General Data Flow Framekworks

Uday Khedker

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

September 2010

Copyright

Part 1

About These Slides

Outline

Part 2

Modelling General Flows

Modelling Flow Functions for General Flows

- General flow functions can be written as

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

where Gen and Kill have constant and dependent parts

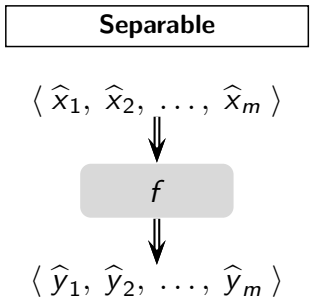
$$\begin{aligned} \text{Gen}_n(X) &= \mathit{ConstGen}_n \cup \mathit{DepGen}_n(X) \\ \text{Kill}_n(X) &= \mathit{ConstKill}_n \cup \mathit{DepKill}_n(X) \end{aligned}$$

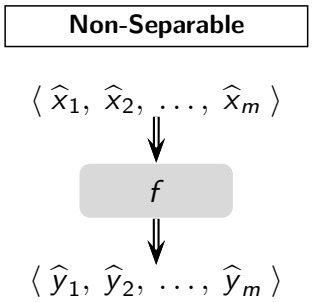
- The dependent parts take care of
  - dependence across different entities as well as
  - dependence on the value of the same entity in the argument X
- Bit vector frameworks are a special case

$$\mathit{DepGen}_n(X) = \mathit{DepKill}_n(X) = \emptyset$$

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 $\widehat{x}_i$



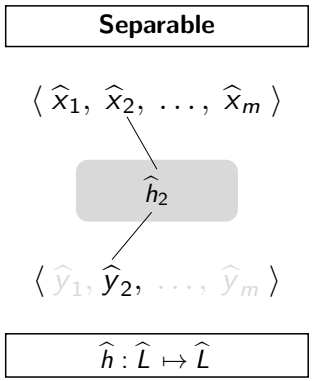


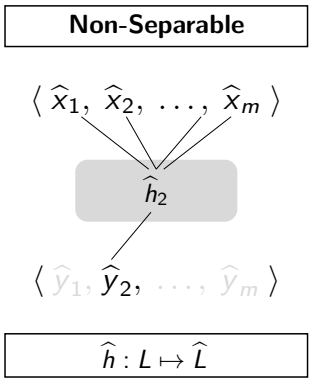
Example: All bit vector frameworks

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 $\widehat{x}_i$





Example: All bit vector frameworks

Example: Constant Propagation

Larger Values of Loop Closure Bounds

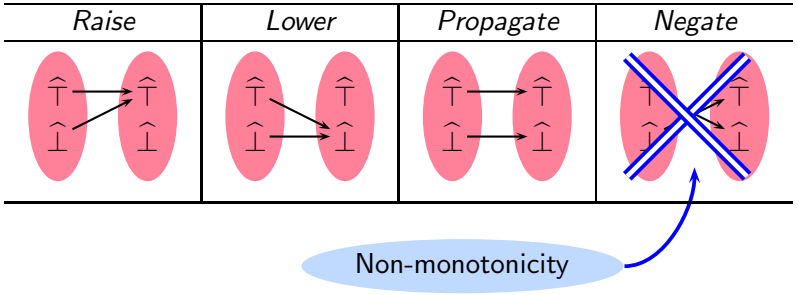
- Fast Frameworks $\equiv$ 2-bounded frameworks (eg. bit vector frameworks)
  Both these conditions must be satisfied
  - *Separability*
    Data flow values of different entities are independent
  - *Constant or Identity Flow Functions*
    Flow functions for an entity are either constant or identity
- Non-fast frameworks
  At least one of the above conditions is violated

Part 3

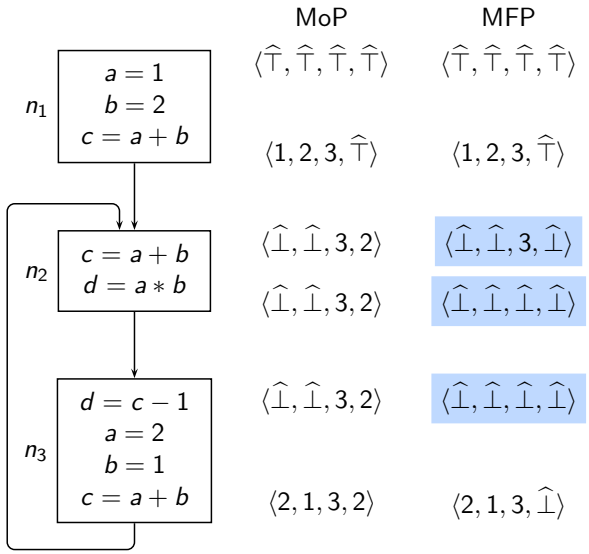
Constant Propagation

Separability of Bit Vector Frameworks

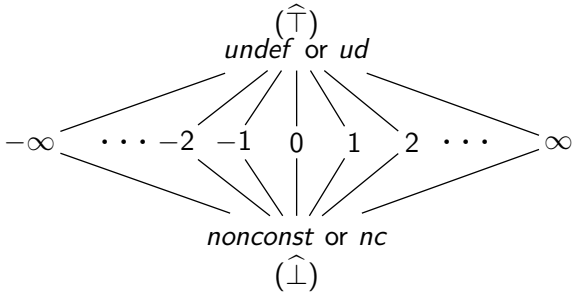
- $\widehat{L}$ is $\{0,1\}$, L is $\{0,1\}^m$
- $\widehat{\sqcap}$ is either boolean AND or boolean OR
- $\widehat{\top}$ and $\widehat{\bot}$ are 0 or 1 depending on $\widehat{\sqcap}$.
- $\widehat{h}$ is a *bit function* and could be one of the following:



Example of Constant Propagation



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 $\widehat{\sqsubseteq}$.

$\widehat{\sqcap}$	$\langle v, ud\rangle$	$\langle v, nc\rangle$	$\langle v, c_1\rangle$
$\langle v, ud\rangle$	$\langle v, ud\rangle$	$\langle v, nc\rangle$	$\langle v, c_1\rangle$
$\langle v, nc\rangle$	$\langle v, nc\rangle$	$\langle v, nc\rangle$	$\langle v, nc\rangle$
$\langle v, c_2\rangle$	$\langle v, c_2\rangle$	$\langle v, nc\rangle$	If $c_1 = c_2$ then $\langle v, c_1\rangle$ else $\langle v, nc\rangle$

Flow Functions for Constant Propagation

- Flow function for $r = a_1 * a_2$

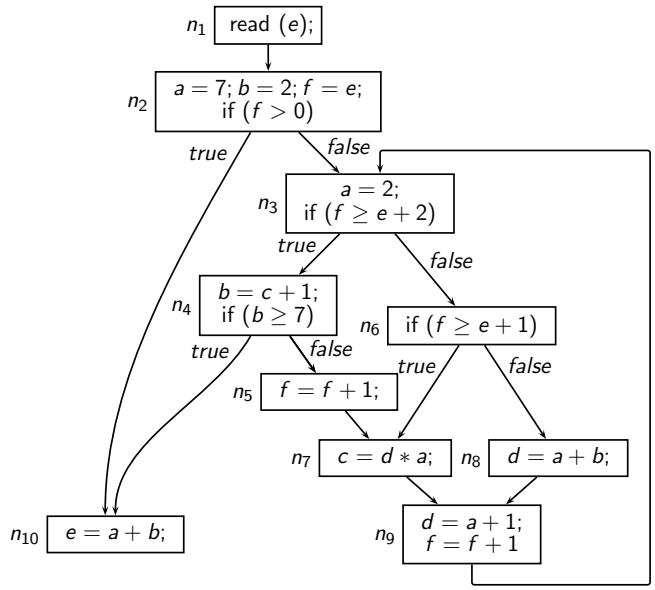
mult	$\langle a_1, ud\rangle$	$\langle a_1, nc\rangle$	$\langle a_1, c_1\rangle$
$\langle a_2, ud\rangle$	$\langle r, ud\rangle$	$\langle r, nc\rangle$	$\langle r, ud\rangle$
$\langle a_2, nc\rangle$	$\langle r, nc\rangle$	$\langle r, nc\rangle$	$\langle r, nc\rangle$
$\langle a_2, c_2\rangle$	$\langle r, ud\rangle$	$\langle r, nc\rangle$	$\langle r, (c_1 * c_2)\rangle$

Defining Data Flow Equations for Constant Propagation

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

$eval(a_1\ op\ a_2, X)$			
	$\langle a_1, ud\rangle \in X$	$\langle a_1, nc\rangle \in X$	$\langle a_1, c_1\rangle \in X$
$\langle a_2, ud\rangle \in X$	*ud*	*nc*	*ud*
$\langle a_2, nc\rangle \in X$	*nc*	*nc*	*nc*
$\langle a_2, c_2\rangle \in X$	*ud*	*nc*	$c_1\ op\ c_2$

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}	$\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\top}$			
Out_{n_1}	$\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\top}$			
In_{n_2}	$\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\top}$			
Out_{n_2}	$7,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$			
In_{n_3}	$7,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},2,6,3,\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$
Out_{n_3}	$2,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$
In_{n_4}	$2,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$
Out_{n_4}	$2,\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,\widehat{\top},\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,7,6,3,\widehat{\bot},\widehat{\bot}$	
In_{n_5}	$2,\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,\widehat{\top},\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,7,6,3,\widehat{\bot},\widehat{\bot}$	
Out_{n_5}	$2,\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,\widehat{\top},\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,7,6,3,\widehat{\bot},\widehat{\bot}$	
In_{n_6}	$2,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$
Out_{n_6}	$2,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$
In_{n_7}	$2,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$	
Out_{n_7}	$2,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$	
In_{n_8}	$2,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$
Out_{n_8}	$2,2,\widehat{\top},4,\widehat{\bot},\widehat{\bot}$	$2,2,\widehat{\top},4,\widehat{\bot},\widehat{\bot}$	$2,2,6,4,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,\widehat{\bot},\widehat{\bot},\widehat{\bot}$
In_{n_9}	$2,2,\widehat{\top},4,\widehat{\bot},\widehat{\bot}$	$2,2,6,\widehat{\bot},\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,\widehat{\bot},\widehat{\bot},\widehat{\bot}$	
Out_{n_9}	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$	
$In_{n_{10}}$	$\widehat{\bot},2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$	
$Out_{n_{10}}$	$\widehat{\bot},2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$	

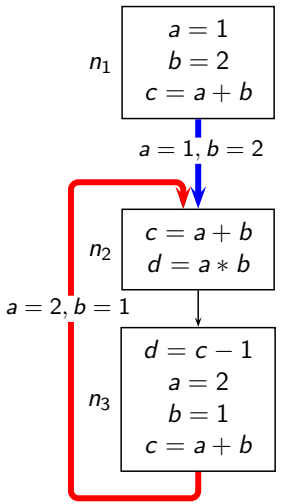
Monotonicity of Constant Propagation

- 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 $X_1 \sqsubseteq X_2 \Rightarrow (X_1 - \mathit{DepKill}_n(X_1)) \sqsubseteq (X_1 - \mathit{DepKill}_n(X_2))$ observe that $\mathit{DepKill}_n$ removes the pair corresponding to the variable modified in statement n. Data flow values of other variables remain unaffected.

Non-Distributivity of Constant Propagation



- $x = \langle 1, 2, 3, ? \rangle$ (Along $Out_{n_1} \rightarrow In_{n_2}$)
- $y = \langle 2, 1, 3, 2 \rangle$ (Along $Out_{n_3} \rightarrow In_{n_2}$)
- Function application before merging

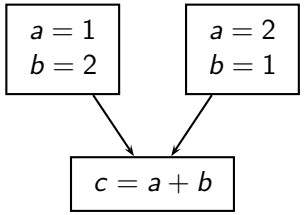
$$\begin{aligned} f(x) \sqcap f(y) &= f(\langle 1, 2, 3, ? \rangle) \sqcap f(\langle 2, 1, 3, 2 \rangle) \\ &= \langle 1, 2, 3, 2 \rangle \sqcap \langle 2, 1, 3, 2 \rangle \\ &= \langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle \end{aligned}$$

- Function application after merging

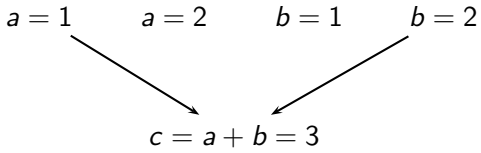
$$\begin{aligned} f(x \sqcap y) &= f(\langle 1, 2, 3, ? \rangle \sqcap \langle 2, 1, 3, 2 \rangle) \\ &= f(\langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle) \\ &= \langle \widehat{\bot}, \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle \end{aligned}$$

- $f(x \sqcap y) \sqsubset f(x) \sqcap f(y)$

Why is Constant Propagation Non-Distributive?

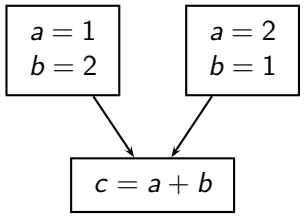


Possible combinations due to merging

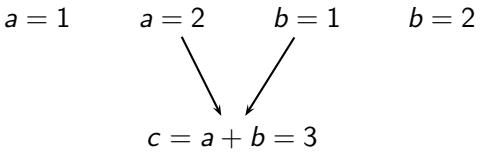


- Correct combination.

Why is Constant Propagation Non-Distributive?

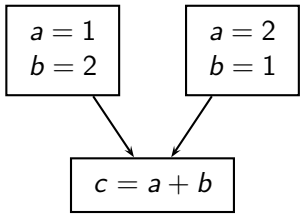


Possible combinations due to merging

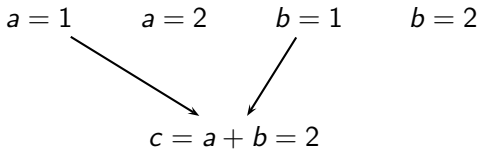


- Correct combination.

Why is Constant Propagation Non-Distributive?

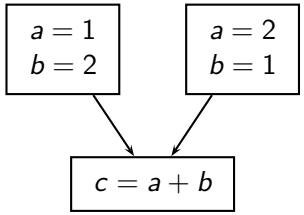


Possible combinations due to merging

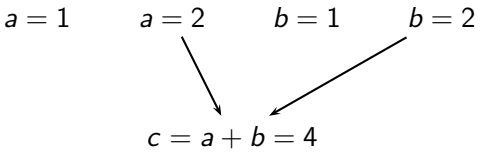


- Wrong combination.
- Mutually exclusive information.
- No execution path along which this information holds.

Why is Constant Propagation Non-Distributive?

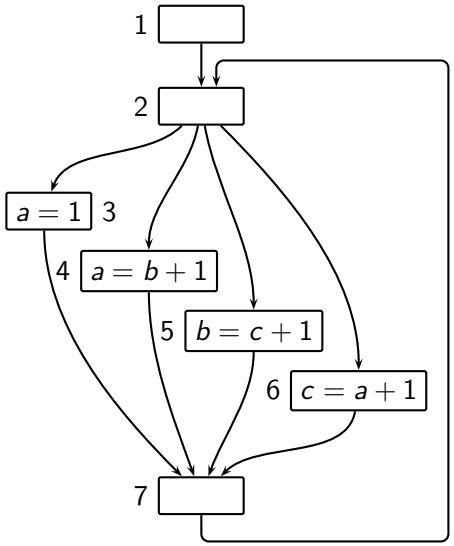


Possible combinations due to merging



- Wrong combination.
- Mutually exclusive information.
- No execution path along which this information holds.

Boundedness of Constant Propagation

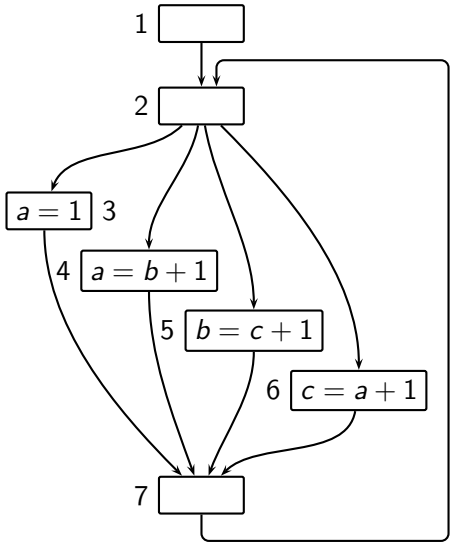


Summary flow function:
(data flow value at node 7)

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

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

Boundedness of Constant Propagation



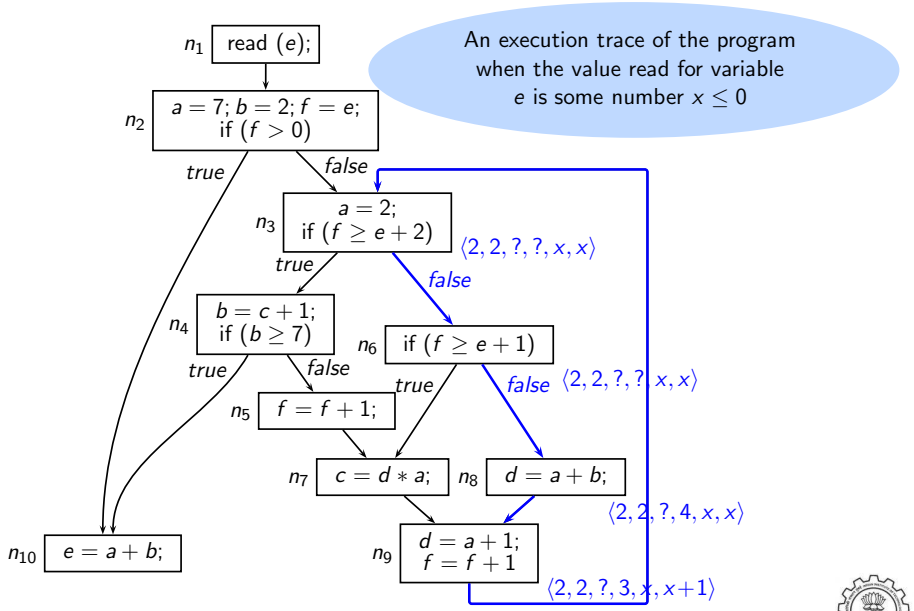
$$f^*(\top) = \prod_{i=0}^{6} f^i(\top)$$

Boundedness of Constant Propagation

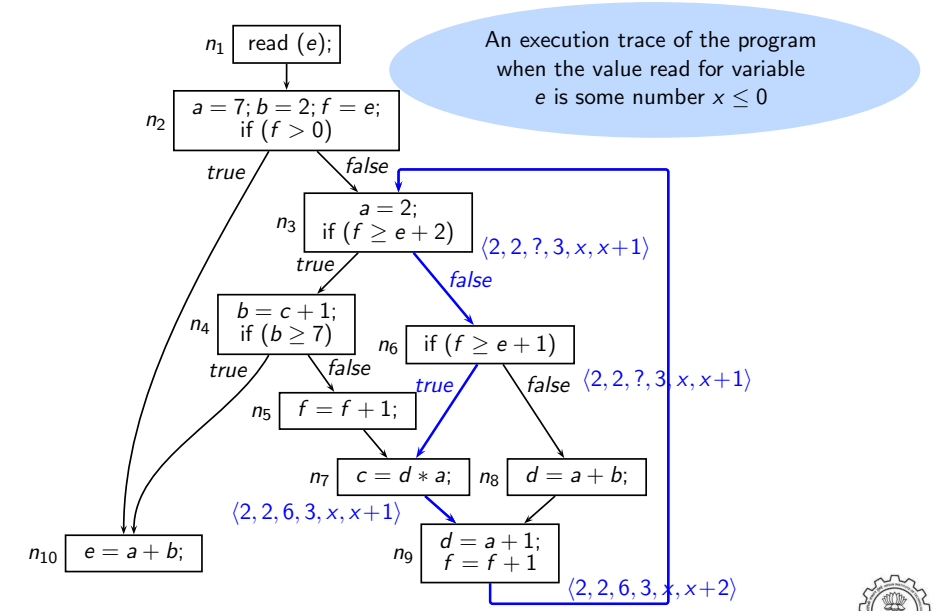
The moral of the story:

- The data flow value of every variable could change twice
- 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 $(2 \times |\mathbb{V}\text{ar}|) + 1$

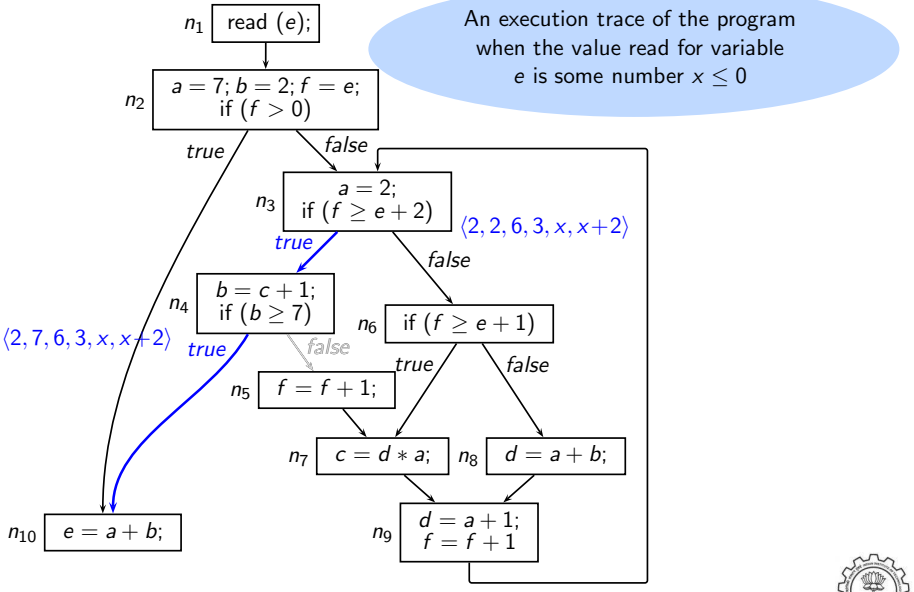
Conditional Constant Propagation



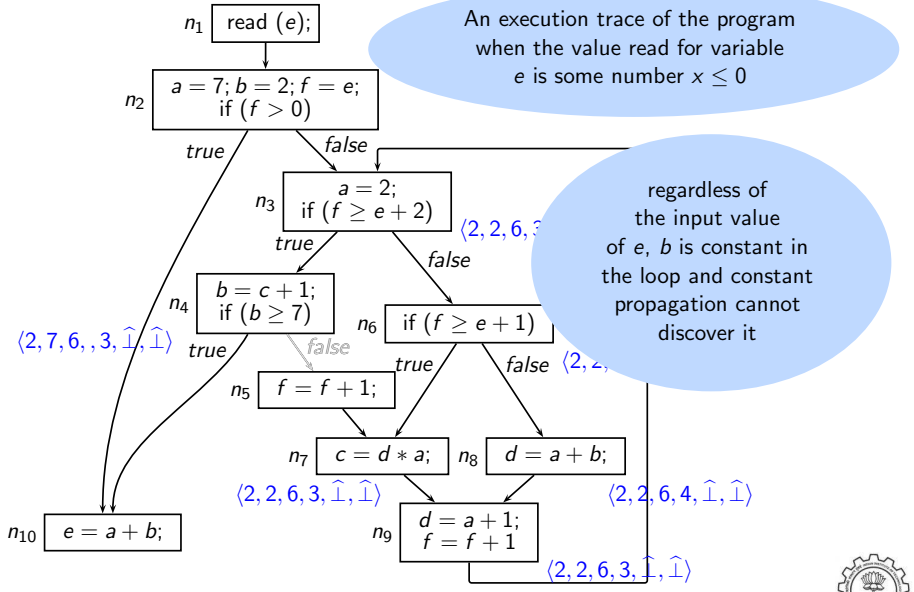
Conditional Constant Propagation



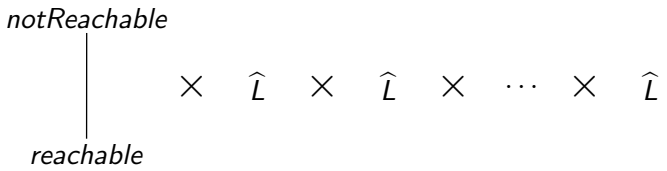
Conditional Constant Propagation



Conditional Constant Propagation



Lattice for Conditional Constant Propagation



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

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

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

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

Conditional Constant Propagation

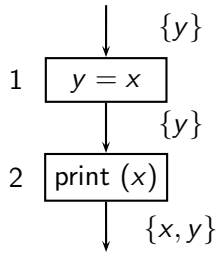
	Iteration #1	Changes in iteration #2	Changes in iteration #3
In_{n_1}	$R, \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$		
Out_{n_1}	$R, \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\top} \rangle$		
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}	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
$In_{n_{10}}$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, \widehat{\bot}, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
$Out_{n_{10}}$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, \widehat{\bot}, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$

Part 4

Faint Variables Analysis

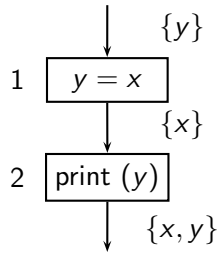
Faint Variables Analysis

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



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

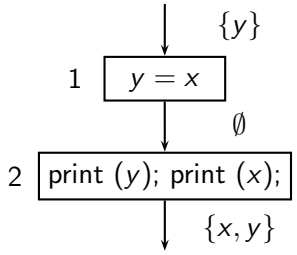
Faintness of x is killed by the print statement (i.e. x becomes live)



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

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

Faint Variables Analysis



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

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

Data Flow Equations for Faint Variables Analysis

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

where,

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

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)
$ConstGen_n$	$x \notin Opd(e) \Rightarrow \{x\}$ $x \in Opd(e) \Rightarrow \emptyset$	$\{x\}$	$\emptyset$
$ConstKill_n$	$\emptyset$	$\emptyset$	$\{x\}$
$DepGen_n(X)$	$\emptyset$	$\emptyset$	$\emptyset$
$DepKill_n(X)$	$x \notin X \Rightarrow Opd(e) \cap \mathbb{V}\text{ar}$ $x \in X \Rightarrow \emptyset$	$\emptyset$	$\emptyset$

Why is $DepGen_n(X) = \emptyset$ in Faint Variables Analysis?

Faintness can only be generated by an assignment statement, a read statement, or BI.
Consider an assignment statement $x = e$ where $e \in \mathbb{E}\text{xpr}$

- If $x \notin Opd(e)$ then x becomes faint unconditionally.
  Case covered by $ConstGen_n$.
- For the operands of e (including x if it is in Opd(e))
  - If $x \notin X$ (i.e. x is not faint after the assignment), operands cannot be faint before the assignment.
  - If $x \in X$ (i.e. is faint after the assignment), faintness of operands depends on their uses after the assignment.
    - If they are faint after the assignment, they continue to remain faint.
    - If the are not faint after the assignment, they connot become faint before the assignment.

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

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

- $ConstGen_n$, $DepGen_n$, and $ConstKill_n$ are trivially distributive. Assume that $DepKill_n$ is $\emptyset$

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

Since $DepGen_n(X) = \emptyset$, the flow function has only constant parts!

Distributivity of Faint Variables Analysis

To show that

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

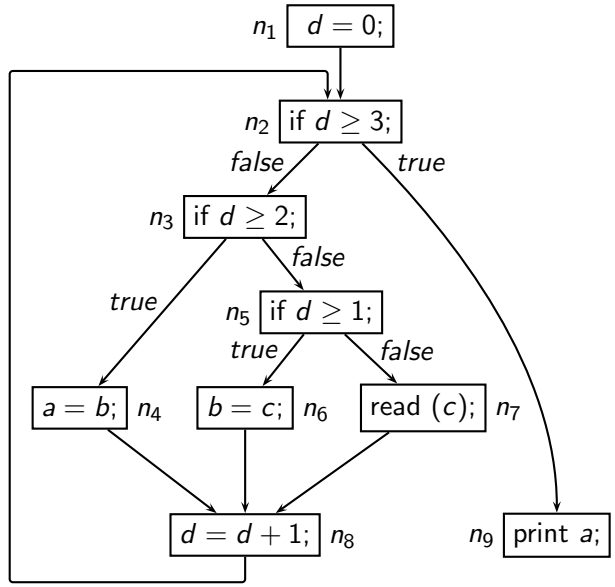
- If n is an assignment statement $x = e$, and $x \notin X_1 \cap X_2$. Assume that x is neither in X_1 nor in X_2.

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

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	$\{b,c\}$	$\{b,c,d\}$	$\{c\}$	$\{c,d\}$	$\emptyset$	$\{d\}$		

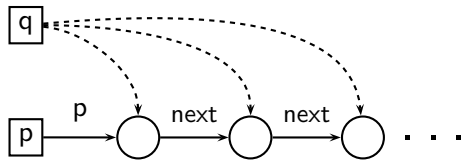
Part 5

Pointer Analyses

Code Optimization In Presence of Pointers

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

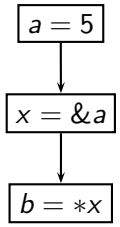
Program



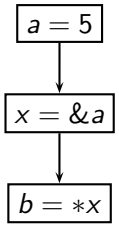
Memory graph at statement 5

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

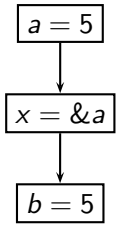
Code Optimization In Presence of Pointers



Original Program

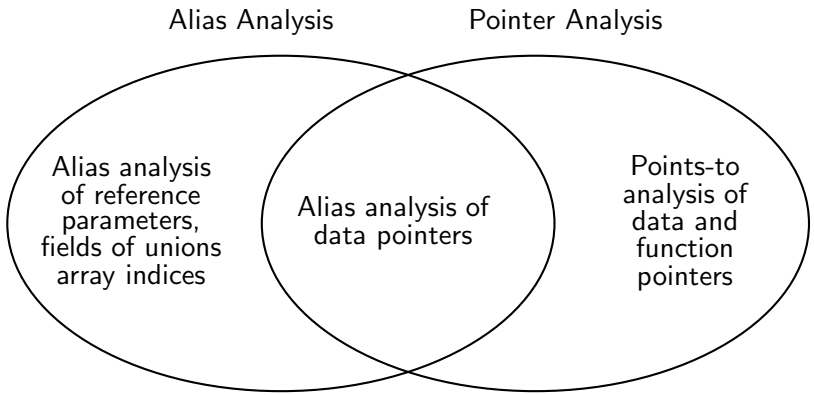


Constant Propagation without aliasing



Constant Propagation with aliasing

The World of Pointer Analysis



The Mathematics of Pointer Analysis

In the most general situation

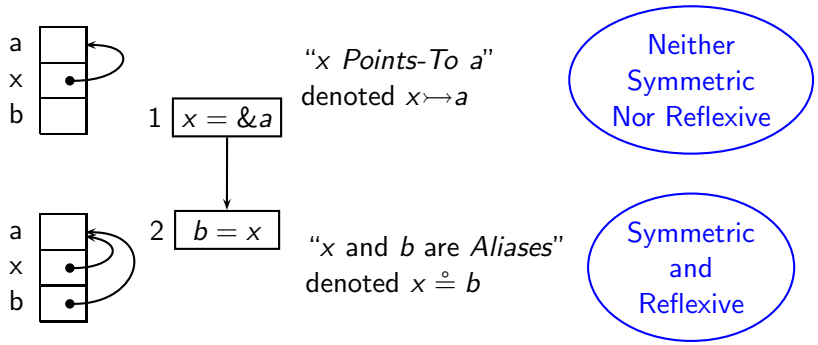
- Alias analysis is undecidable.
  Landi-Ryder [POPL 1991], Landi [LOPLAS 1992],
  Ramalingam [TOPLAS 1994]
- Flow insensitive alias analysis is NP-hard
  Horwitz [TOPLAS 1997]
- Points-to analysis is undecidable
  Chakravarty [POPL 2003]

Motivation for a Good Science of Pointer Analysis

- To quote Hind [PASTE 2001]
  - 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.

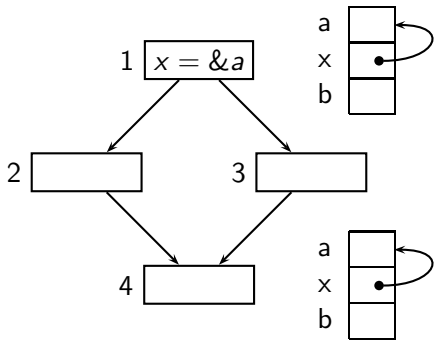
  $\Rightarrow$ Results in many questionable perceptions.

Alias Information Vs. Points-To Information

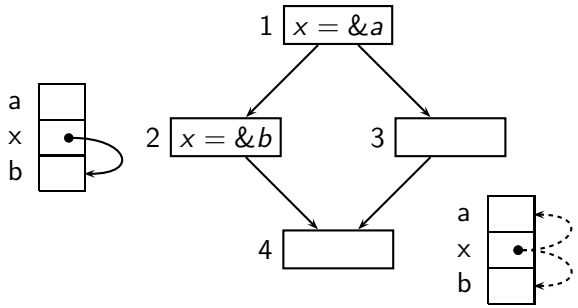


- What about transitivity?
  - Points-To: No.
  - Alias: Depends.

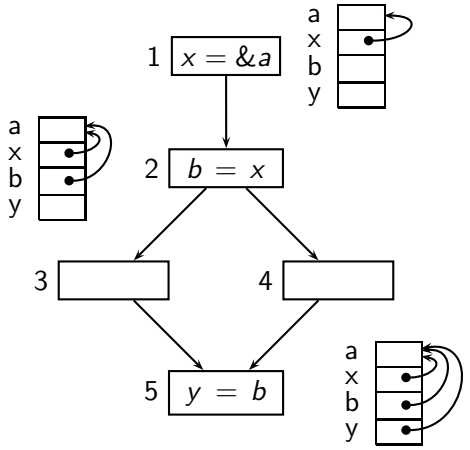
Must Points-To Information



May Points-To Information

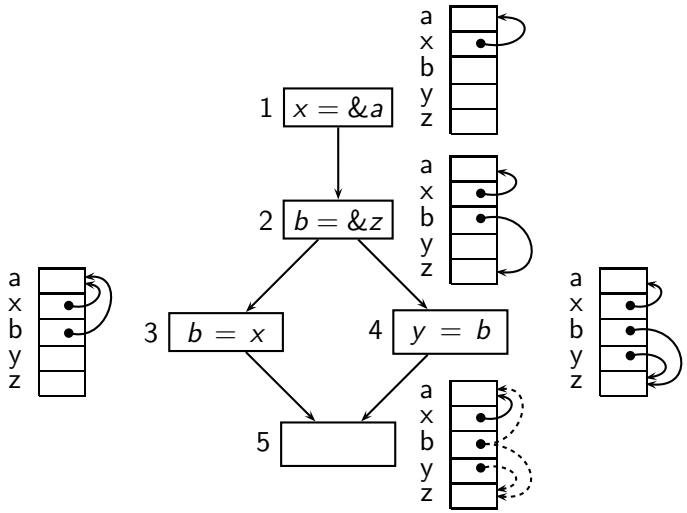


Must Alias Information



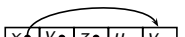
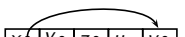
$x \stackrel{\circ}{=} b$ and $b \stackrel{\circ}{=} y \Rightarrow x \stackrel{\circ}{=} y$

May Alias Information

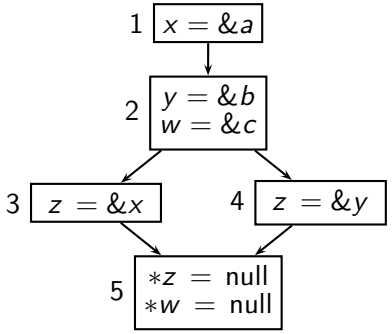


$x \stackrel{\circ}{=} b$ and $b \stackrel{\circ}{=} y \not\Rightarrow x \stackrel{\circ}{=} y$

A Comparison of Points-To and Alias Relations

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



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 givem 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	$\{x\}$	$\emptyset$
$*x$	$\emptyset$	$\{y \mid (x \rightarrowtail y) \in X\}$

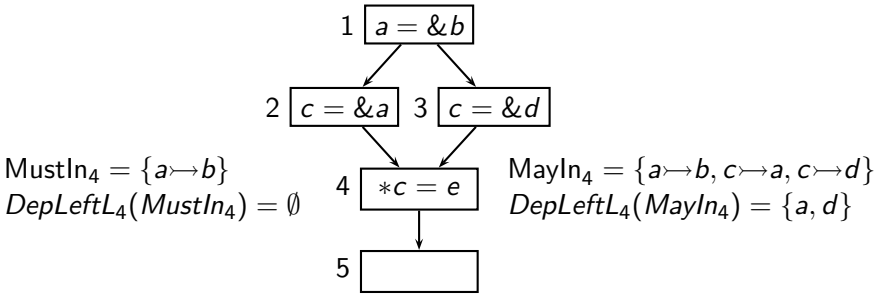
- Right Locations

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

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

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



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

$DepKill(X)$ 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

Data Flow Equations for Points-To Analysis

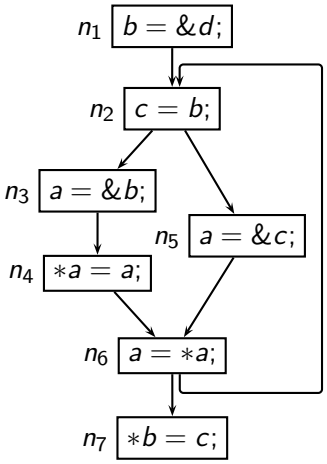
$$
\begin{aligned}
MayIn_n &= \begin{cases} BI & n \text{ is } Start \\ \bigcup_{p \in pred(n)} MayOut_n & \text{otherwise} \end{cases} \\
MayOut_n &= f_n(MayIn_n, MustIn_n) \\
MustIn_n &= \begin{cases} BI & n \text{ is } Start \\ \bigcap_{p \in pred(n)} MustOut_n & \text{otherwise} \end{cases} \\
MustOut_n &= f_n(MustIn_n, MayIn_n) \\
\\
f_n(X_1, X_2) &= (X_1 - \text{Kill}_n(X_2)) \cup \text{Gen}_n(X_1)
\end{aligned}
$$

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

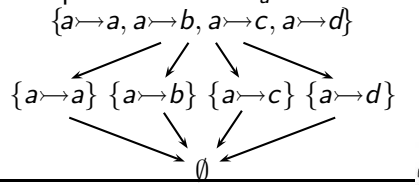


- Variables and points-to sets:

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

- $L_{may} = \langle 2^{\mathbb{U}}, \supseteq\rangle,\ \top_{may} = \emptyset, \bot_{may} = \mathbb{U}$
- $L_{must} = \widehat{L}_a \times \widehat{L}_b \times \widehat{L}_c \times \widehat{L}_d$

  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

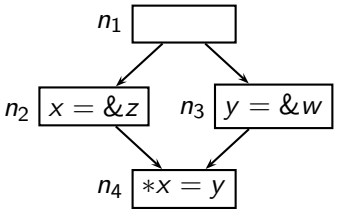
	Iteration #1	Changes in Iteration #2	Changes in Iteration #3
$MayIn_{n_4}$	$\{a\rightarrowtail b, b\rightarrowtail d, c\rightarrowtail d\}$	$\{a\rightarrowtail b, b\rightarrowtail b, b\rightarrowtail d, c\rightarrowtail b, c\rightarrowtail d\}$	
$MustIn_{n_4}$	$\{a\rightarrowtail b, b\rightarrowtail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

Result of Pointer Analysis

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

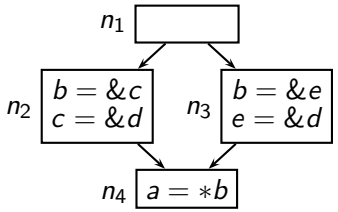
Non-Distributivity of Points-To Analysis

May Points-To



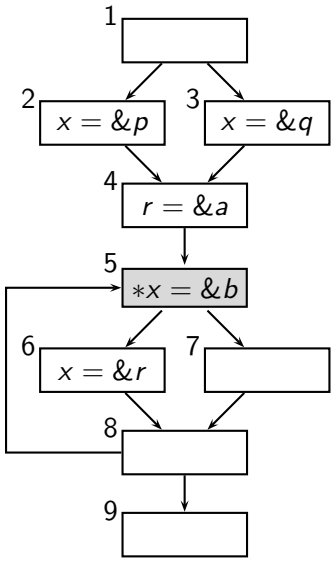
$z \rightarrowtail w$ is spurious

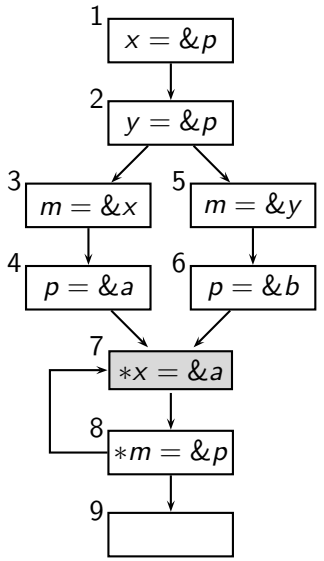
Must Points-To



$a \rightarrowtail d$ is missing

Tutorial Problems for May and Must Points-To Analysis





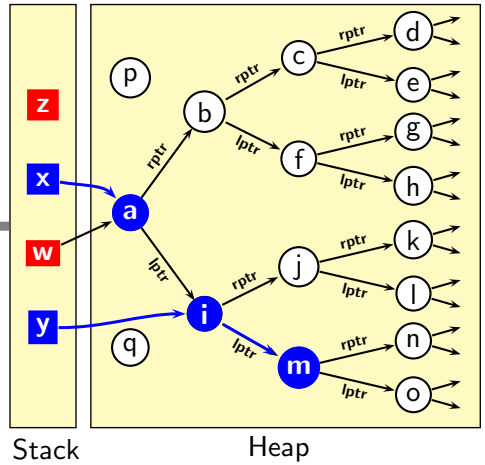
Part 6

Heap Reference Analysis

Motivating Example for Heap Liveness Analysis

If the while loop is not executed even once.

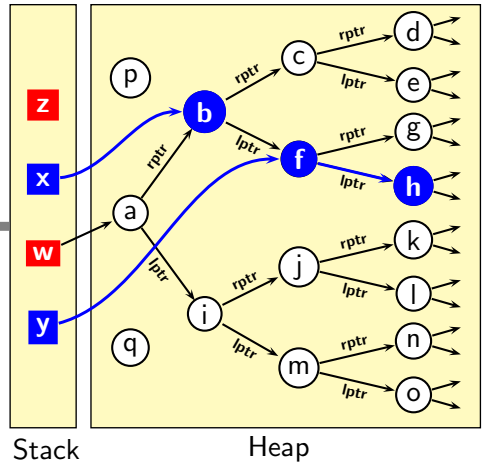
```
1  w = x          // x points to m_a
2  while (x.data < max)
3        x = x.rptr
4  y = x.lptr
5  z = New class_of_z
6  y = y.lptr
7  z.sum = x.data + y.data
```



Motivating Example for Heap Liveness Analysis

If the while loop is executed once.

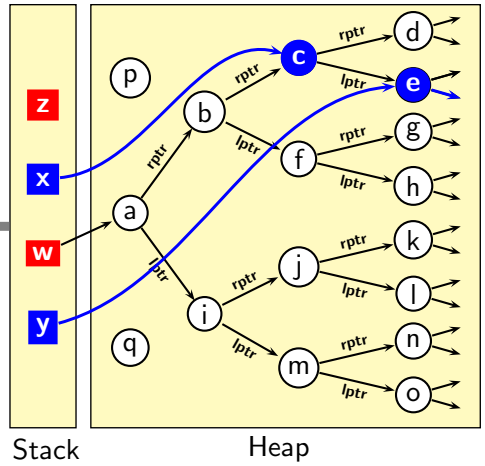
```
1  w = x          // x points to m_a
2  while (x.data < max)
3        x = x.rptr
4  y = x.lptr
5  z = New class_of_z
6  y = y.lptr
7  z.sum = x.data + y.data
```



Motivating Example for Heap Liveness Analysis

If the while loop is executed twice.

```
1  w = x          // x points to m_a
2  while (x.data < max)
3        x = x.rptr
4  y = x.lptr
5  z = New class_of_z
6  y = y.lptr
7  z.sum = x.data + y.data
```



The Moral of the Story

- Mappings between access expressions and l-values keep changing
- This is a *rule* for heap data
  For stack and static data, it is an *exception*!
- Static analysis of programs has made significant progress for stack and static data.

  What about heap data?
  - Given two access expressions at a program point, do they have the same l-value?
  - Given the same access expression at two program points, does it have the same l-value?

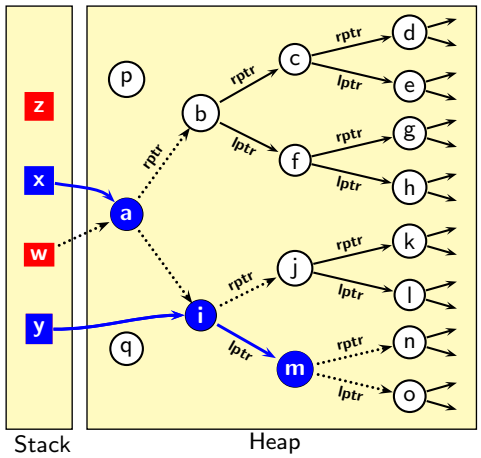
Our Solution

		y = z = 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
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
```

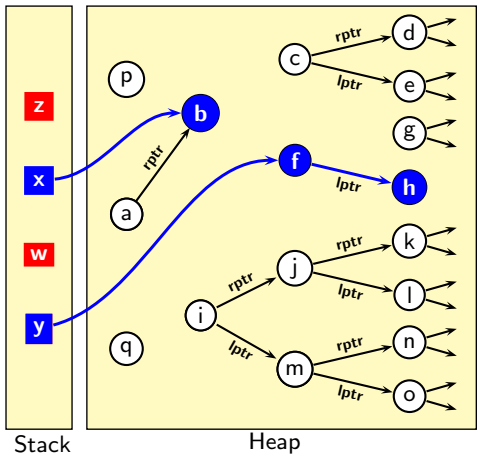
While loop is not executed even once



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
```

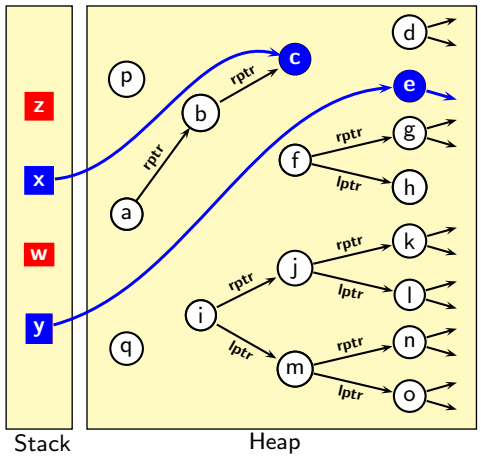
While loop is executed once



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
```

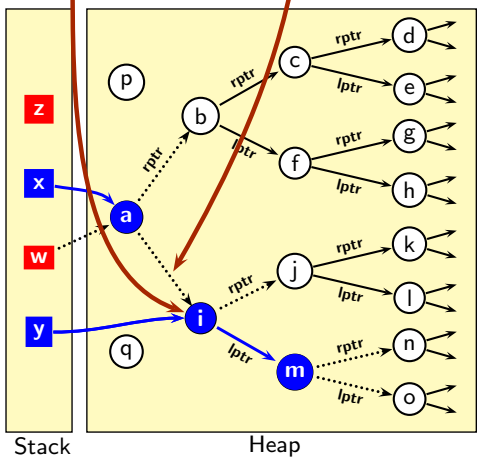
While loop is executed twice



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
```

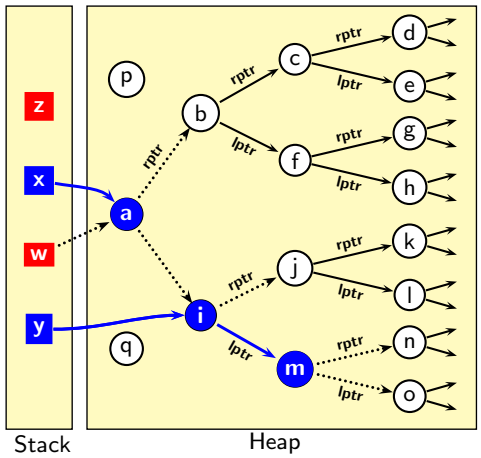
Node i is live but link $a \rightarrow i$ is nullified



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*.

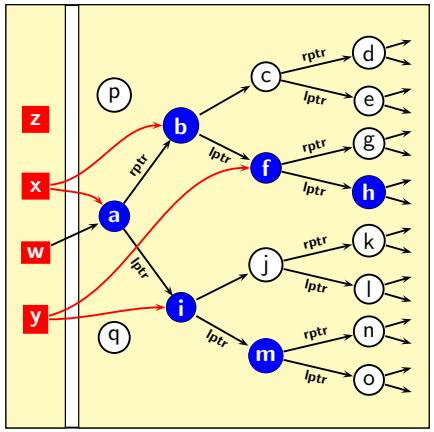
  Eg. "$x \rightarrow$ lptr $\rightarrow$ rptr"
- A link may be accessed in multiple ways
- Setting links to null
  - *Alias Analysis*. Identify all possible ways of accessing a link
  - *Liveness Analysis*. For each program point, identify "dead" links (i.e. links which are not accessed after that program 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



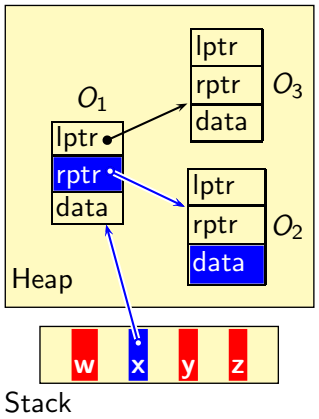
- Root variables : x, y, z
- Field names : rptr, lptr
- Access path : x→rptr→lptr
  Semantically, sequence of "links"
- Frontier : name of the last link
- Live access path : If the link corresponding to its frontier is used in future

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



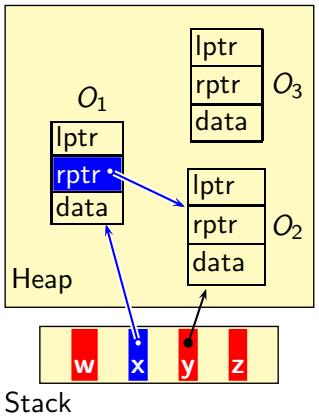
CS 618 General Frameworks: Heap Reference Analysis 65/98

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



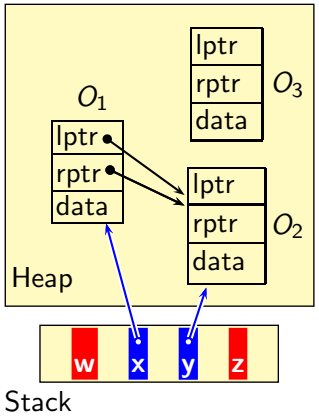
CS 618 General Frameworks: Heap Reference Analysis 65/98

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
x.lptr = y	x, O_1, y	x



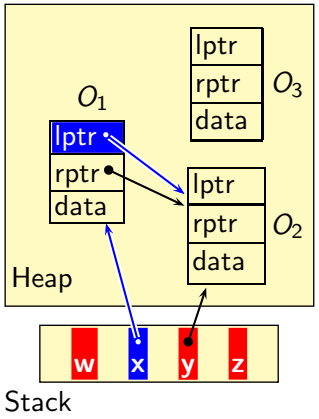
CS 618 General Frameworks: Heap Reference Analysis 65/98

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

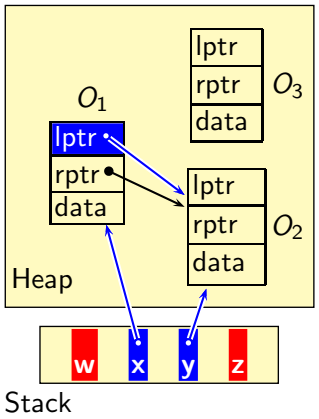


What Makes a Link Live?

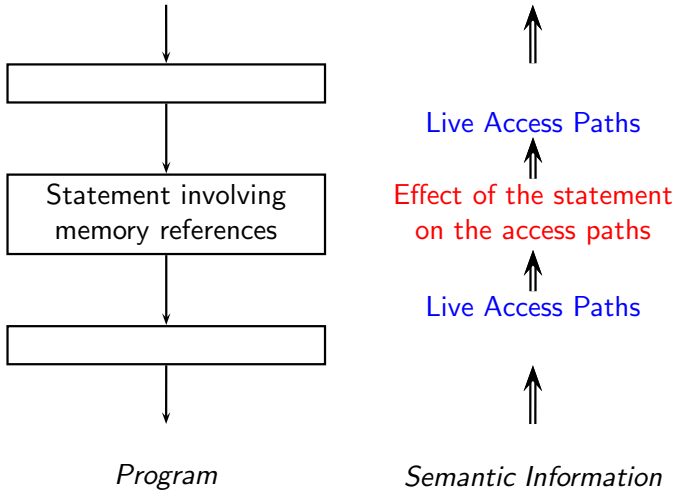
Assuming that a statement is the last statement in the program, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

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

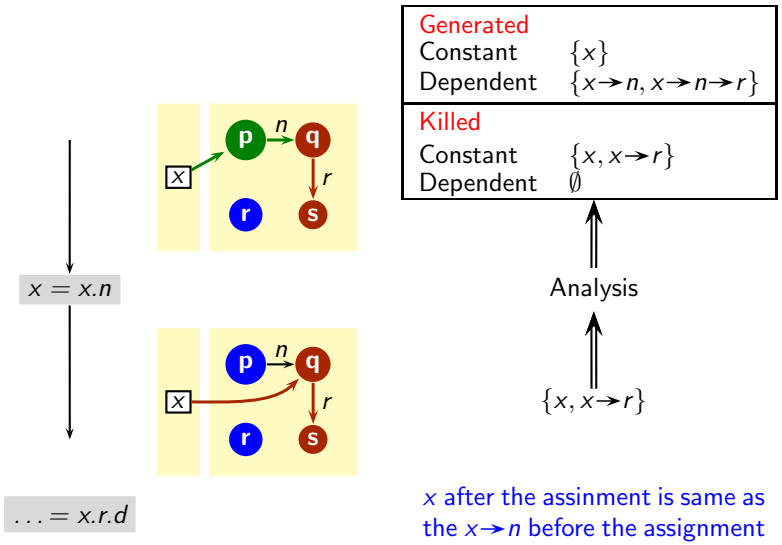
Example	Objects read	Live access paths
if (x.lptr == null)	x, O_1	$x, x\rightarrow$lptr
if ($y == x$.lptr)	x, O_1, y	$x, x\rightarrow$lptr, y



Liveness Analysis

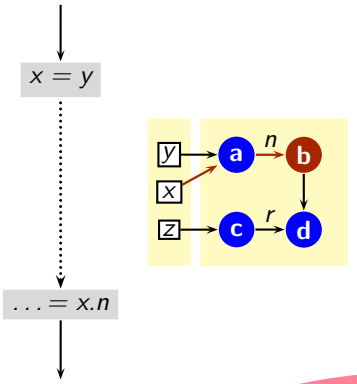


Key Idea #2 : Transfer of Access Paths



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

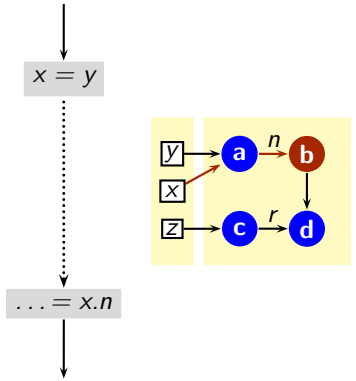
Key Idea #3 : Liveness Closure Under Link Aliasing



x and y are node aliases
$x.n$ 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

$y \rightarrow n$ 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.
- 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

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

ρ_x	frontier(ρ_x)	$base(\rho_x)$	$prefixes(\rho_x)$	$summary(\{\rho_x\})$
$x \rightarrow n \rightarrow r$	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

Flow Functions for Explicit Liveness Analysis

access expression

corresponding access path

Statement	$ConstKill$	$DepKill(X)$	$ConstGen$	$DepGen(X)$
Use α_y	$\emptyset$	$\emptyset$	$prefixes(base(\rho_y))$	$\emptyset$
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	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

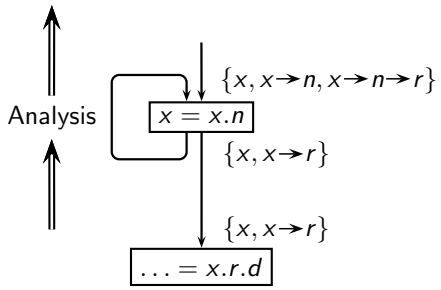
End of procedure

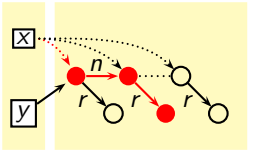
Transfer

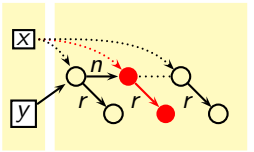
Flow Functions for Handling Procedure Calls in Computing Explicit Liveness

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

Computing Explicit Liveness Using Sets of Access Paths

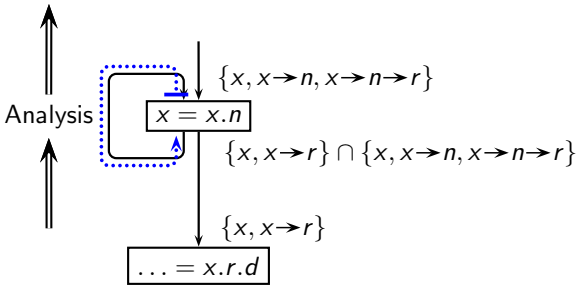




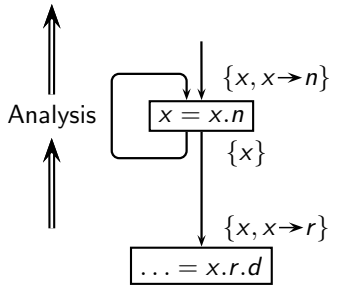


Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An *All Paths* problem

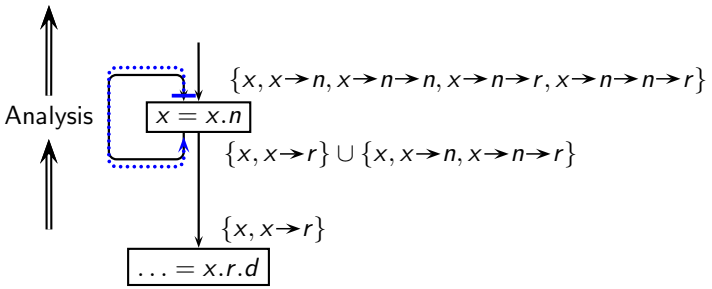


Computing Explicit Liveness Using Sets of Access Paths



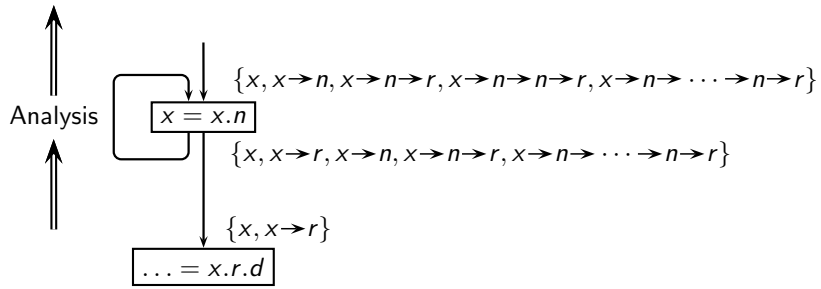
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



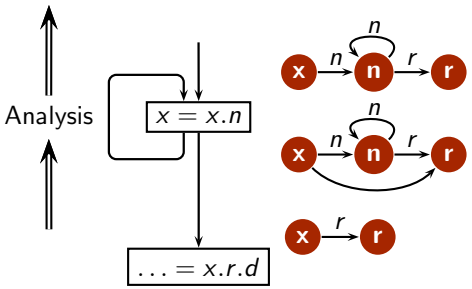
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



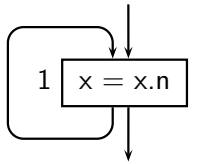
Infinite Number of Unbounded Access Paths

Key Idea #5: Using Graphs as Data Flow Values



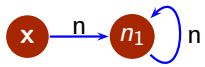
Finite Number of Bounded Structures

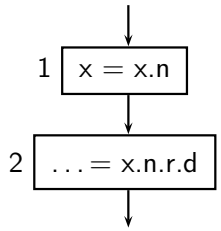
Key Idea #6 : Include Program Point in Graphs



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

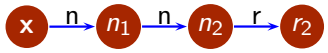
Different occurrences of n's in an access path are **Indistinguishable**

Access Graph : 

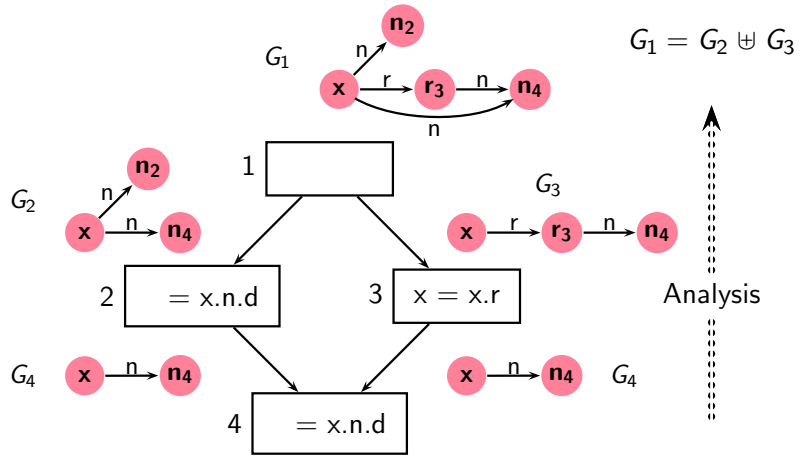


$\{x, x\to n, x\to n\to n, x\to n\to n\to r\}$

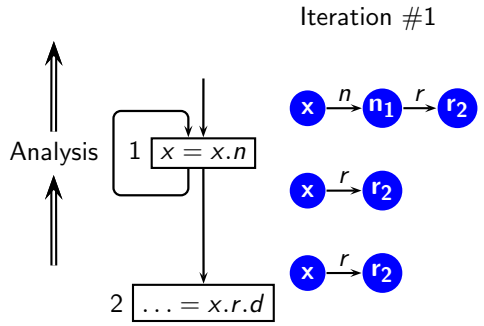
Different occurrences of n's in an access path are **Distinct**

Access Graph : 

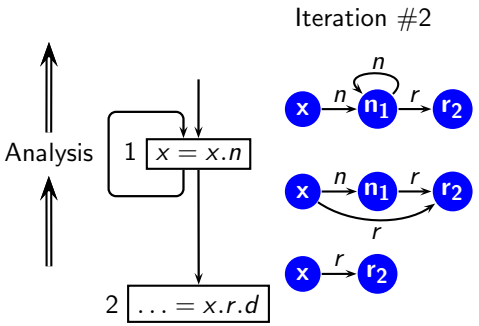
Inclusion of Program Point Facilitates Summarization



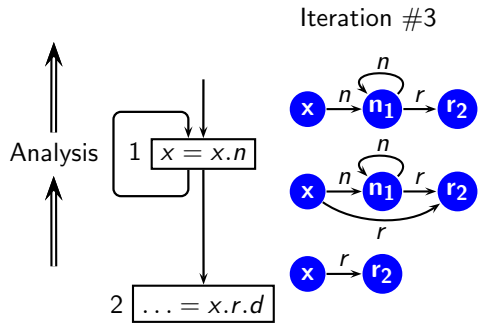
Inclusion of Program Point Facilitates Summarization



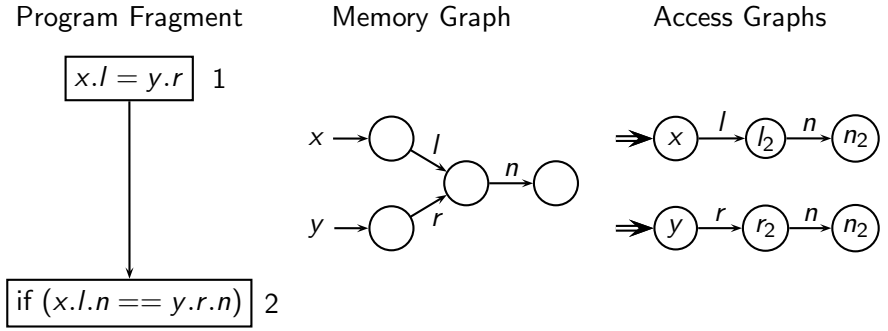
Inclusion of Program Point Facilitates Summarization



Inclusion of Program Point Facilitates Summarization



Access Graph and Memory Graph



- Memory Graph: Nodes represent locations and edges represent links (i.e. pointers).
- Access Graphs: 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.

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

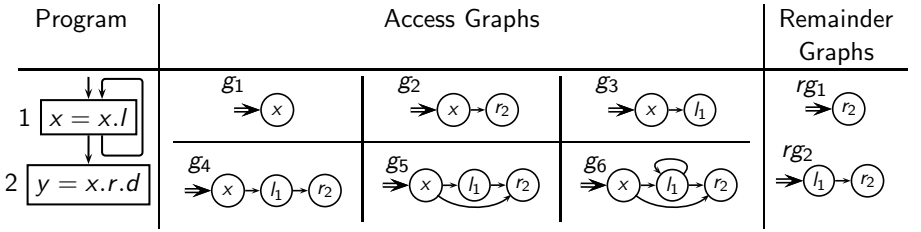
- $P(G, M)$ is the set of paths in graph G terminating on nodes in M. For graph G_i, M_i is the set of all nodes in G_i.
- S is the set of remainder graphs and $P(S, M_s)$ is the set of all paths in all remainder graphs in S.

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

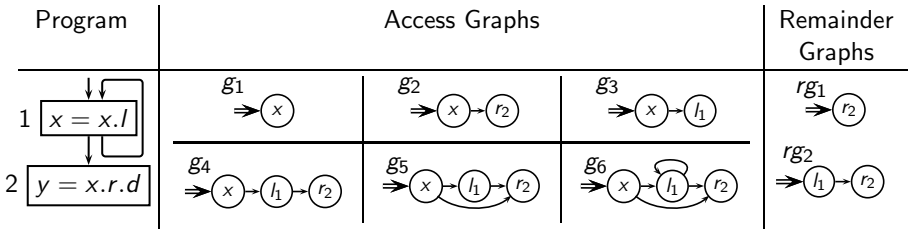
ρ' represents quotient

Access Graph Operations: Examples



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

Access Graph Operations: Examples



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

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.

$$ELIn_n(v) = (ELOut_n(v) \ominus ELKillPath_n(v)) \uplus ELGen_n(v)$$

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

$$ELGen_n(v) = ELConstGen_n(v) \uplus ELDepGen_n(v)$$

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

Flow Functions for Explicit Liveness Analysis

Access Paths ⟹

	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 ⇓

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

	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

The singleton set containing the last node corresponding to ρ_y

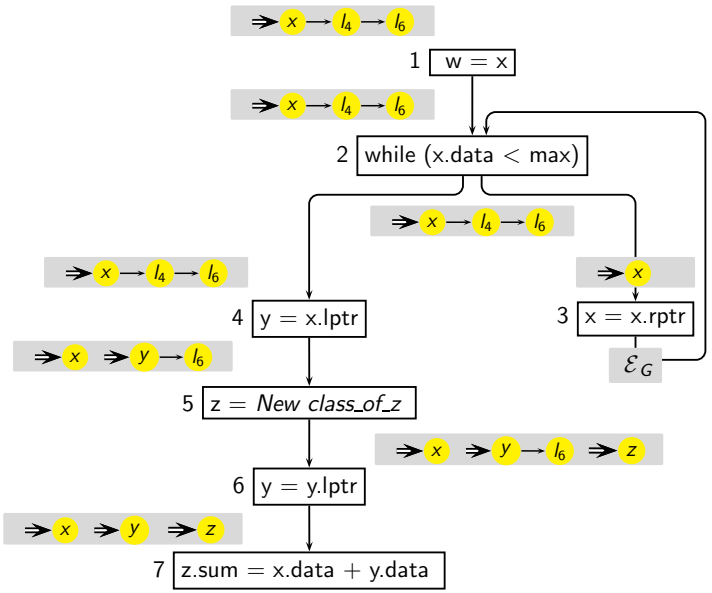
Access Paths ⟹

Access Graphs ⇓

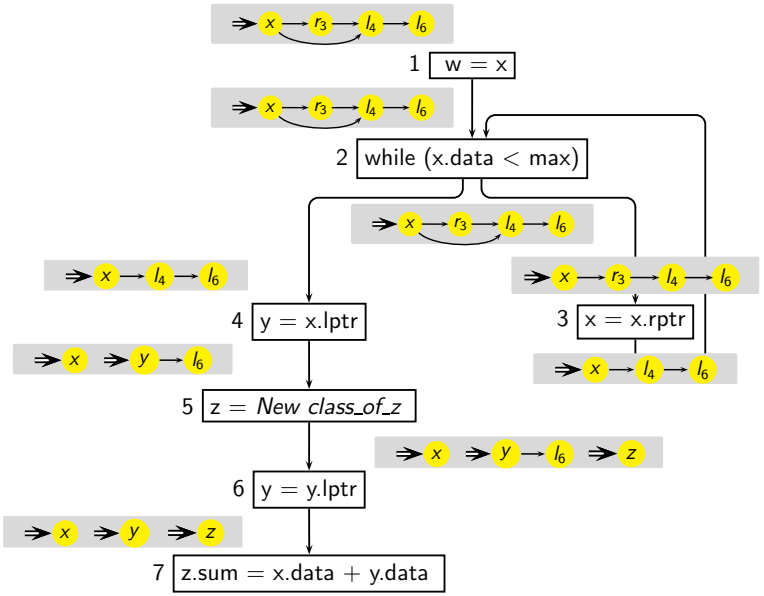
$G_x = makeGraph(\rho_x) \quad G_x^B = makeGraph(base(\rho_x))$
$G_y = makeGraph(\rho_y) \quad G_y^B = makeGraph(base(\rho_y))$

	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	$\mathcal{E}_G$
$ELDepGen_n(x)(X)$	$\mathcal{E}_G$	$\mathcal{E}_G$	$\mathcal{E}_G$	$\mathcal{E}_G$
$ELDepGen_n(y)(X)$	$\mathcal{E}_G$	$\mathcal{E}_G$	$(G_y, M_y)\#(X/(G_x, M_x))$	$\mathcal{E}_G$

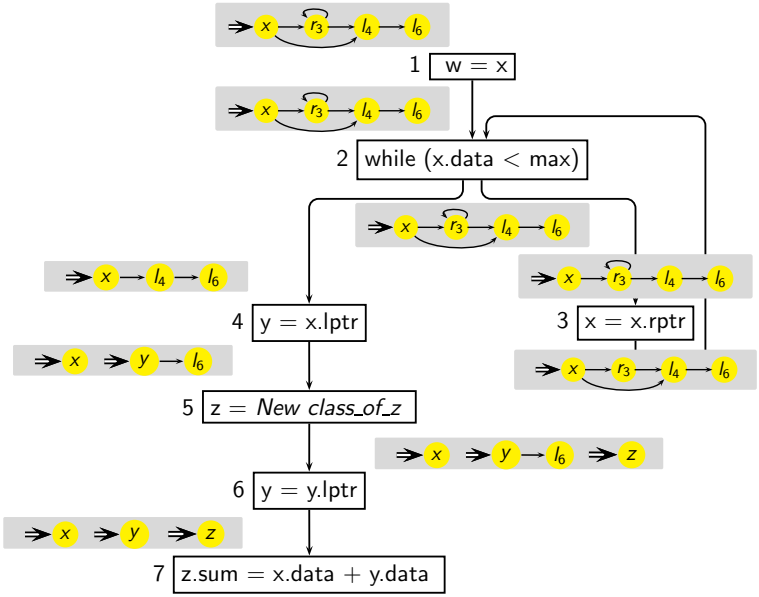
Liveness Analysis of Example Program: Ist Iteration



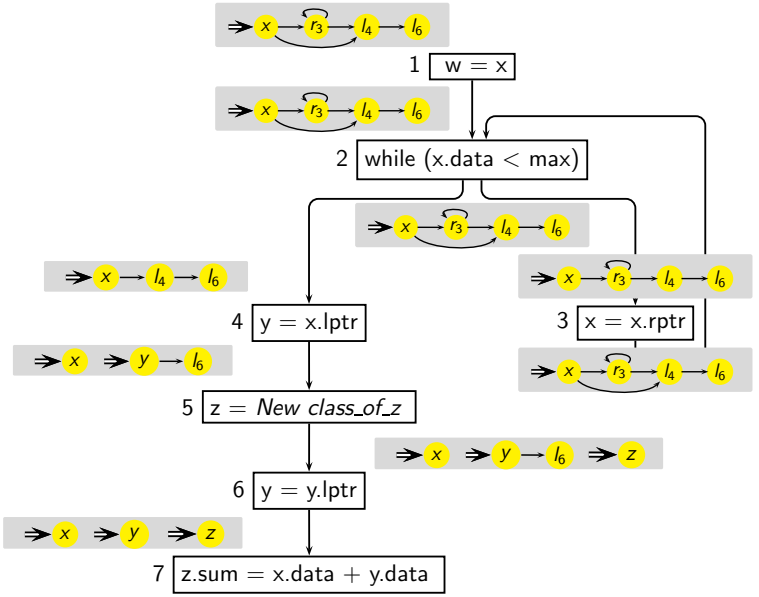
Liveness Analysis of Example Program: 2nd Iteration



Liveness Analysis of Example Program: 3rd Iteration



Liveness Analysis of Example Program: 4th Iteration



Which Access Paths Can be Nullified?

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 $\rho \rightarrow f$ = null at p subject to profitability

- For simple access paths, ρ is empty and f is the root variable name.

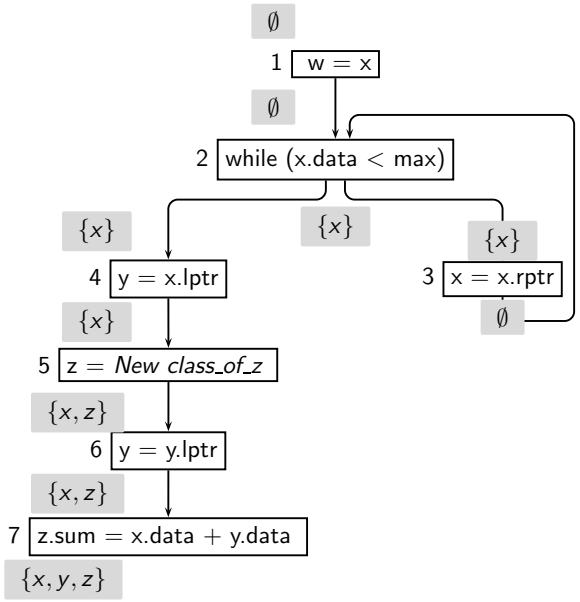
Cannot be hoisted and is not redefined at p

Availability and Anticipability Analyses

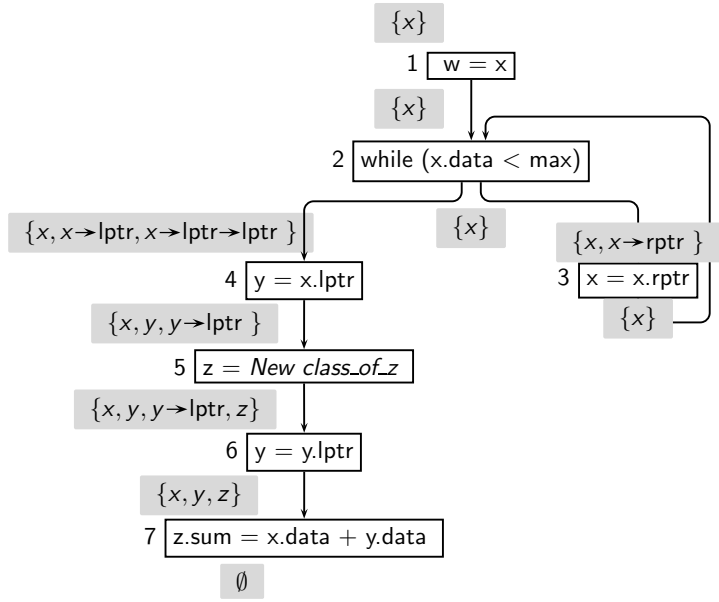
- ρ is available at program point p if the target of each prefix of ρ is guaranteed to be created along every control flow path reaching p.
- ρ is anticipable at program point p if the target of each prefix of ρ is guaranteed to be dereferenced along every control flow path starting at p.
- Finiteness.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

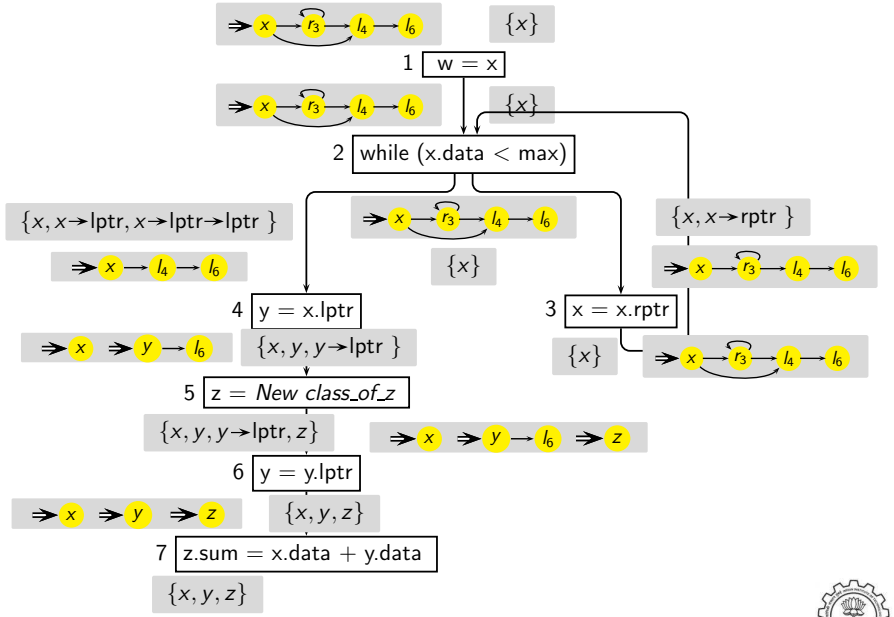
Availability Analysis of Example Program



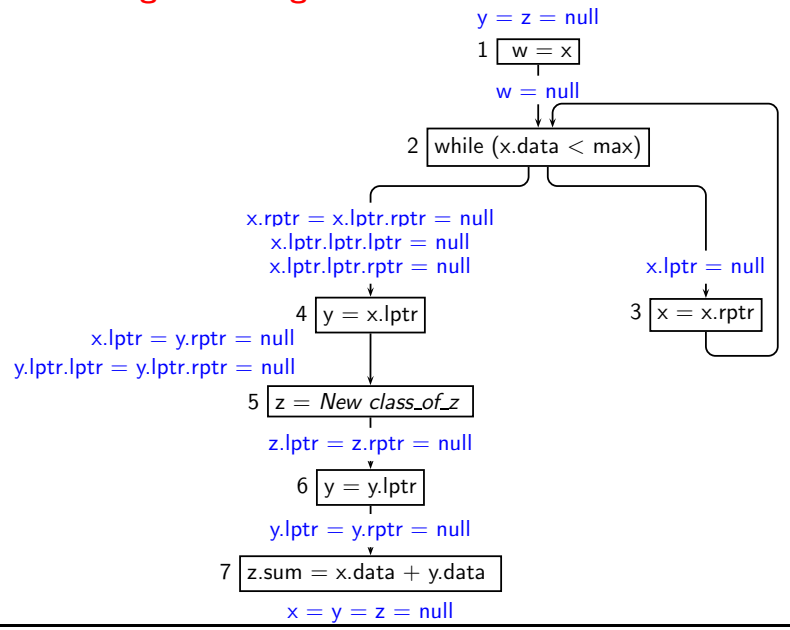
Anticipability Analysis of Example Program



Live and Accessible Paths



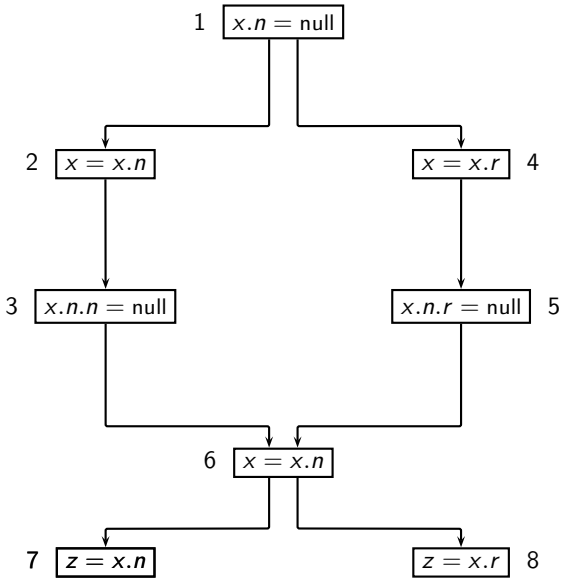
Creating null Assignments from Live and Accessible Paths



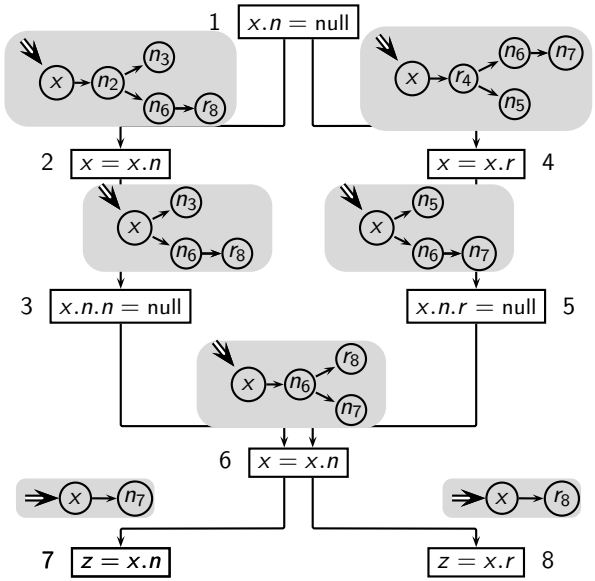
The Resulting Program

```
                                  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
```

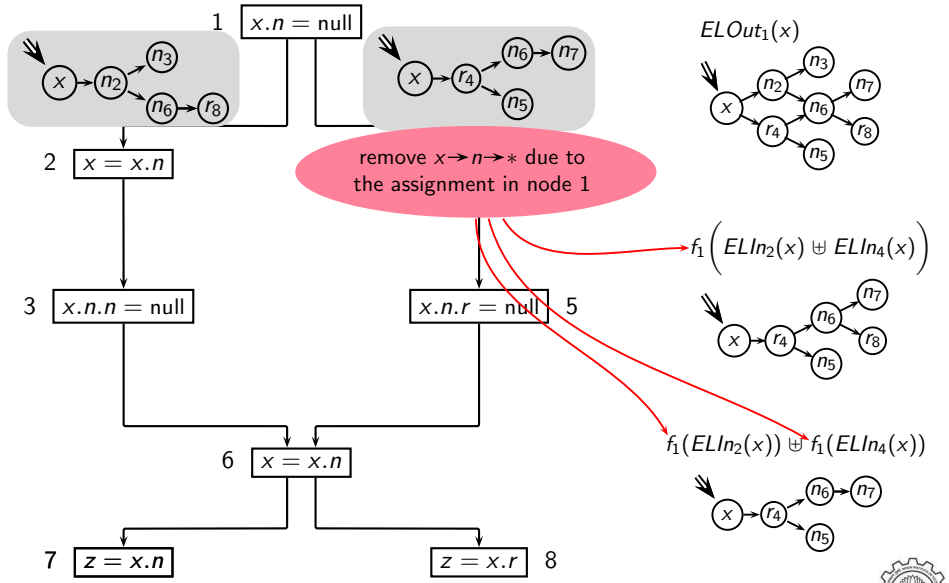

Non-Distributivity of Explicit Liveness Analysis



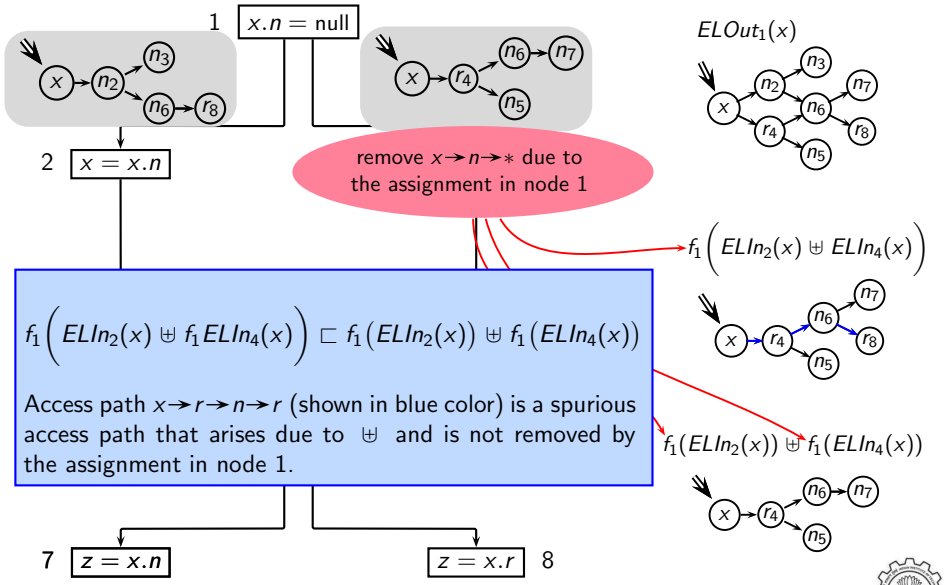
Non-Distributivity of Explicit Liveness Analysis



Non-Distributivity of Explicit Liveness Analysis



Non-Distributivity of Explicit Liveness Analysis



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++