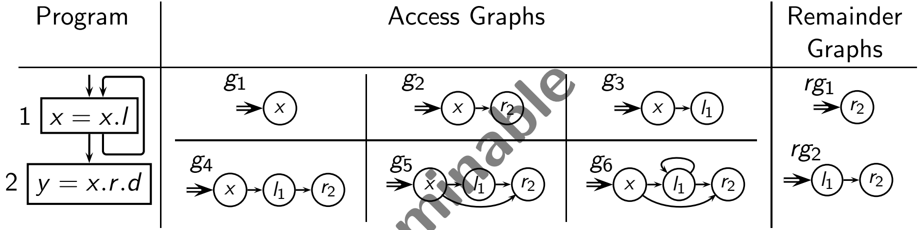
Union	Path Removal	Factorisation	Extension
$g_3 \uplus g_4 = g_4$ $g_2 \uplus g_4 = g_5$ $g_5 \uplus g_4 = g_5$ $g_5 \uplus g_6 = g_6$			

Access Graph Operations: Examples



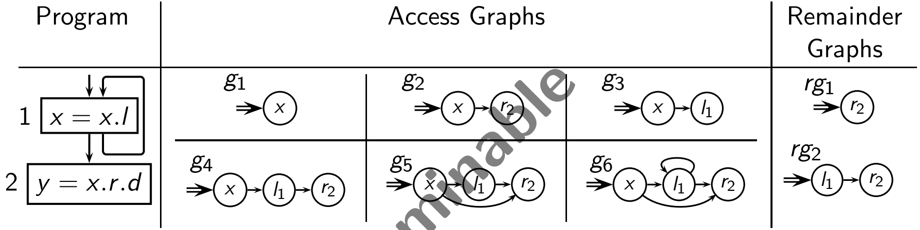
Union	Path Removal	Factorisation	Extension
$g_3 \uplus g_4 = g_4$	$g_6 \ominus x \rightarrow l = g_2$		
$g_2 \uplus g_4 = g_5$	$g_5 \ominus x = \mathcal{E}_G$		
$g_5 \uplus g_4 = g_5$	$g_4 \ominus x \rightarrow r = g_4$		
$g_5 \uplus g_6 = g_6$	$g_4 \ominus x \rightarrow l = g_1$		

Access Graph Operations: Examples



Union	Path Removal	Factorisation	Extension
$g_3 \uplus g_4 = g_4$	$g_6 \ominus x \rightarrow l = g_2$	$g_2 / (g_1, \{x\}) = \{rg_1\}$	
$g_2 \uplus g_4 = g_5$	$g_5 \ominus x = \mathcal{E}_G$	$g_5 / (g_1, \{x\}) = \{rg_1, rg_2\}$	
$g_5 \uplus g_4 = g_5$	$g_4 \ominus x \rightarrow r = g_4$	$g_5 / (g_2, \{r_2\}) = \{\epsilon_{RG}\}$	
$g_5 \uplus g_6 = g_6$	$g_4 \ominus x \rightarrow l = g_1$	$g_4 / (g_2, \{r_2\}) = \emptyset$	

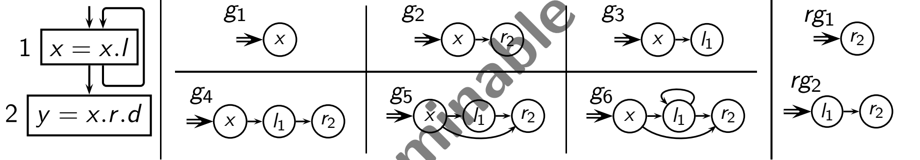
Access Graph Operations: Examples



Union	Path Removal	Factorisation	Extension
$g_3 \uplus g_4 = g_4$	$g_6 \ominus x \rightarrow l = g_2$	$g_2 / (g_1, \{x\}) = \{rg_1\}$	$(g_3, \{l_1\}) \# \{rg_1\} = g_4$
$g_2 \uplus g_4 = g_5$	$g_5 \ominus x = \mathcal{E}_G$	$g_5 / (g_1, \{x\}) = \{rg_1, rg_2\}$	$(g_3, \{x, l_1\}) \# \{rg_1, rg_2\} = g_6$
$g_5 \uplus g_4 = g_5$	$g_4 \ominus x \rightarrow r = g_4$	$g_5 / (g_2, \{r_2\}) = \{\epsilon_{RG}\}$	$(g_2, \{r_2\}) \# \{\epsilon_{RG}\} = g_2$
$g_5 \uplus g_6 = g_6$	$g_4 \ominus x \rightarrow l = g_1$	$g_4 / (g_2, \{r_2\}) = \emptyset$	$(g_2, \{r_2\}) \# \emptyset = \mathcal{E}_G$

Access Graph Operations: Examples

Program	Access Graphs	Remainder Graphs



Non-Examinable

Union	Path Removal	Factorisation	Extension
$g_3 \uplus g_4 = g_4$	$g_6 \ominus x \rightarrow l = g_2$	$g_2 / (g_1, \{x\}) = \{rg_1\}$	$(g_3, \{l_1\}) \# \{rg_1\} = g_4$
$g_2 \uplus g_4 = g_5$	$g_5 \ominus x = \mathcal{E}_G$	$g_5 / (g_1, \{x\}) = \{rg_1, rg_2\}$	$(g_3, \{x, l_1\}) \# \{rg_1, rg_2\} = g_6$
$g_5 \uplus g_4 = g_5$	$g_4 \ominus x \rightarrow r = g_4$	$g_5 / (g_2, \{r_2\}) = \{\epsilon_{RG}\}$	$(g_2, \{r_2\}) \# \{\epsilon_{RG}\} = g_2$
$g_5 \uplus g_6 = g_6$	$g_4 \ominus x \rightarrow l = g_1$	$g_4 / (g_2, \{r_2\}) = \emptyset$	$(g_2, \{r_2\}) \# \emptyset = \mathcal{E}_G$

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.

$$
\begin{aligned}
ELIn_n(v) &= (ELOut_n(v) \ominus ELKillPath_n(v)) \uplus ELGen_n(v) \\
ELOut_n(v) &= \begin{cases} makeGraph(v \rightarrow *) & n = End,\ v \in Globals \\ \mathcal{E}_G & n = End,\ v \notin Globals \\ \biguplus\limits_{s \in succ(n)} ELIn_s(v) & \text{otherwise} \end{cases} \\
ELGen_n(v) &= ELConstGen_n(v) \uplus ELDepGen_n(v)
\end{aligned}
$$

(Note: This notation is slightly different from the notation in the book.)

Flow Functions for Explicit Liveness Analysis

Access Paths $\Longrightarrow$

	Use $\alpha_x.d$	$\alpha_x = \alpha_y$
$ConstKill_n$	$\emptyset$	$\{\rho_x \rightarrow *\}$
$ConstGen_n$	$prefixes(\rho_x)$	$prefixes(base(\rho_x)) \cup prefixes(base(\rho_y))$
$DepGen_n(X)$	$\emptyset$	$\{\rho_y \rightarrow \sigma \mid \rho_x \rightarrow \sigma \in X\}$

Access Graphs $\Downarrow$

$$G_x = makeGraph(\rho_x) \quad G_x^B = makeGraph(base(\rho_x))$$
$$G_y = makeGraph(\rho_y) \quad G_y^B = makeGraph(base(\rho_y))$$

	Use $\alpha_x.d$	Use α_x	$\alpha_x = \alpha_y$	$\alpha_x = Null$, $\alpha_x = New$
$ELKillPath_n(x)$	$\mathcal{E}$	$\mathcal{E}$	ρ_x	ρ_x
$ELKillPath_n(y)$	$\mathcal{E}$	$\mathcal{E}$	$\mathcal{E}$	$\mathcal{E}$
$ELConstGen_n(x)$	G_x	G_x^B	G_x^B	G_x^B
$ELConstGen_n(y)$	$\mathcal{E}_G$	$\mathcal{E}_G$	G_y^B	$\mathcal{E}_G$
$ELDepGen_n(x)(X)$	$\mathcal{E}_G$	$\mathcal{E}_G$	$\mathcal{E}_G$	$\mathcal{E}_G$
$ELDepGen_n(y)(X)$	$\mathcal{E}_G$	$\mathcal{E}_G$	$(G_y, M_y) \# (X/(G_x, M_x))$	$\mathcal{E}_G$

Flow Functions for Explicit Liveness Analysis

The singleton set containing the last node corresponding to ρ_x

Access Paths $\Longrightarrow$

$ConstKill_n$		
$ConstGen_n$	$prefixes(\rho_x)$	$prefixes(base(\rho_x)) \cup prefixes(base(\rho_y))$
$DepGen_n(X)$	$\emptyset$	$\{\rho_y \rightarrow \sigma \mid \rho_x \rightarrow \sigma \in X\}$

Access Graphs $\Downarrow$

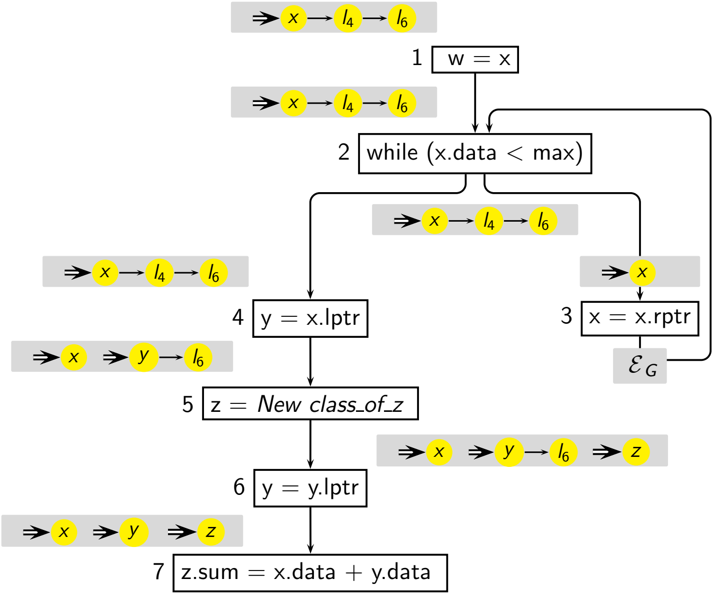
$G_x = makeGraph(\rho_x) \quad G_x^B = makeGraph(base(\rho_x))$

$\ldots(\rho_y) \quad G_y^B = makeGraph(base(\rho_y))$

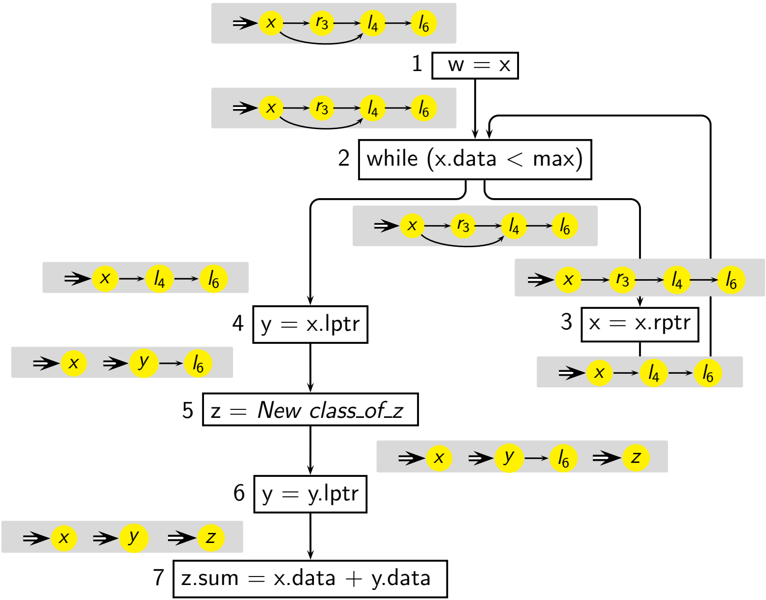
The singleton set containing the last node corresponding to ρ_y

			$\alpha_x = \alpha_y$	$\alpha_x = Null$, $\alpha_x = New$
$ELKillPath_n(x)$	$\mathcal{E}$	$\mathcal{E}$	ρ_x	ρ_x
$ELKillPath_n(y)$	$\mathcal{E}$	$\mathcal{E}$	$\mathcal{E}$	$\mathcal{E}$
$ELConstGen_n(x)$	G_x	G_x^B	G_x^B	G_x^B
$ELConstGen_n(y)$	$\mathcal{E}_G$	$\mathcal{E}_G$	G_y^B	$\mathcal{E}_G$
$ELDepGen_n(x)(X)$	$\mathcal{E}_G$	$\mathcal{E}_G$	$\mathcal{E}_G$	$\mathcal{E}_G$
$ELDepGen_n(y)(X)$	$\mathcal{E}_G$	$\mathcal{E}_G$	$(G_y, M_y)\#(X/(G_x, M_x))$	$\mathcal{E}_G$

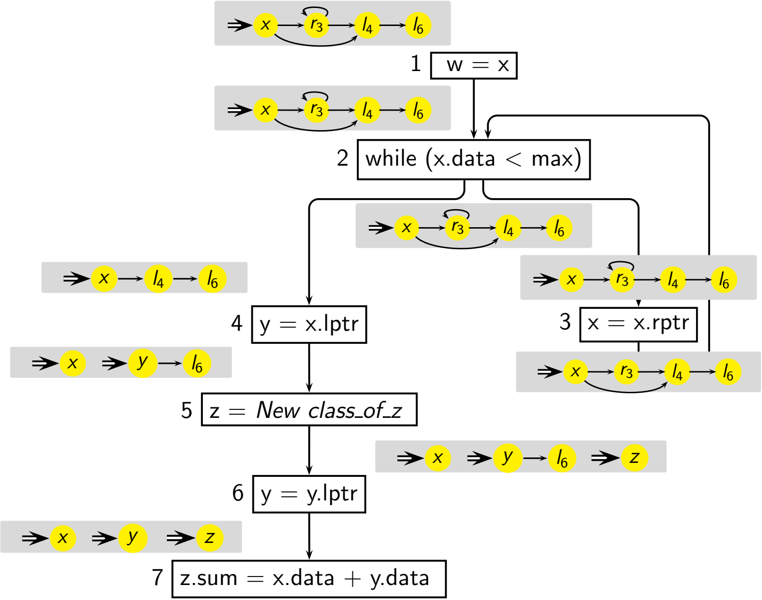
Liveness Analysis of Example Program: Ist Iteration



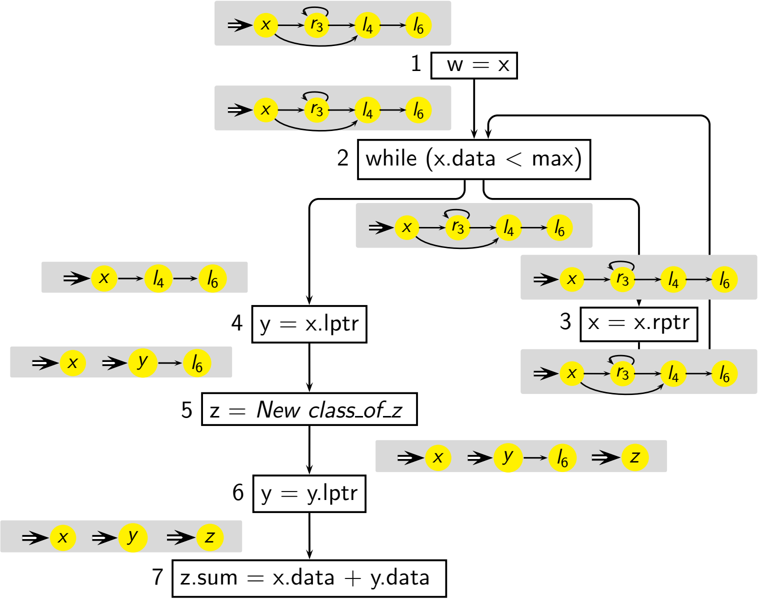
Liveness Analysis of Example Program: 2nd Iteration



Liveness Analysis of Example Program: 3rd Iteration



Liveness Analysis of Example Program: 4th Iteration



Which Access Paths Can be Nullified?

- Consider extensions of accessible paths for nullification.

Let ρ be accessible at p (i.e. available or anticipable)
for each reference field f of the object pointed to by ρ
  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, ρ is empty and f is the root variable name.

Which Access Paths Can be Nullified?

Can be safely dereferenced

- Consider extensions of accessible paths for nullification.

> Let ρ be accessible at p (i.e. available or anticipable)
> **for** each reference field f of the object pointed to by ρ
>   **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, ρ is empty and f is the root variable name.

Which Access Paths Can be Nullified?

Can be safely dereferenced

Consider link aliases at p

- Consider extensions of accessible paths for nullification.

Let ρ be accessible at p (i.e. available or anticipable)
for each reference field f of the object pointed to by ρ
    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, ρ is empty and f is the root variable name.

Which Access Paths Can be Nullified?

Can be safely dereferenced

Consider link aliases at p

- Consider extensions of accessible paths for nullification.

> Let ρ be accessible at p (i.e. available or anticipable)
> **for** each reference field f of the object pointed to by ρ
>   **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, ρ is empty and f is the root variable name.

Cannot be hoisted and is not redefined at p

Availability and Anticipability Analyses

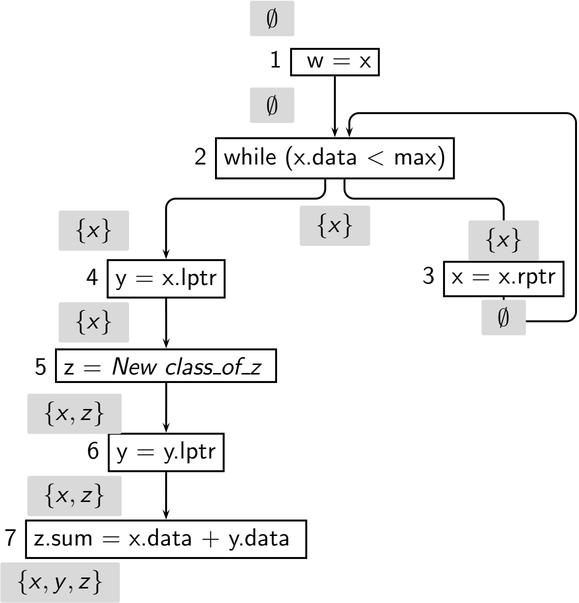
- ρ is available at program point p if the target of each prefix of ρ is guaranteed to be created along every control flow path reaching p.
- ρ is anticipable at program point p if the target of each prefix of ρ is guaranteed to be dereferenced along every control flow path starting at p.

Availability and Anticipability Analyses

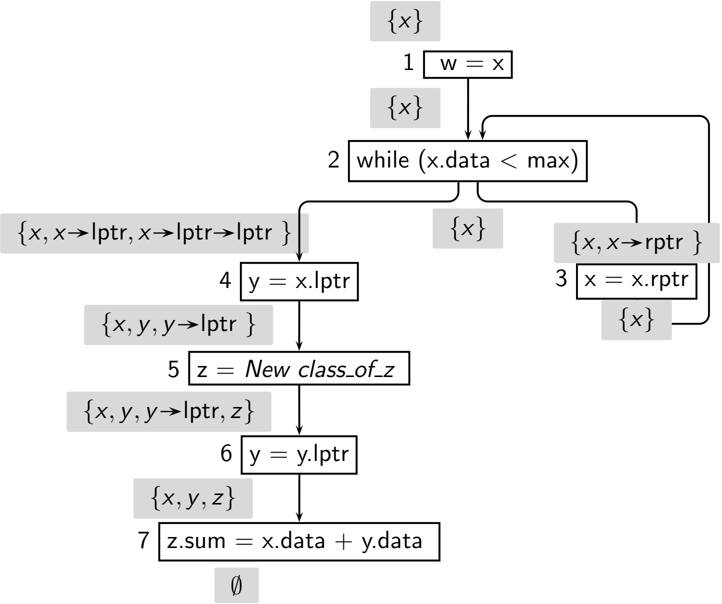
- ρ is available at program point p if the target of each prefix of ρ is guaranteed to be created along every control flow path reaching p.
- ρ is anticipable at program point p if the target of each prefix of ρ is guaranteed to be dereferenced along every control flow path starting at p.
- Finiteness.
  - An anticipable (available) access path must be anticipable (available) along every paths. 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.

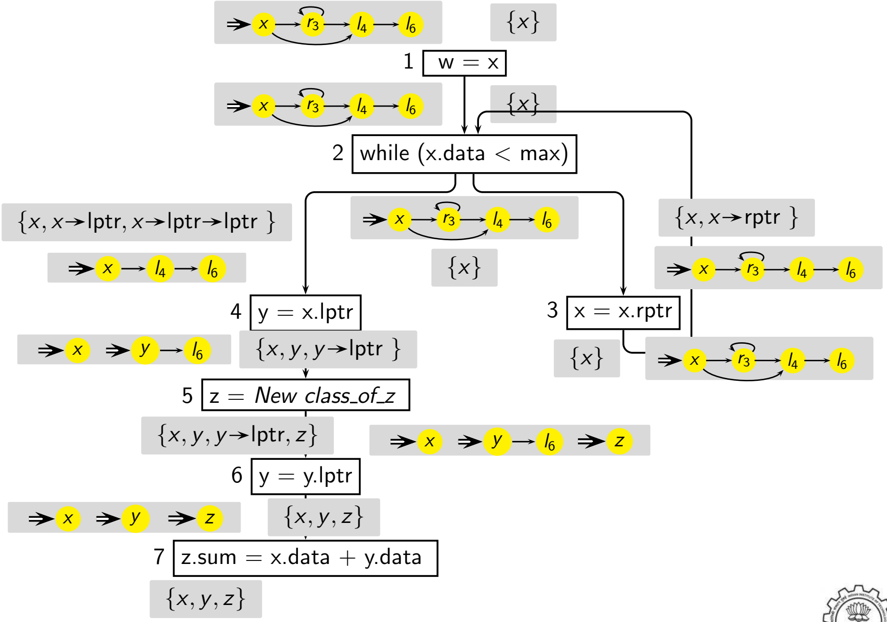
Availability Analysis of Example Program



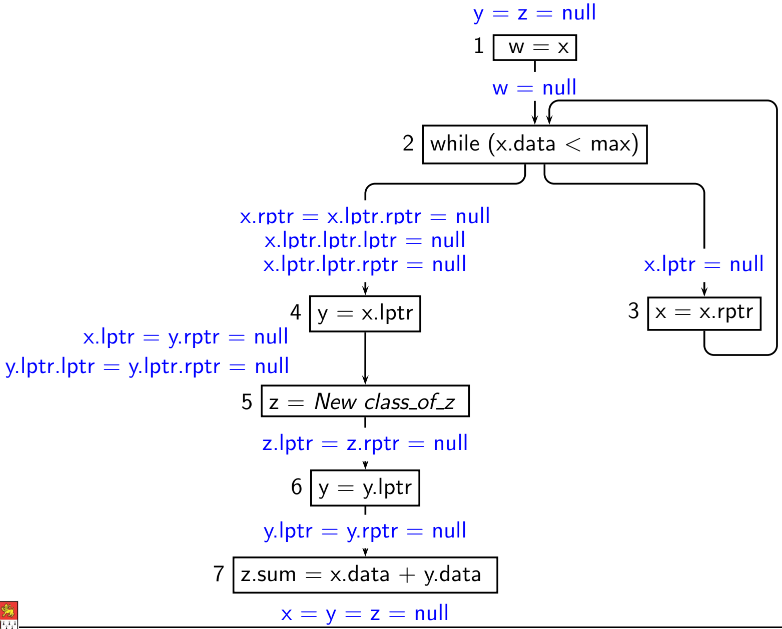
Anticipability Analysis of Example Program



Live and Accessible Paths



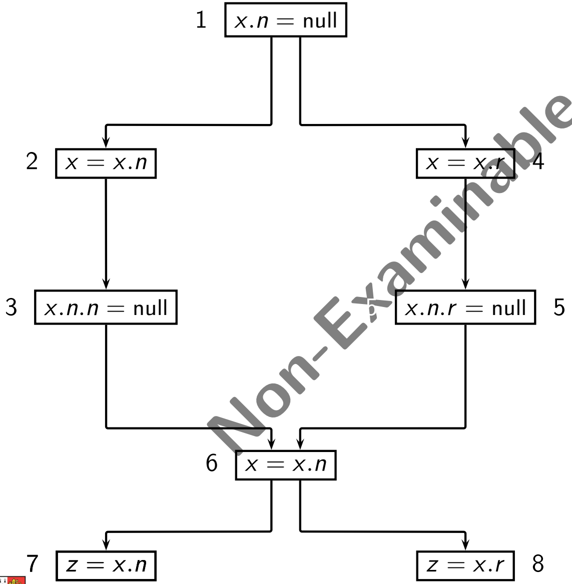
Creating null Assignments from Live and Accessible Paths



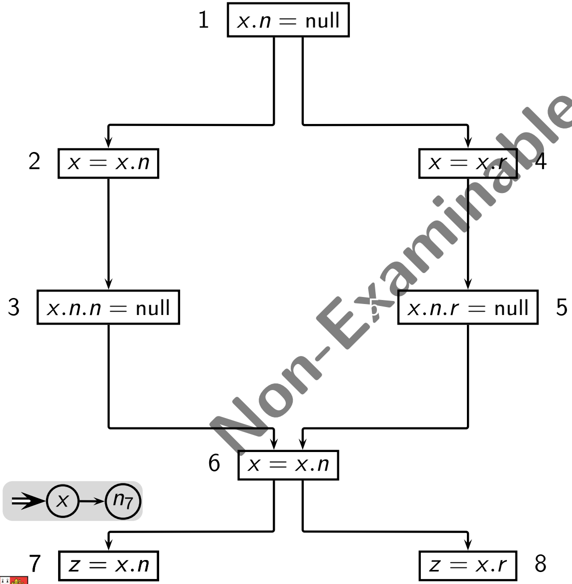
The Resulting Program

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

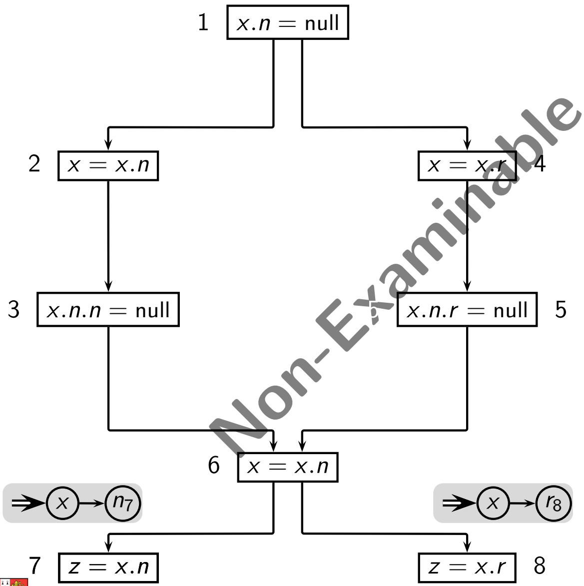
Non-Distributivity of Explicit Liveness Analysis



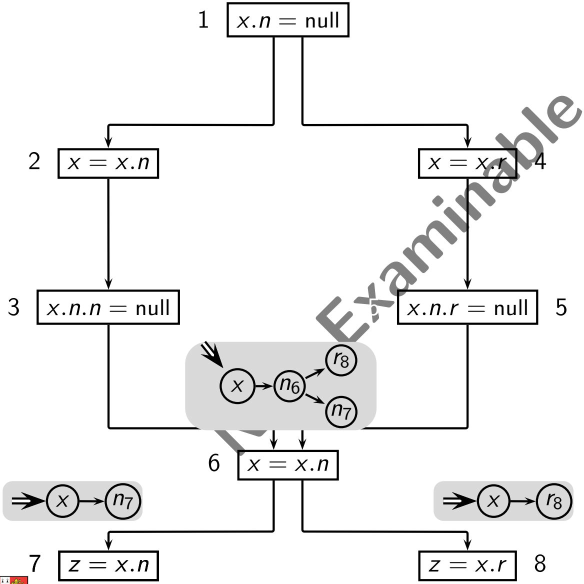
Non-Distributivity of Explicit Liveness Analysis



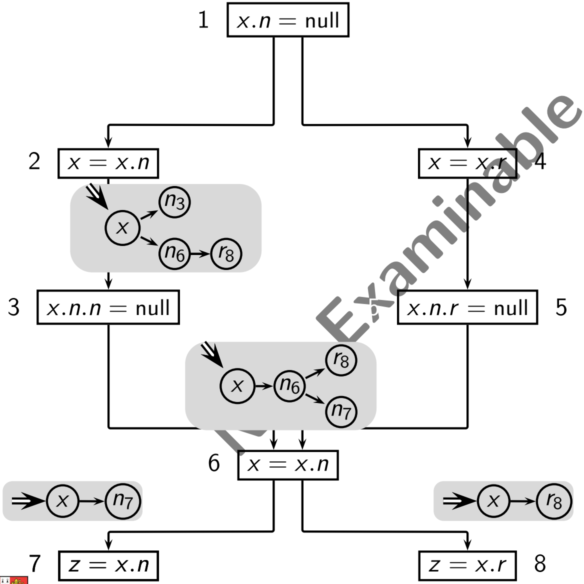
Non-Distributivity of Explicit Liveness Analysis



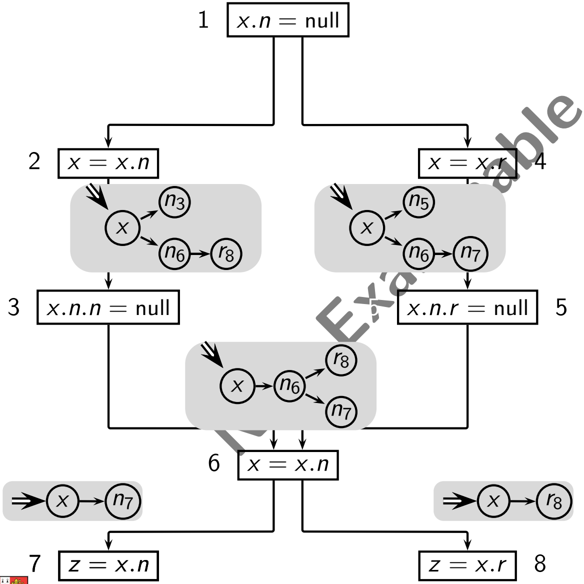
Non-Distributivity of Explicit Liveness Analysis



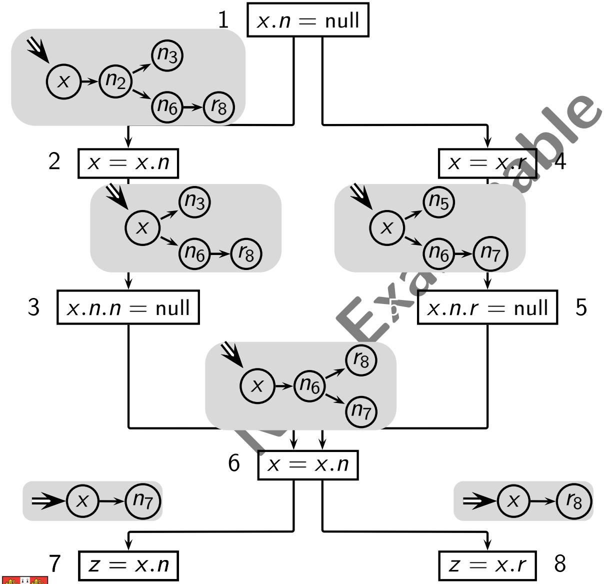
Non-Distributivity of Explicit Liveness Analysis



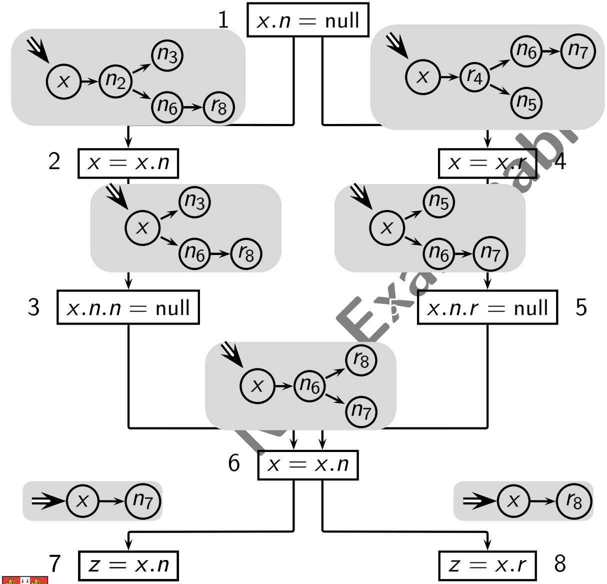
Non-Distributivity of Explicit Liveness Analysis



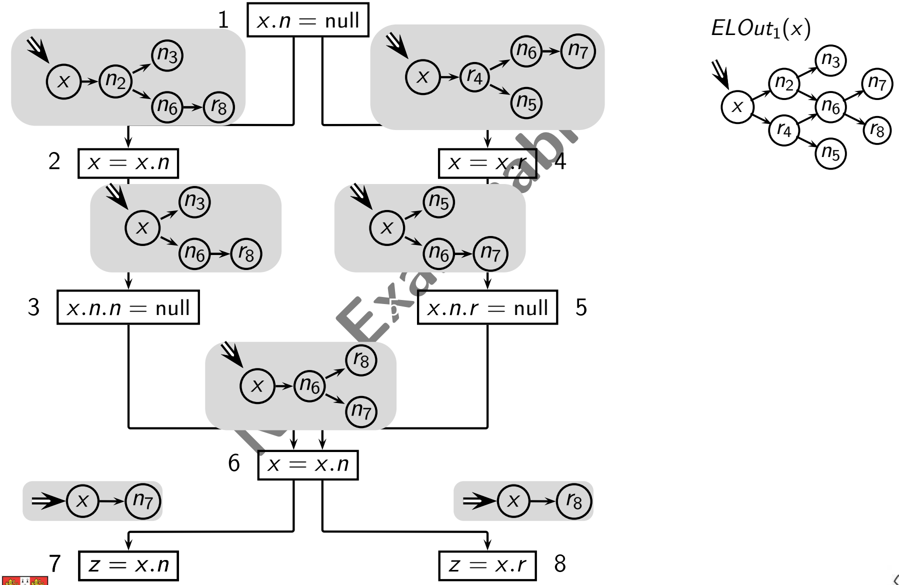
Non-Distributivity of Explicit Liveness Analysis



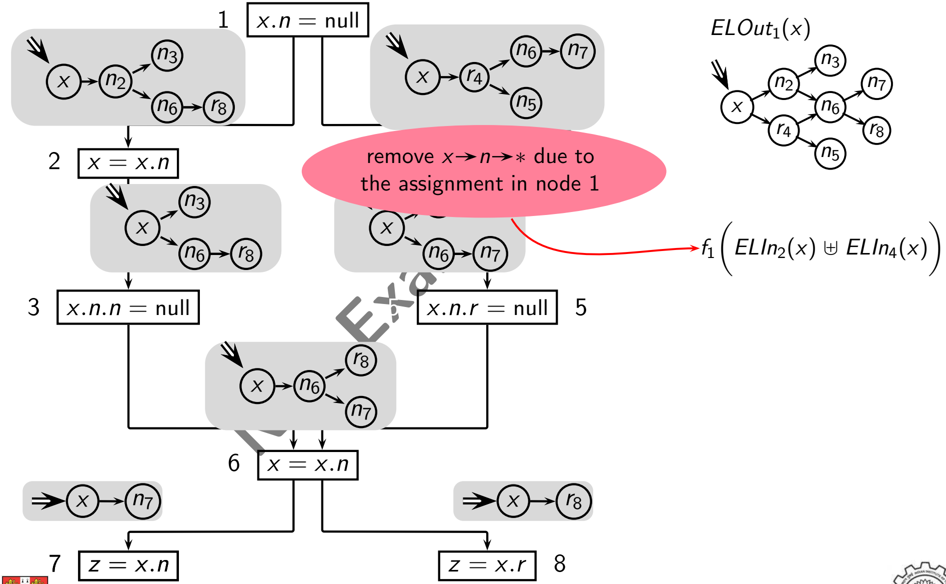
Non-Distributivity of Explicit Liveness Analysis



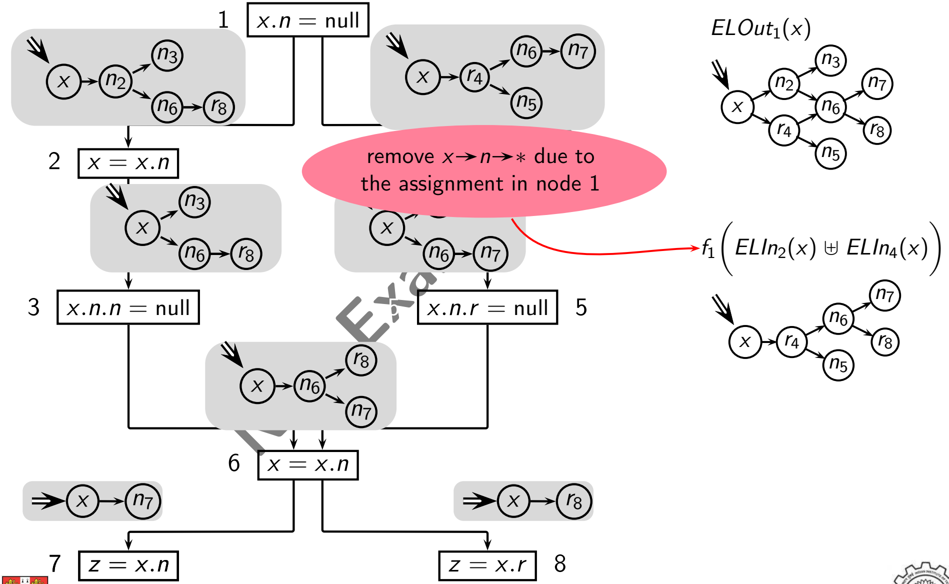
Non-Distributivity of Explicit Liveness Analysis



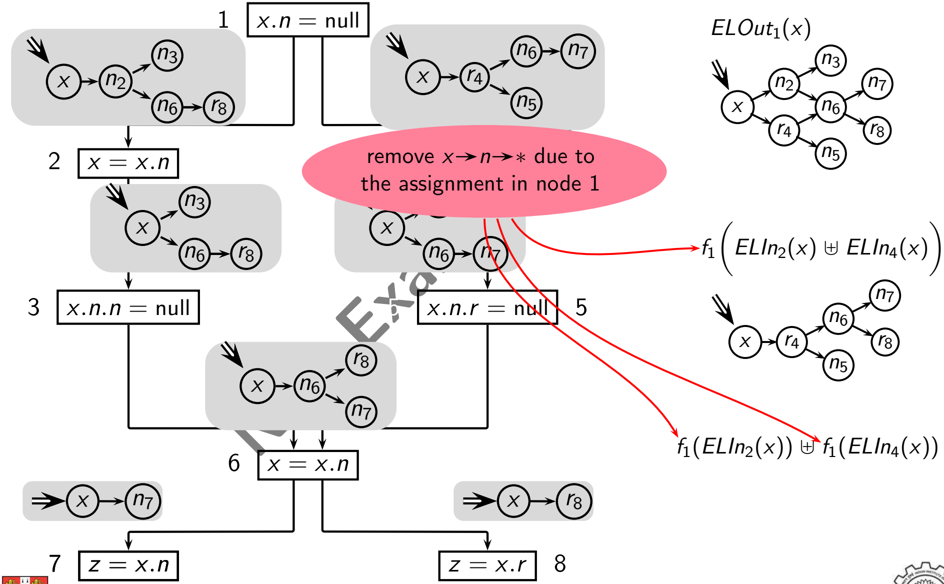
Non-Distributivity of Explicit Liveness Analysis



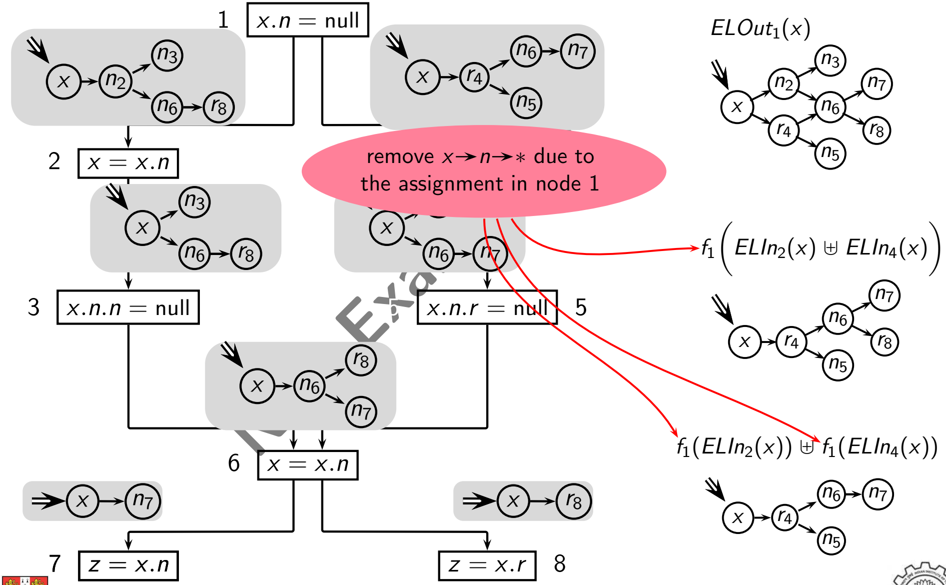
Non-Distributivity of Explicit Liveness Analysis



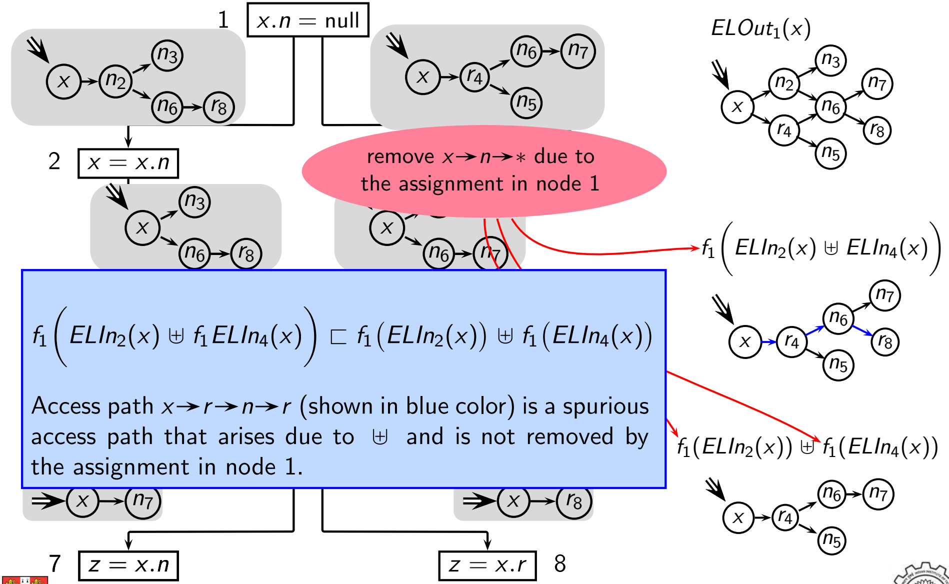
Non-Distributivity of Explicit Liveness Analysis



Non-Distributivity of Explicit Liveness Analysis



Non-Distributivity of Explicit Liveness Analysis



$$f_1\left(ELIn_2(x) \uplus f_1 ELIn_4(x)\right) \sqsubset f_1\left(ELIn_2(x)\right) \uplus f_1\left(ELIn_4(x)\right)$$

Access path $x \rightarrow r \rightarrow n \rightarrow r$ (shown in blue color) is a spurious access path that arises due to $\uplus$ 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

    $\Rightarrow$ 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

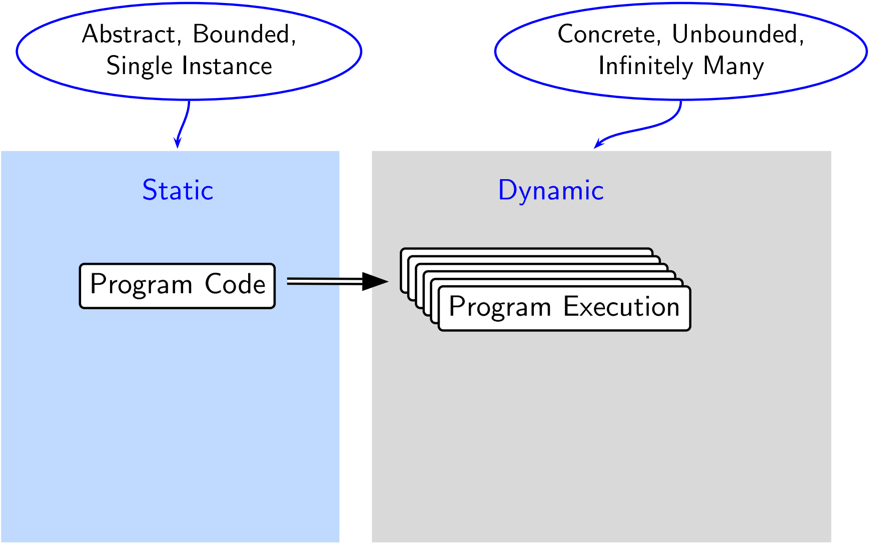
    Heap manipulations consist of repeating patterns which bear a close resemblance to program structure

    Analysis of heap data is possible despite the fact that the mappings between access expressions and l-values keep changing

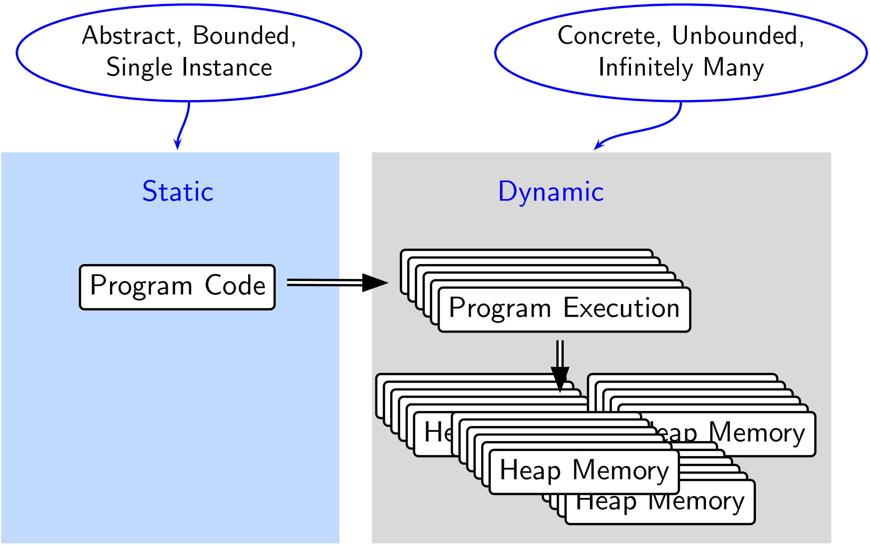
BTW, What is Static Analysis of Heap?

Static	Dynamic

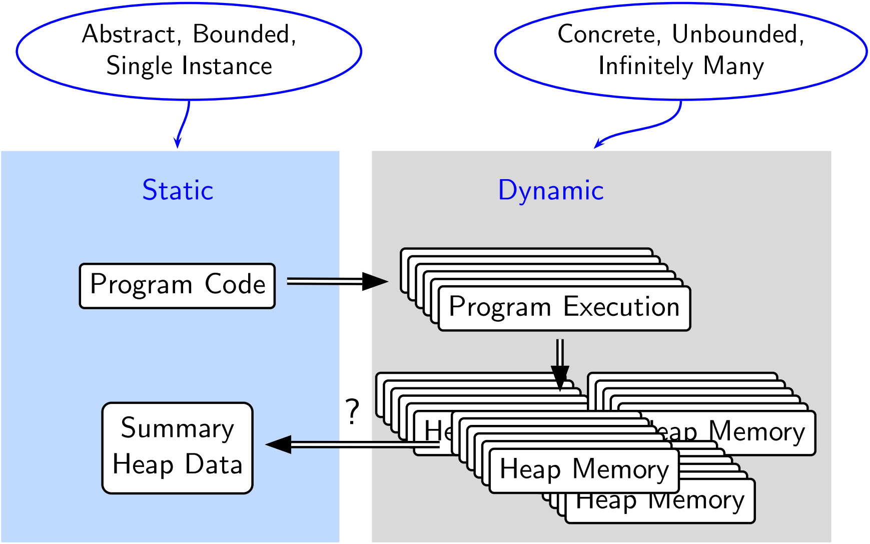
BTW, What is Static Analysis of Heap?



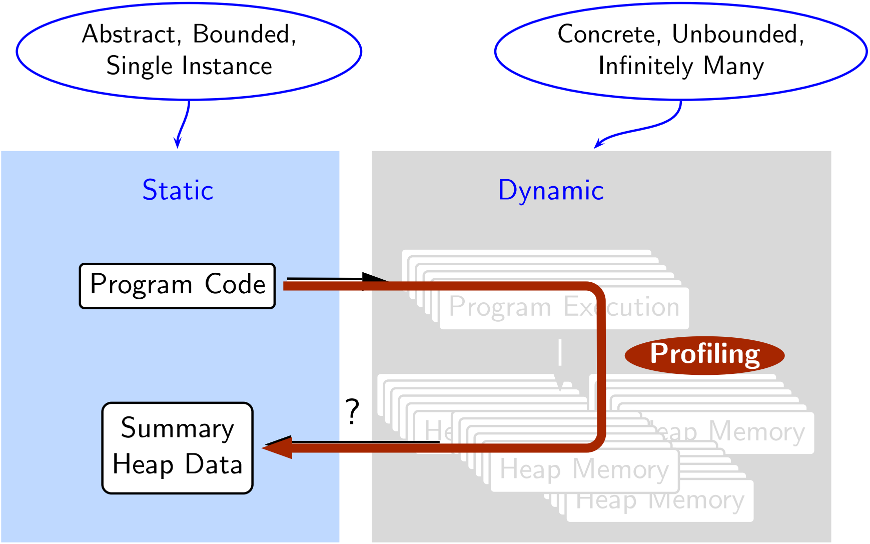
BTW, What is Static Analysis of Heap?



BTW, What is Static Analysis of Heap?



BTW, What is Static Analysis of Heap?



BTW, What is Static Analysis of Heap?