General Data Flow Frameworks

Uday Khedker

(www.cse.iitb.ac.in/~uday)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

September 2018

Part 1

About These Slides

Copyright

These slides constitute the lecture notes for CS618 Program Analysis course at IIT Bombay and have been made available as teaching material accompanying the book:

- Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press (Taylor and Francis Group). 2009.

  (Indian edition published by Ane Books in 2013)

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

- M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland Inc. 1977.

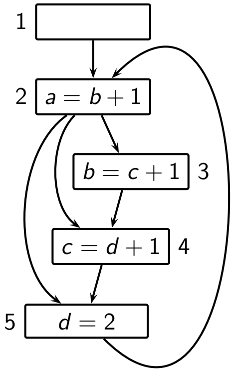
Outline

- Modelling General Flows
- Constant Propagation
- Strongly Live Variables Analysis (after mid-sem)
- Pointer Analyses (after mid-sem)
- Heap Reference Analysis (after mid-sem)

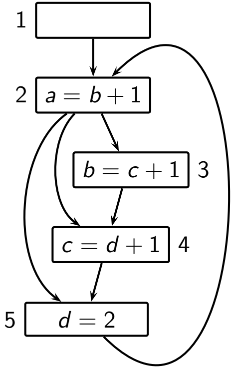
Part 2

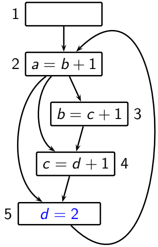
Precise Modelling of General Flows

Complexity of Constant Propagation?



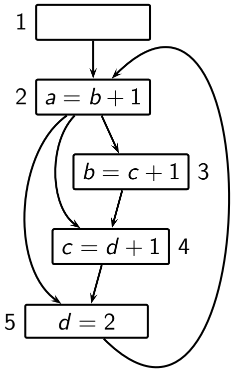
Complexity of Constant Propagation?

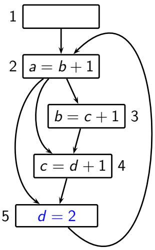




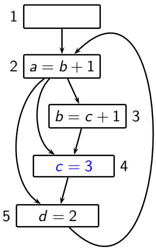
Iteration #1

Complexity of Constant Propagation?



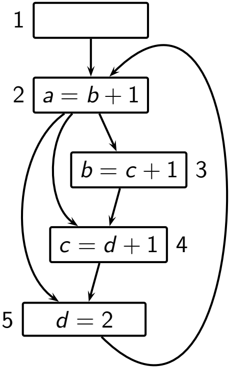


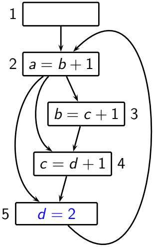
Iteration #1



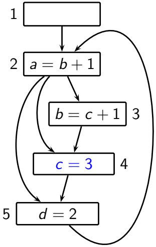
Iteration #2

Complexity of Constant Propagation?

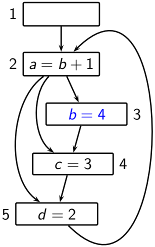




Iteration #1

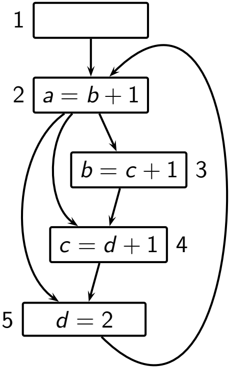


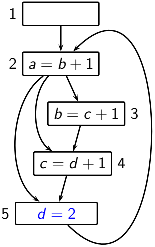
Iteration #2



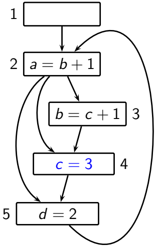
Iteration #3

Complexity of Constant Propagation?

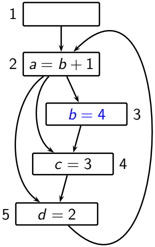




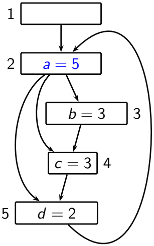
Iteration #1



Iteration #2

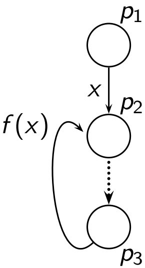


Iteration #3



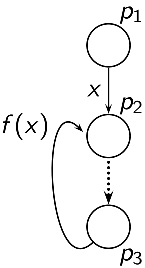
Iteration #4

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

Loop Closures of Flow Functions

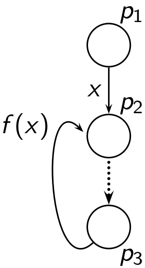


Paths Terminating at p_2	Data Flow Value
p_1, p_2	x
p_1, p_2, p_3, p_2	$f(x)$
$p_1, p_2, p_3, p_2, p_3, p_2$	$f(f(x)) = f^2(x)$
$p_1, p_2, p_3, p_2, p_3, p_2, p_3, p_2$	$f(f(f(x))) = f^3(x)$
...	...

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

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

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 Closure Boundedness

- Boundedness of f requires the existence of some k such that

$$f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap \ldots \sqcap f^{k-1}(x)$$

- This follows from the descending chain condition
- For efficiency, we need a constant k that is independent of the size of the lattice

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

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

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

Loop Closures in Bit Vector Frameworks

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

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

- *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

$f : L \rightarrow L$ 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

$f : L \rightarrow L$ 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

$f : L \to L$ 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

$f : L \to L$ 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

$f : L \rightarrow L$ 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

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

$$\widehat{h}_2$$

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

$$\boxed{\widehat{h} : \widehat{L} \rightarrow \widehat{L}}$$

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

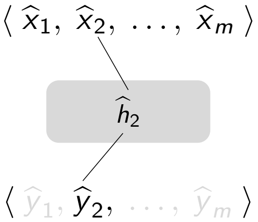
$f : L \to L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

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

Separability

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

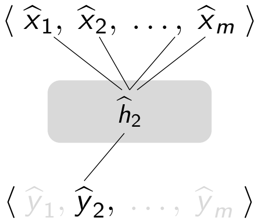
Separable



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

Example: All bit vector frameworks

Non-Separable

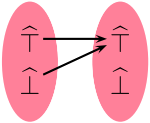
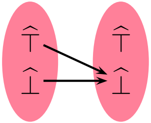
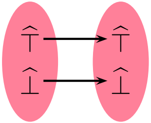
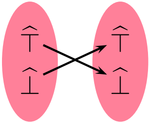


$\widehat{h} : L \rightarrow \widehat{L}$

Example: Constant Propagation

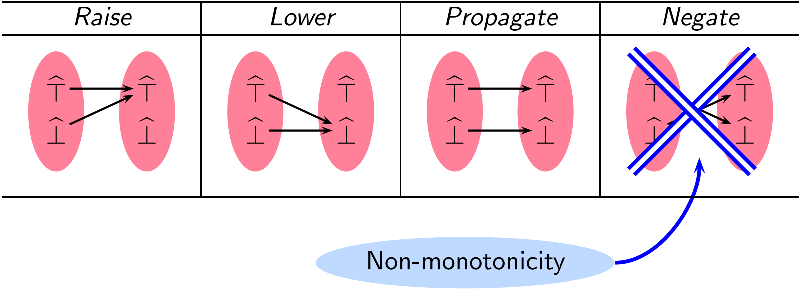
Separability of Bit Vector Frameworks

- $\widehat{L}$ 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:

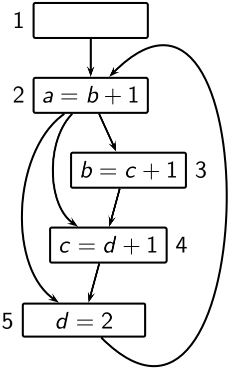
Raise	*Lower*	*Propagate*	*Negate*
			

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:



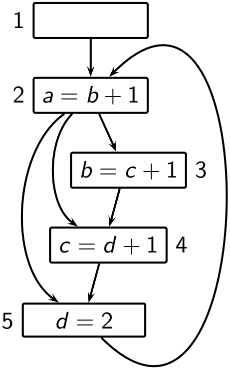
Larger Values of Loop Closure Bounds



The summary flow function for the loop is

$$f(\langle v_a, v_b, v_c, v_d \rangle) = \langle v_b + 1, v_c + 1, v_d + 1, 2 \rangle$$

Larger Values of Loop Closure Bounds

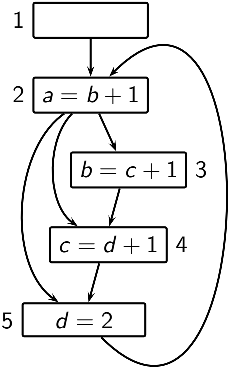


The summary flow function for the loop is

$$f(\langle v_a, v_b, v_c, v_d \rangle) = \langle v_b + 1, v_c + 1, v_d + 1, 2 \rangle$$

f is not 2-bounded because:

Larger Values of Loop Closure Bounds



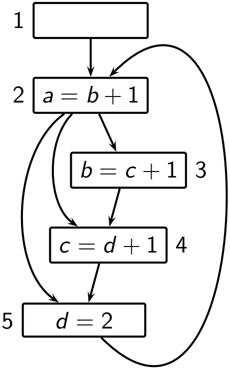
The summary flow function for the loop is

$$f(\langle v_a, v_b, v_c, v_d\rangle) = \langle v_b + 1, v_c + 1, v_d + 1, 2\rangle$$

f is not 2-bounded because:

$$f(\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}\rangle) \quad = \quad \langle \widehat{\top},\ \widehat{\top},\ \widehat{\top},\ \ 2\rangle$$

Larger Values of Loop Closure Bounds



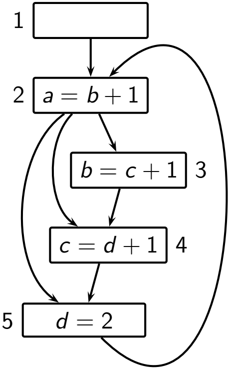
The summary flow function for the loop is

$$f(\langle v_a, v_b, v_c, v_d \rangle) = \langle v_b + 1, v_c + 1, v_d + 1, 2 \rangle$$

f is not 2-bounded because:

$$f(\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle) = \langle \widehat{\top},\ \widehat{\top},\ \widehat{\top},\ 2 \rangle$$
$$f^2(\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle) = \langle \widehat{\top},\ \widehat{\top},\ 3,\ 2 \rangle$$

Larger Values of Loop Closure Bounds



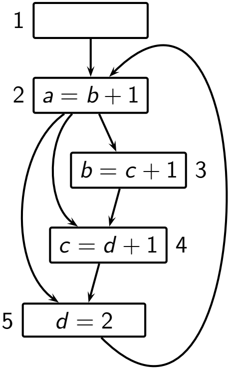
The summary flow function for the loop is

$$f(\langle v_a, v_b, v_c, v_d\rangle) = \langle v_b + 1, v_c + 1, v_d + 1, 2\rangle$$

f is not 2-bounded because:

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

Larger Values of Loop Closure Bounds



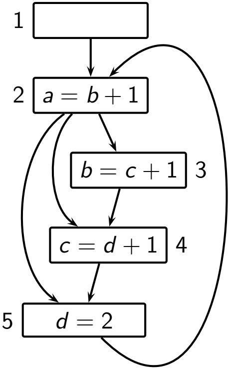
The summary flow function for the loop is

$$f(\langle v_a, v_b, v_c, v_d \rangle) = \langle v_b + 1, v_c + 1, v_d + 1, 2 \rangle$$

f is not 2-bounded because:

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

Larger Values of Loop Closure Bounds



The summary flow function for the loop is

$$f(\langle v_a, v_b, v_c, v_d \rangle) = \langle v_b + 1, v_c + 1, v_d + 1, 2 \rangle$$

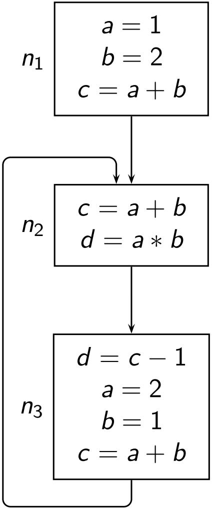
f is not 2-bounded because:

$$\begin{array}{rcl}
f(\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle) & = & \langle \widehat{\top}, \ \widehat{\top}, \ \widehat{\top}, \ 2 \rangle \\
f^2(\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle) & = & \langle \widehat{\top}, \ \widehat{\top}, \ 3, \ 2 \rangle \\
f^3(\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle) & = & \langle \widehat{\top}, \ 4, \ 3, \ 2 \rangle \\
f^4(\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle) & = & \langle 5, \ 4, \ 3, \ 2 \rangle \\
f^5(\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle) & = & \langle 5, \ 4, \ 3, \ 2 \rangle
\end{array}$$

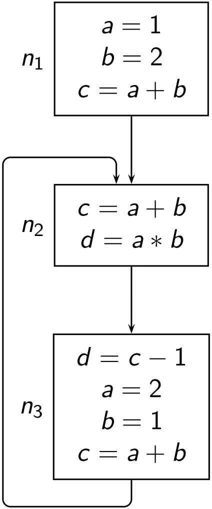
Part 3

Constant Propagation

Example of Constant Propagation



Example of Constant Propagation



MoP

$\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$

$\langle 1, 2, 3, \widehat{\top} \rangle$

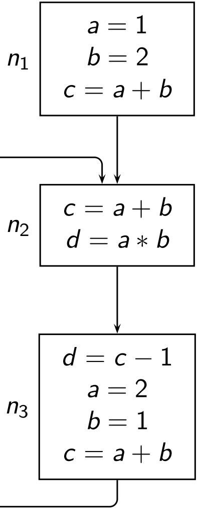
$\langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle$

$\langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle$

$\langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle$

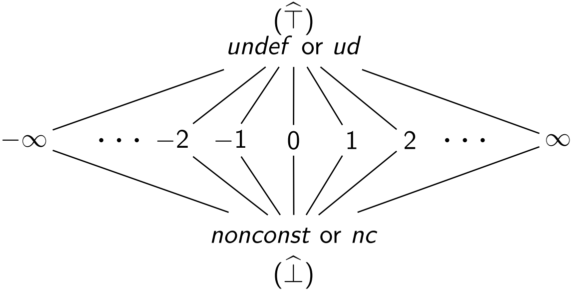
$\langle 2, 1, 3, 2 \rangle$

Example of Constant Propagation



MoP	MFP
$\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$	$\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$
$\langle 1, 2, 3, \widehat{\top} \rangle$	$\langle 1, 2, 3, \widehat{\top} \rangle$
$\langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle$	$\langle \widehat{\bot}, \widehat{\bot}, 3, \widehat{\bot} \rangle$
$\langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle$	$\langle \widehat{\bot}, \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle$
$\langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle$	$\langle \widehat{\bot}, \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle$
$\langle 2, 1, 3, 2 \rangle$	$\langle 2, 1, 3, \widehat{\bot} \rangle$

Component Lattice for Integer Constant Propagation



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

Overall Lattice for Integer Constant Propagation

- In_n/Out_n values are mappings $\mathbb{V}\text{ar} \to \widehat{L}$: $In_n, Out_n \in \mathbb{V}\text{ar} \to \widehat{L}$
- Overall lattice L is a set of mappings $\mathbb{V}\text{ar} \to \widehat{L}$: $L = \mathbb{V}\text{ar} \to \widehat{L}$
- $\sqcap$ and $\widehat{\sqcap}$ get defined by $\sqsubseteq$ and $\widehat{\sqsubseteq}$
    - Partial order is restricted to data flow values of the same variable
      Data flow values of different variables are incomparable

$$(x, v_1) \sqsubseteq (y, v_2) \Leftrightarrow x = y \wedge v_1 \widehat{\sqsubseteq} v_2$$

$$OR \qquad x \mapsto v_1 \sqsubseteq y \mapsto v_2 \Leftrightarrow x = y \wedge v_1 \widehat{\sqsubseteq} v_2$$

-
    - For meet operation, we assume that X is a total function
      Partial functions are made total by using $\widehat{\top}$value

$$X \sqcap Y = \left\{(x, v_1 \widehat{\sqcap} v_2) \mid (x, v_1) \in X, (x, v_2) \in Y\right\}$$

$$OR \qquad X \sqcap Y = \left\{x \mapsto v_1 \widehat{\sqcap} v_2 \mid x \mapsto v_1 \in X, x \mapsto v_2 \in Y\right\}$$

Notations for Mappings as Data Flow Values

Accessing and manipulating a mapping $X \subseteq A \rightarrow B$

- $X(a)$ denotes the image of $a \in A$
  $X(a) \in B$
- $X\,[a \mapsto v]$ changes the image of a in X to v

$$X\,[a \mapsto v] = (X - \{(a,u) \mid u \in B\}) \;\cup\; \{(a,v)\}$$

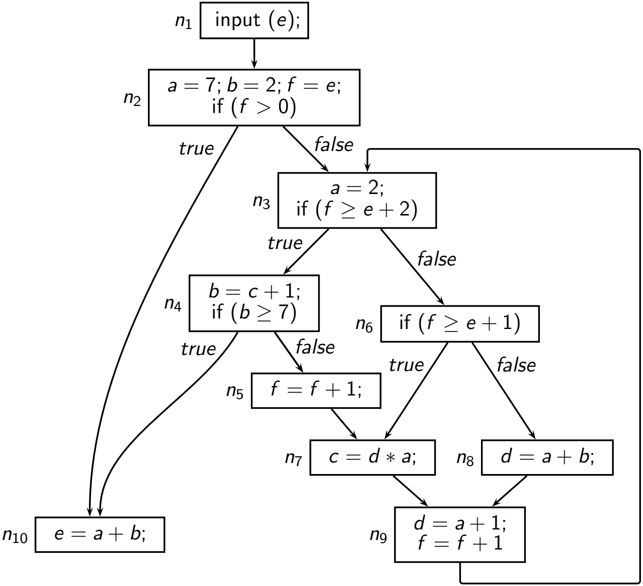
Defining Data Flow Equations for Constant Propagation

$$
\begin{aligned}
In_n &= \begin{cases} BI = \{\langle y, ud\rangle \mid y \in \mathbb{V}\text{ar}\} & n = Start \\ \displaystyle\bigsqcap_{p \in pred(n)} Out_p & \text{otherwise} \end{cases} \\
Out_n &= f_n(In_n)
\end{aligned}
$$

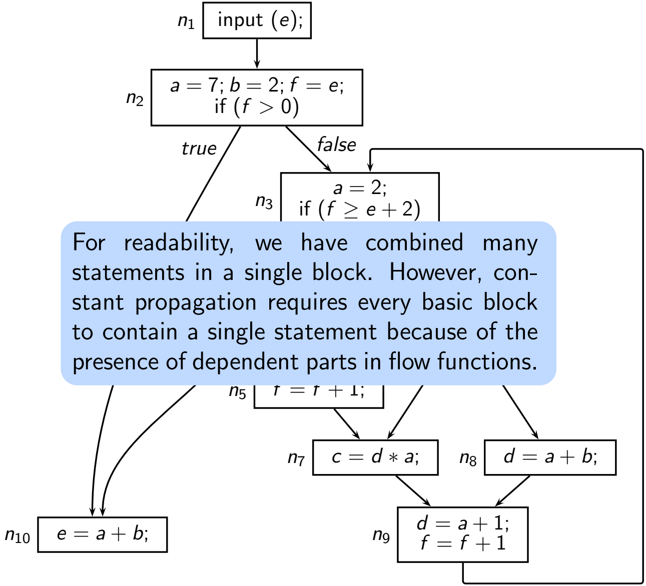
$$
f_n(X) = \begin{cases} X\,[y \mapsto c] & n \text{ is } y = c, y \in \mathbb{V}\text{ar}, c \in \mathbb{C}\text{onst} \\ X\,[y \mapsto nc] & n \text{ is } input(y), y \in var \\ X\,[y \mapsto X(z)] & n \text{ is } y = z, y \in \mathbb{V}\text{ar}, z \in \mathbb{V}\text{ar} \\ X\,[y \mapsto eval(e, X)] & n \text{ is } y = e, y \in \mathbb{V}\text{ar}, e \in \mathbb{E}\text{xpr} \\ X & \text{otherwise} \end{cases}
$$

$$
eval(e, X) = \begin{cases} nc & a \in Opd(e) \cap \mathbb{V}\text{ar}, X(a) = nc \\ ud & a \in Opd(e) \cap \mathbb{V}\text{ar}, X(a) = ud \\ -X(a) & e \text{ is } -a \\ X(a) \oplus X(b) & e \text{ is } a \oplus b \end{cases}
$$

Example Program for Constant Propagation



Example Program for Constant Propagation

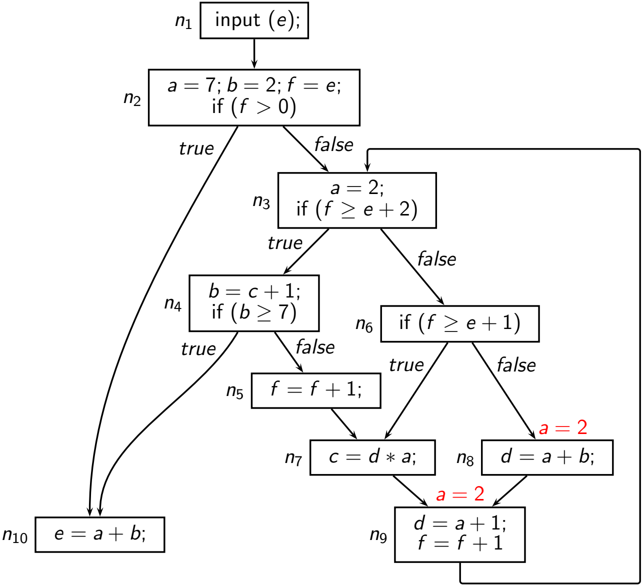


For readability, we have combined many statements in a single block. However, constant propagation requires every basic block to contain a single statement because of the presence of dependent parts in flow functions.

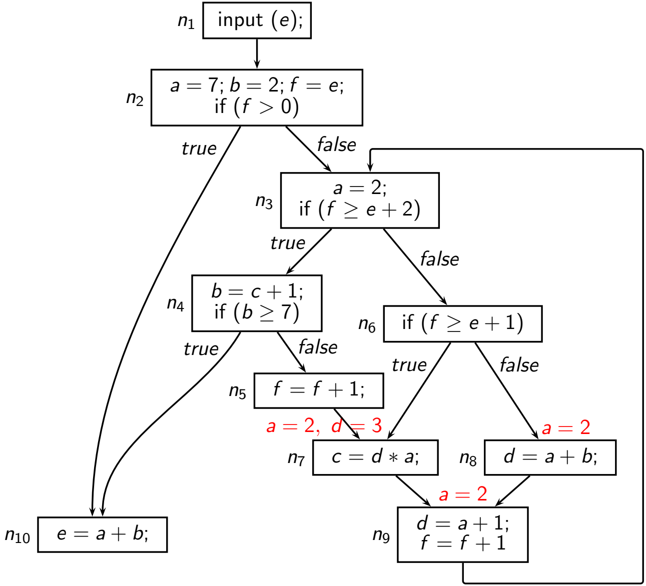
Result of Constant Propagation

	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}	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$	
$In_{n_{10}}$	$\widehat{\bot},2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$	
$Out_{n_{10}}$	$\widehat{\bot},2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$	

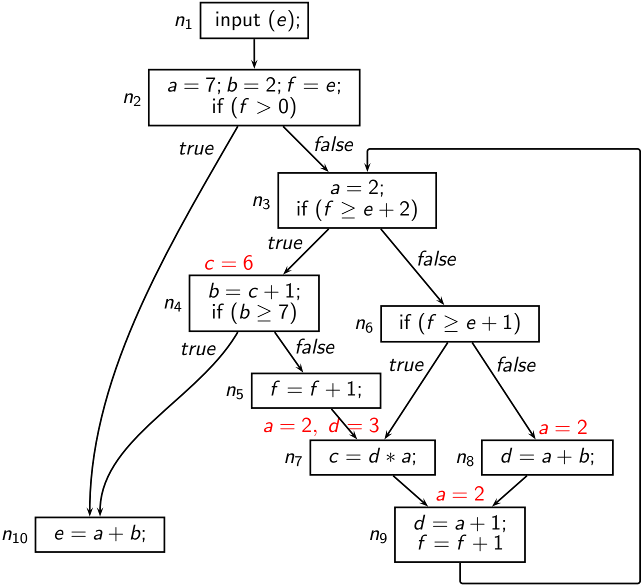
Result of Constant Propagation



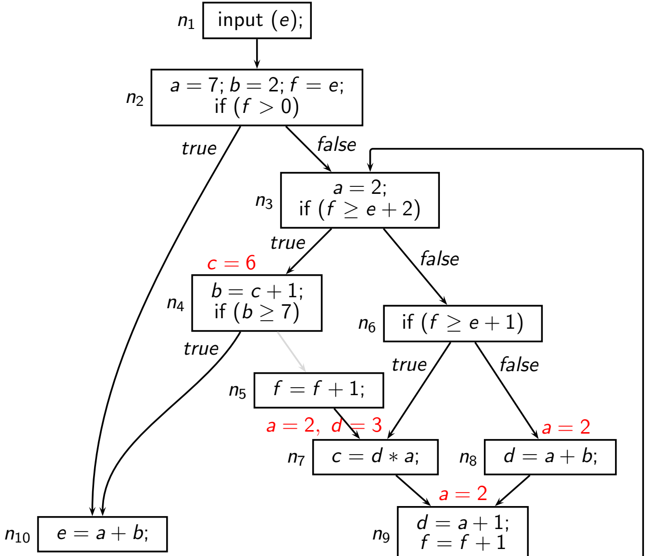
Result of Constant Propagation



Result of Constant Propagation



Result of Constant Propagation



Monotonicity of Constant Propagation

Proof obligation: $X_1 \sqsubseteq X_2 \;\Rightarrow\; f_n(X_1) \sqsubseteq f_n(X_2)$
where,

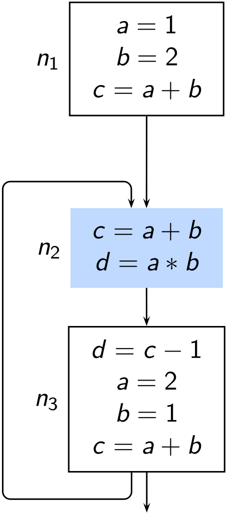
$$f_n(X) = \begin{cases} X\,[y \mapsto c] & n \text{ is } y = c, y \in \mathbb{V}\text{ar}, c \in \mathbb{C}\text{onst} & (C1) \\ X\,[y \mapsto nc] & n \text{ is } input(y), y \in var & (C2) \\ X\,[y \mapsto X(z)] & n \text{ is } y = z, y \in \mathbb{V}\text{ar}, z \in \mathbb{V}\text{ar} & (C3) \\ X\,[y \mapsto eval(e, X)] & n \text{ is } y = e, y \in \mathbb{V}\text{ar}, e \in \mathbb{E}\text{xpr} & (C4) \\ X & \text{otherwise} & (C5) \end{cases}$$

- The proof obligation trivially follows for cases C1, C2, and C5
- For case C3, $X_1 \sqsubseteq X_2 \;\Rightarrow\; X_1(z) \sqsubseteq X_2(z)$
- For case C4, it requires showing

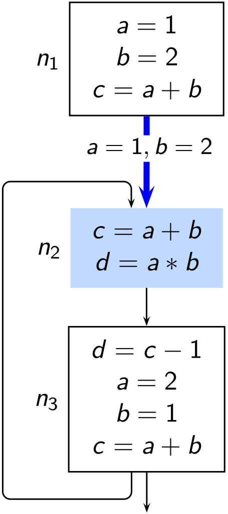
  $X_1 \sqsubseteq X_2 \;\Rightarrow\; eval(e, X_1) \sqsubseteq eval(e, X_2)$

  which follows from the definition of $eval(e, X)$

Non-Distributivity of Constant Propagation

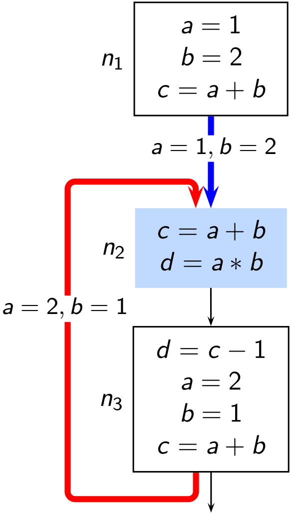


Non-Distributivity of Constant Propagation



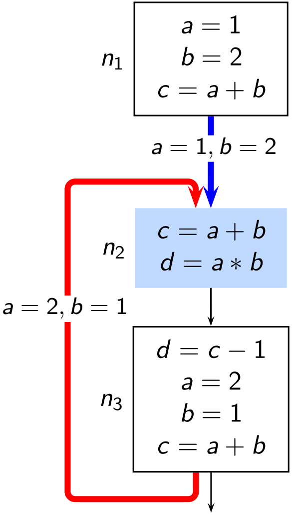
- $x = \langle 1, 2, 3, ud \rangle$ (Along $Out_{n_1} \rightarrow In_{n_2}$)

Non-Distributivity of Constant Propagation



- $x = \langle 1, 2, 3, ud \rangle$ (Along $Out_{n_1} \rightarrow In_{n_2}$)
- $y = \langle 2, 1, 3, 2 \rangle$ (Along $Out_{n_3} \rightarrow In_{n_2}$)

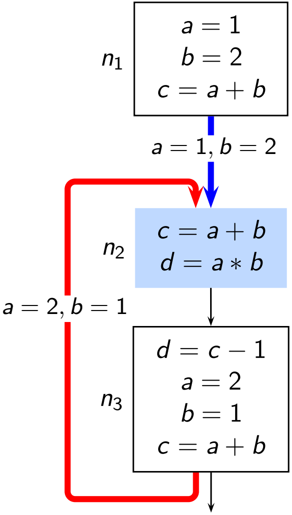
Non-Distributivity of Constant Propagation



- $x = \langle 1, 2, 3, ud \rangle$ (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 for block n_2 before merging

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

Non-Distributivity of Constant Propagation



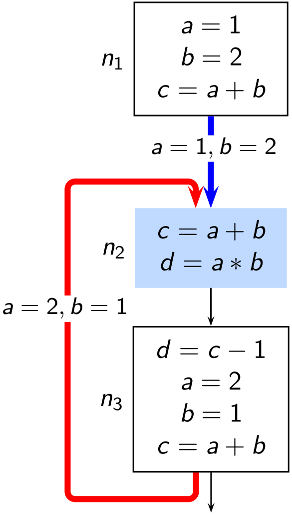
- $x = \langle 1, 2, 3, ud \rangle$ (Along $Out_{n_1} \rightarrow In_{n_2}$)
- $y = \langle 2, 1, 3, 2 \rangle$ (Along $Out_{n_3} \rightarrow In_{n_2}$)
- Function application for block n_2 before merging

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

- Function application for block n_2 after merging

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

Non-Distributivity of Constant Propagation



- $x = \langle 1, 2, 3, ud \rangle$ (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 for block n_2 before merging

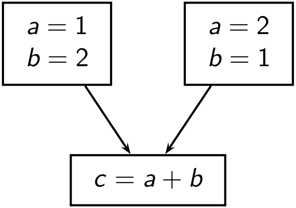
$$\begin{aligned} f(x) \sqcap f(y) &= f(\langle 1, 2, 3, ud \rangle) \sqcap f(\langle 2, 1, 3, 2 \rangle) \\ &= \langle 1, 2, 3, 2 \rangle \sqcap \langle 2, 1, 3, 2 \rangle \\ &= \langle \widehat{\perp}, \widehat{\perp}, 3, 2 \rangle \end{aligned}$$

- Function application for block n_2 after merging

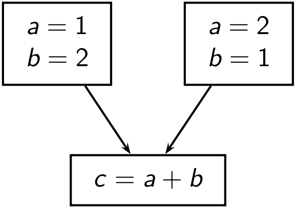
$$\begin{aligned} f(x \sqcap y) &= f(\langle 1, 2, 3, ud \rangle \sqcap \langle 2, 1, 3, 2 \rangle) \\ &= f(\langle \widehat{\perp}, \widehat{\perp}, 3, 2 \rangle) \\ &= \langle \widehat{\perp}, \widehat{\perp}, \widehat{\perp}, \widehat{\perp} \rangle \end{aligned}$$

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

Why is Constant Propagation Non-Distributive?



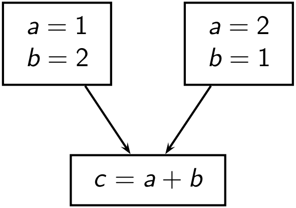
Why is Constant Propagation Non-Distributive?



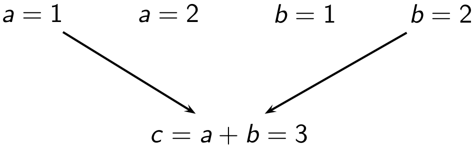
Possible combinations due to merging

$a = 1$ $\quad$ $a = 2$ $\quad$ $b = 1$ $\quad$ $b = 2$

Why is Constant Propagation Non-Distributive?

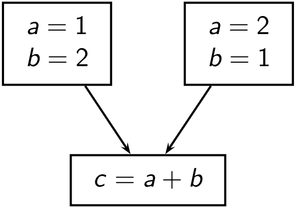


Possible combinations due to merging

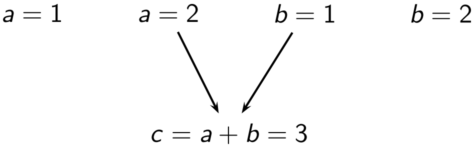


- Correct combination.

Why is Constant Propagation Non-Distributive?

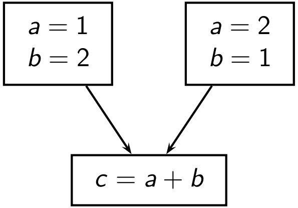


Possible combinations due to merging

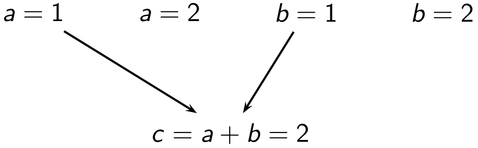


- Correct combination.

Why is Constant Propagation Non-Distributive?

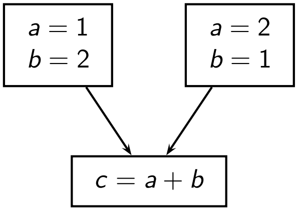


Possible combinations due to merging

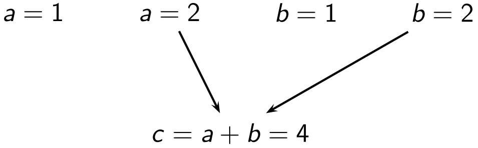


- Wrong combination.
- Mutually exclusive information.
- No execution path along which this information holds.

Why is Constant Propagation Non-Distributive?



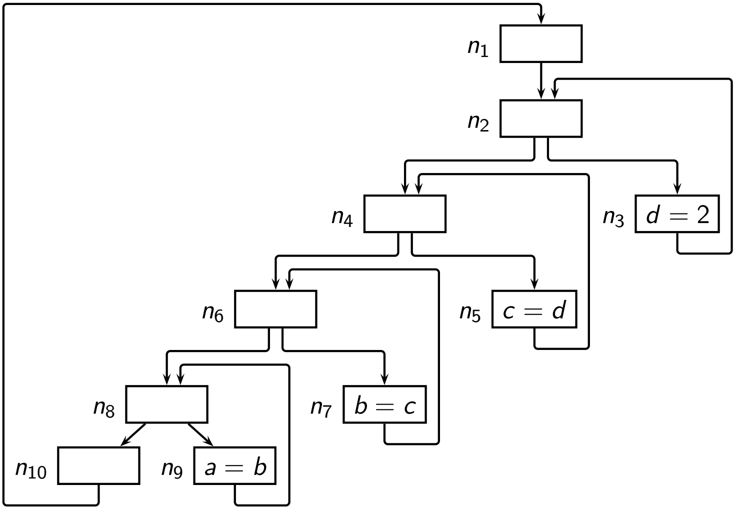
Possible combinations due to merging



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

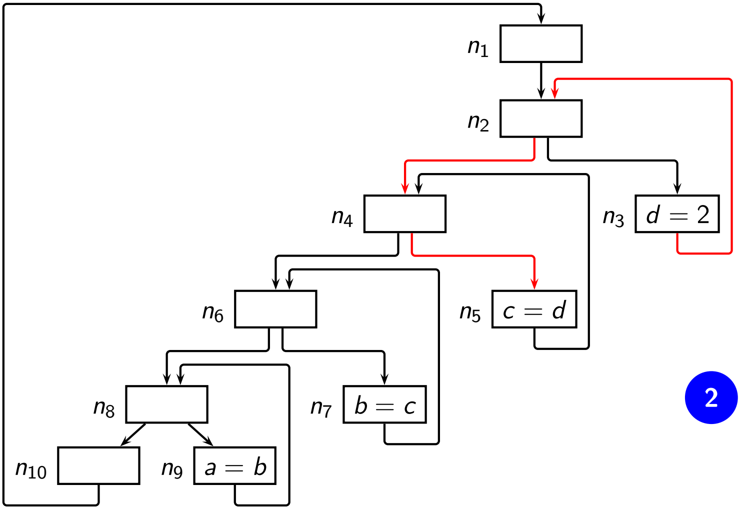
Tutorial Problem on Constant Propagation

How many iterations do we need?



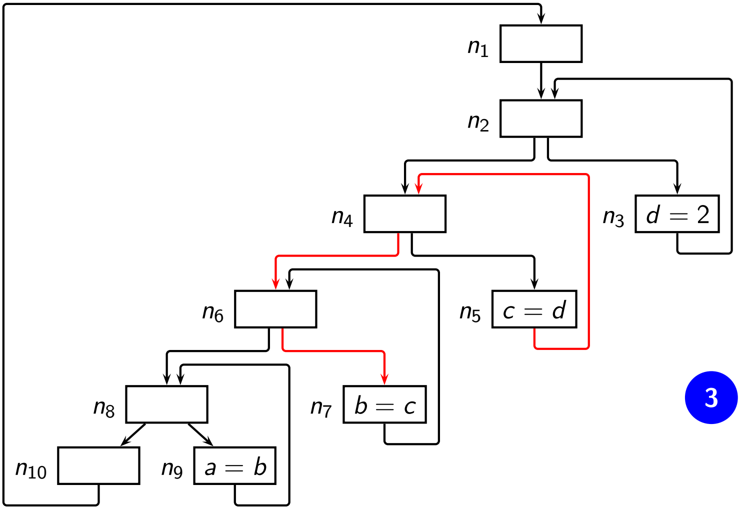
Tutorial Problem on Constant Propagation

How many iterations do we need?



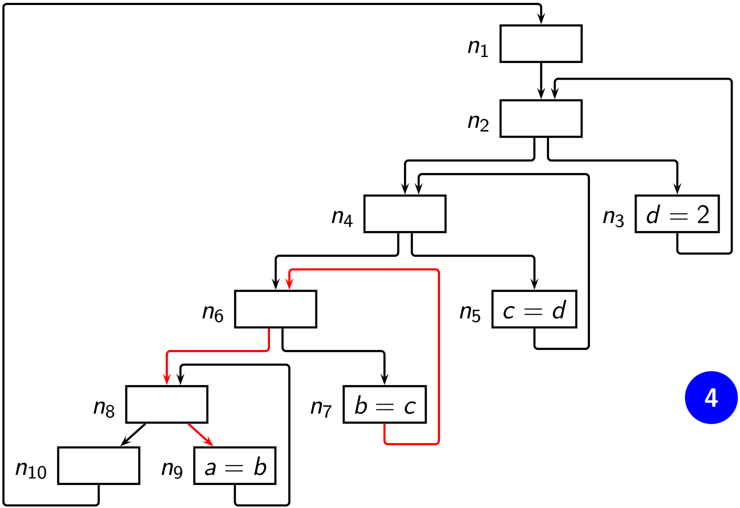
Tutorial Problem on Constant Propagation

How many iterations do we need?



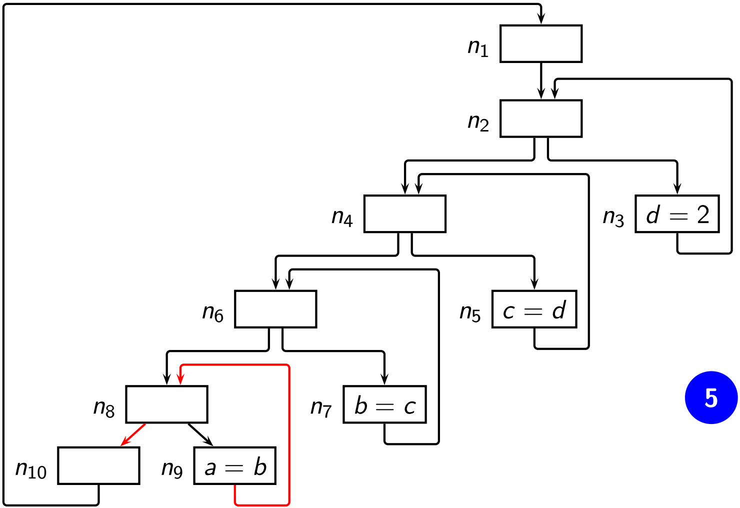
Tutorial Problem on Constant Propagation

How many iterations do we need?



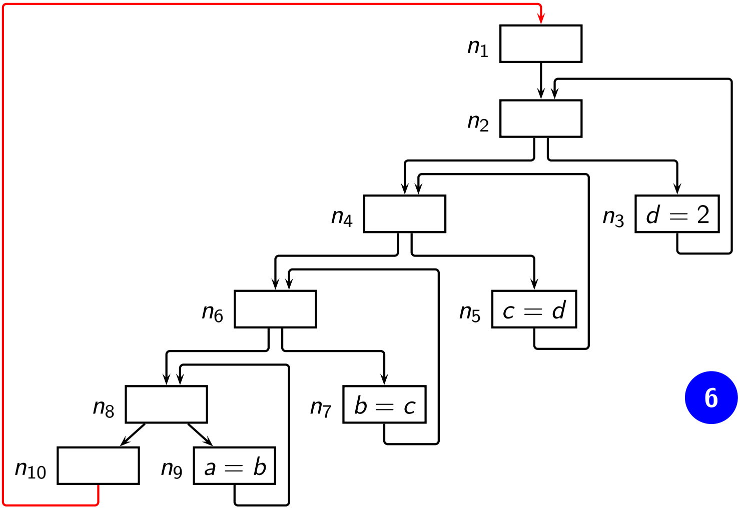
Tutorial Problem on Constant Propagation

How many iterations do we need?



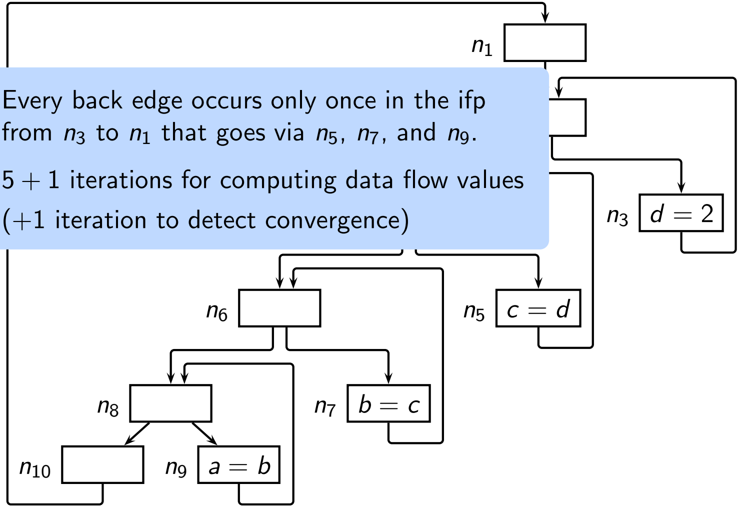
Tutorial Problem on Constant Propagation

How many iterations do we need?



Tutorial Problem on Constant Propagation

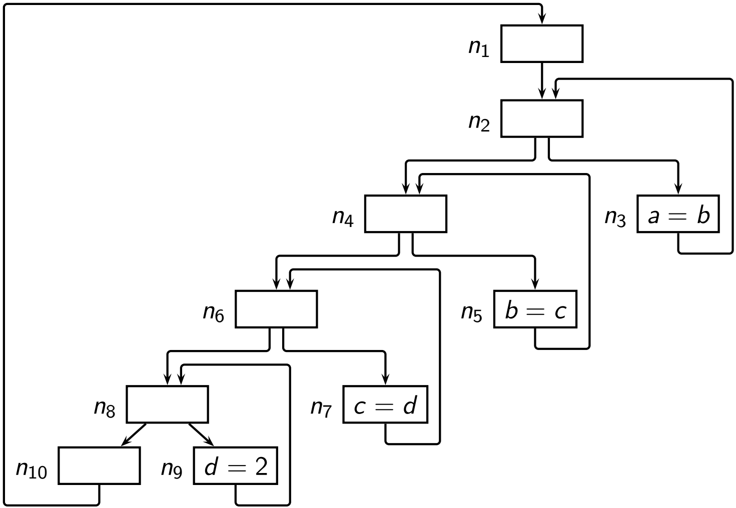
How many iterations do we need?



- Every back edge occurs only once in the ifp from n_3 to n_1 that goes via n_5, n_7, and n_9.
- $5+1$ iterations for computing data flow values ($+1$ iteration to detect convergence)

Tutorial Problem on Constant Propagation

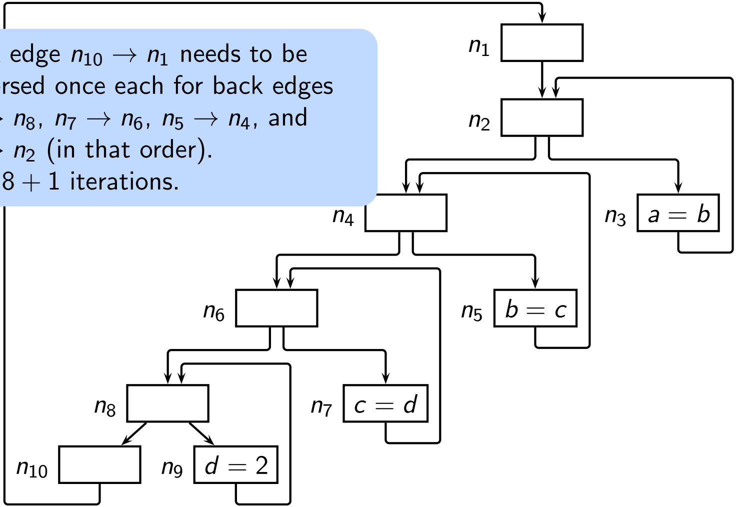
And now how many iterations do we need?



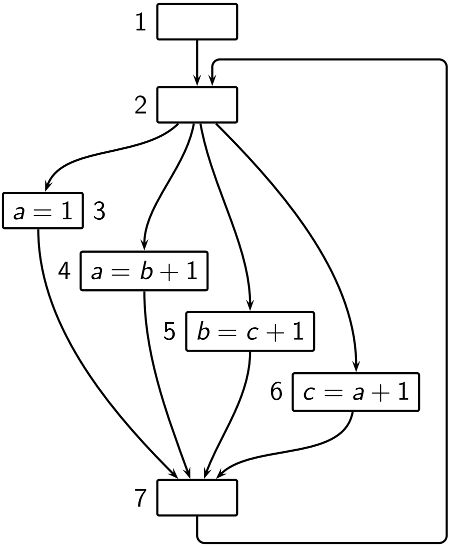
Tutorial Problem on Constant Propagation

And now how many iterations do we need?

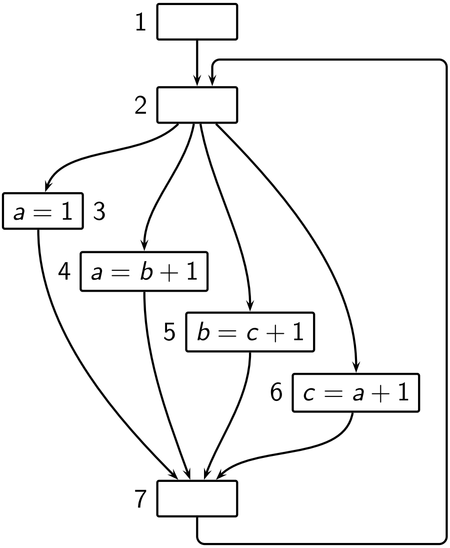
Back edge $n_{10} \rightarrow n_1$ needs to be traversed once each for back edges $n_9 \rightarrow n_8$, $n_7 \rightarrow n_6$, $n_5 \rightarrow n_4$, and $n_3 \rightarrow n_2$ (in that order).
$\Rightarrow \quad 8 + 1$ iterations.



Boundedness of Constant Propagation



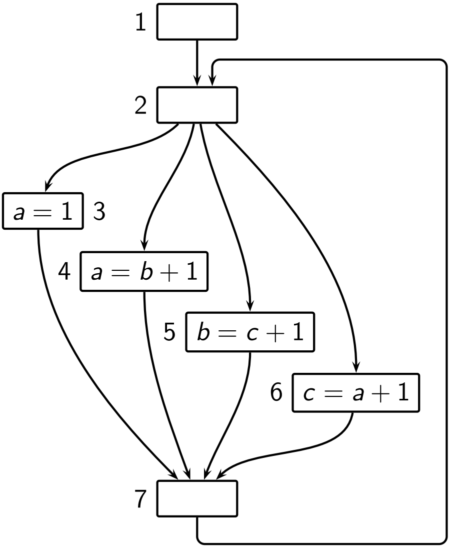
Boundedness of Constant Propagation



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

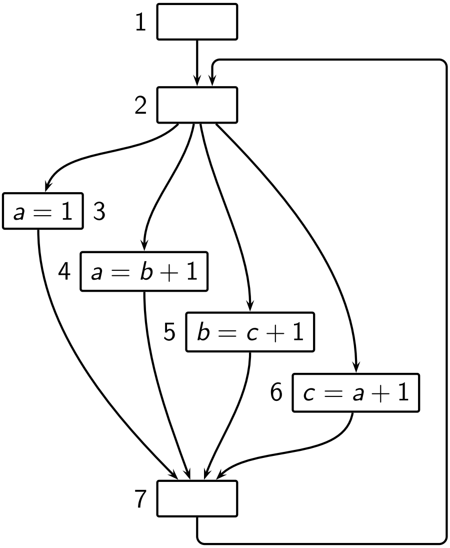


Summary flow function:
(data flow value at node 7)

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

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

Boundedness of Constant Propagation

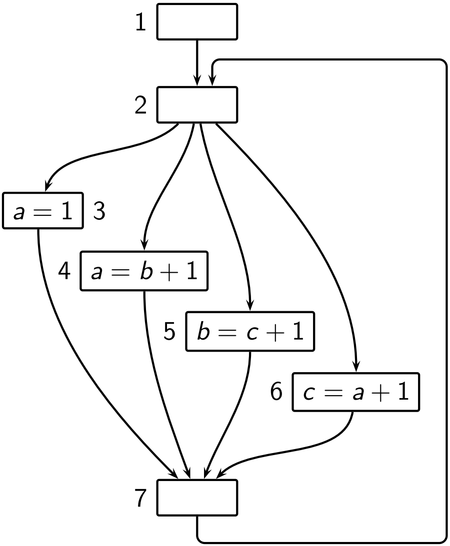


Summary flow function:
(data flow value at node 7)

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

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

Boundedness of Constant Propagation



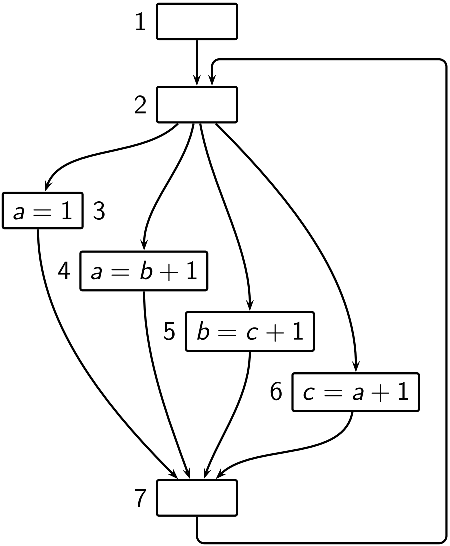
Summary flow function:

(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

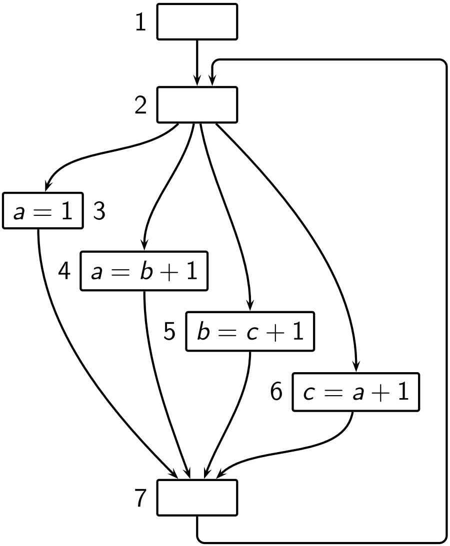


Summary flow function:
(data flow value at node 7)

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

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

Boundedness of Constant Propagation

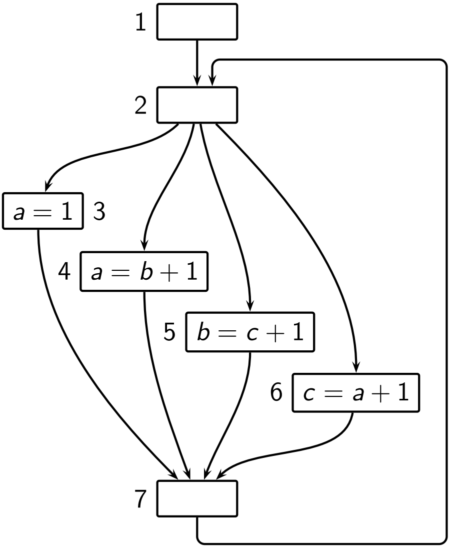


Summary flow function:
(data flow value at node 7)

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

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

Boundedness of Constant Propagation

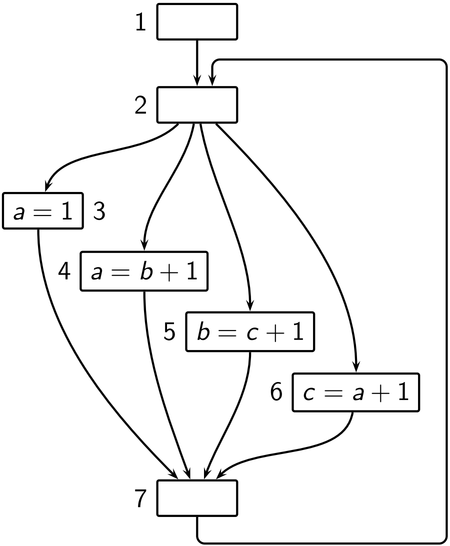


Summary flow function:
(data flow value at node 7)

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

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

Boundedness of Constant Propagation

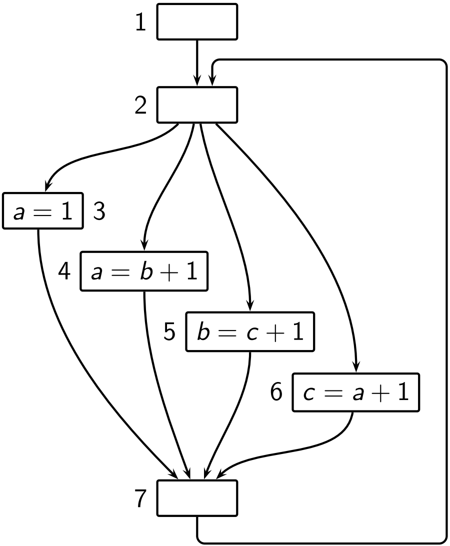


Summary flow function:
(data flow value at node 7)

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

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

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:

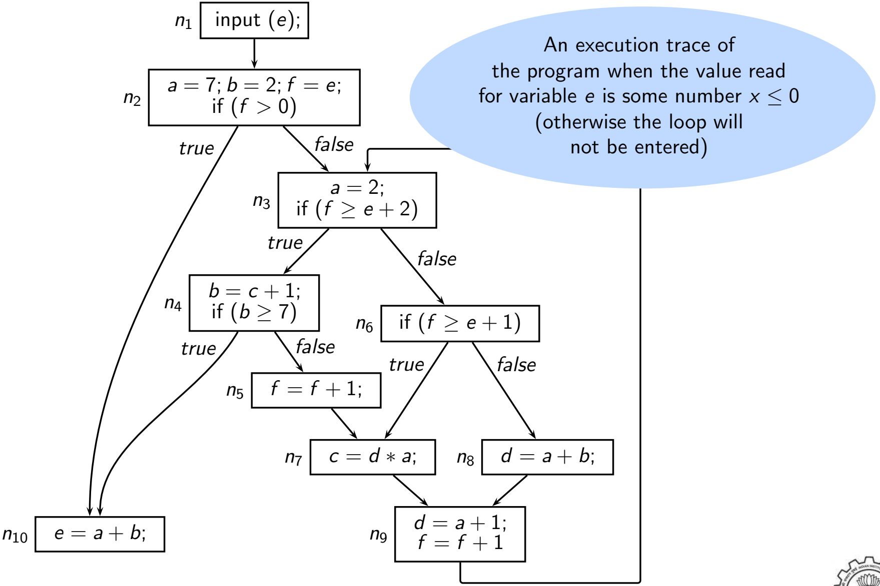
- The data flow value of every variable could change twice
- In the worst case, only one change may happen in every step of a function application
- Maximum number of steps: $2 \times |\mathbb{V}\text{ar}|$

Boundedness of Constant Propagation

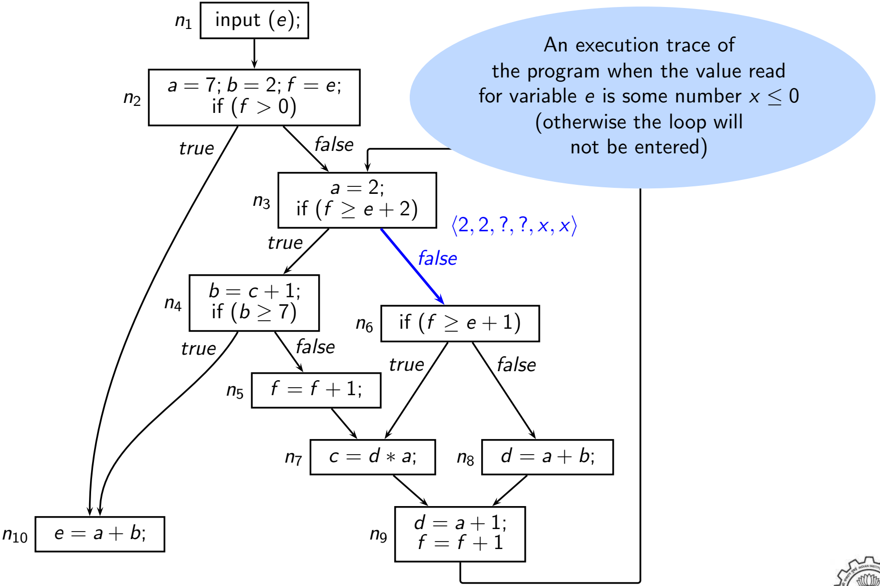
The moral of the story:

- 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 $(2 \times |\mathbb{V}\text{ar}|) + 1$

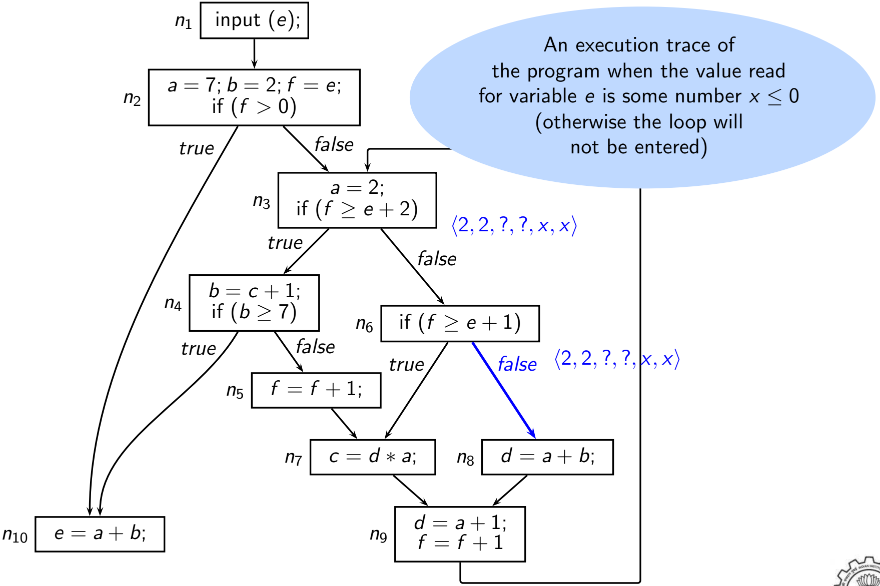
Conditional Constant Propagation



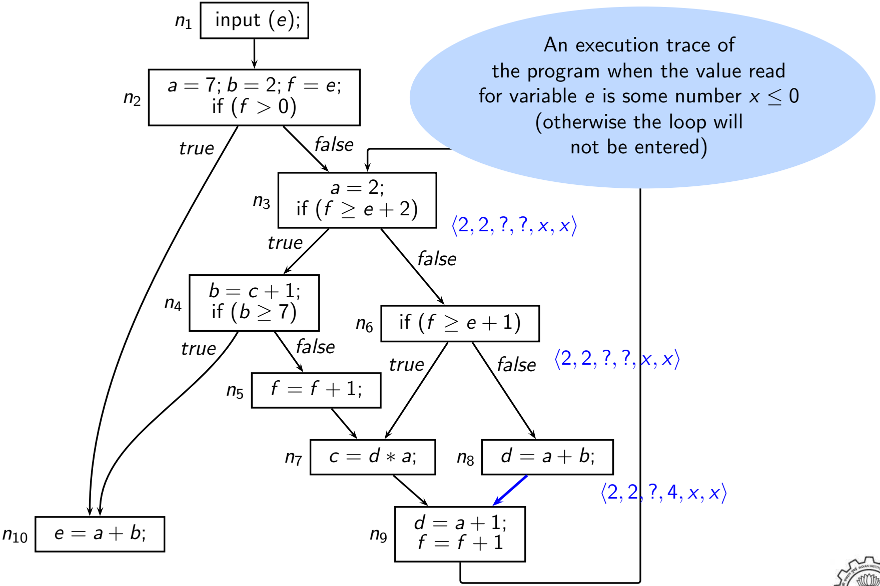
Conditional Constant Propagation



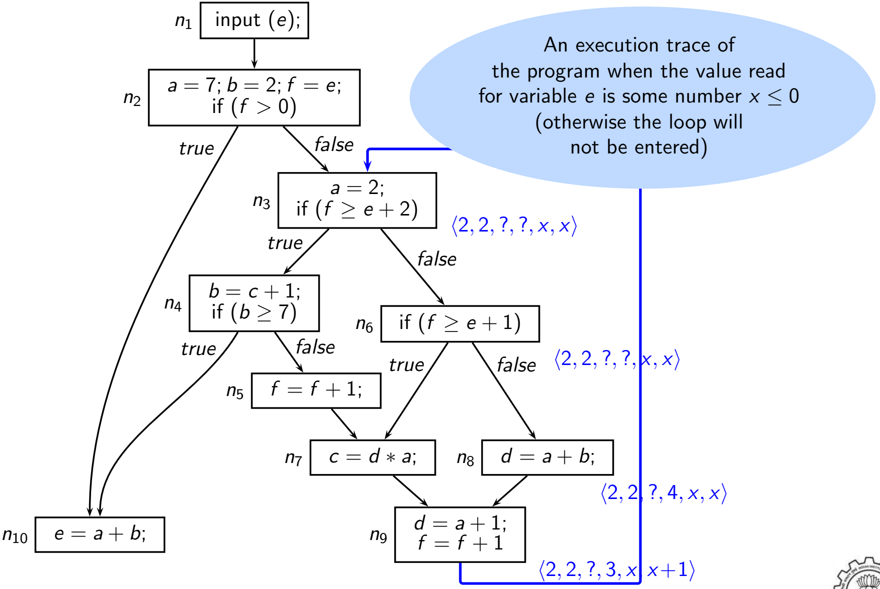
Conditional Constant Propagation



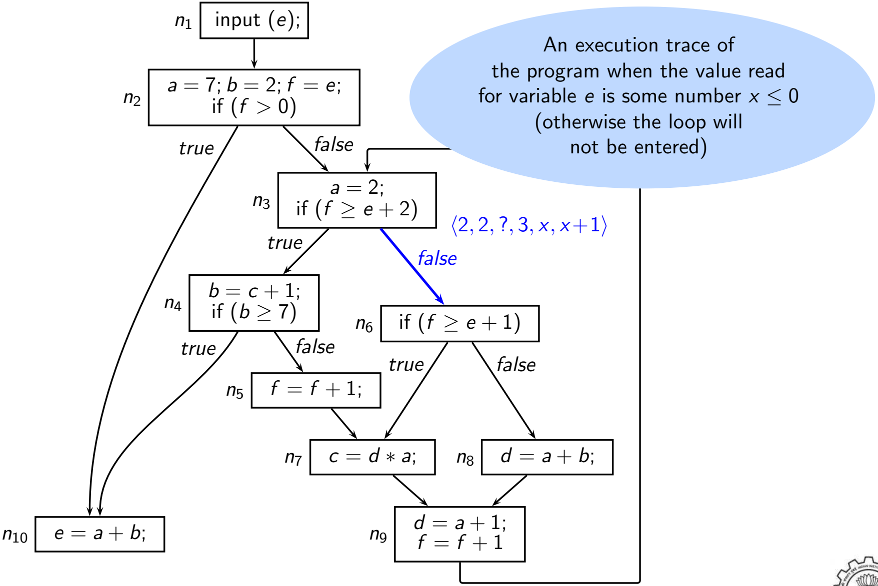
Conditional Constant Propagation



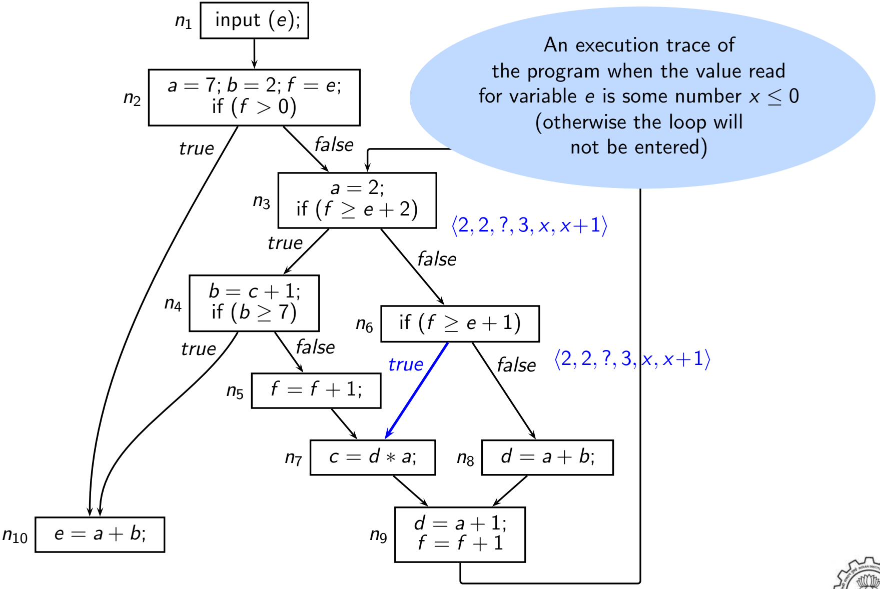
Conditional Constant Propagation



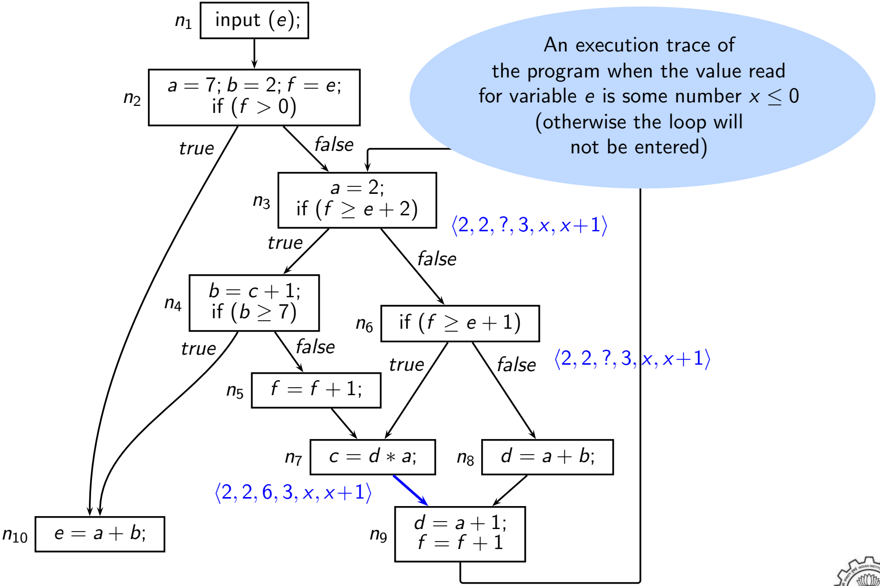
Conditional Constant Propagation



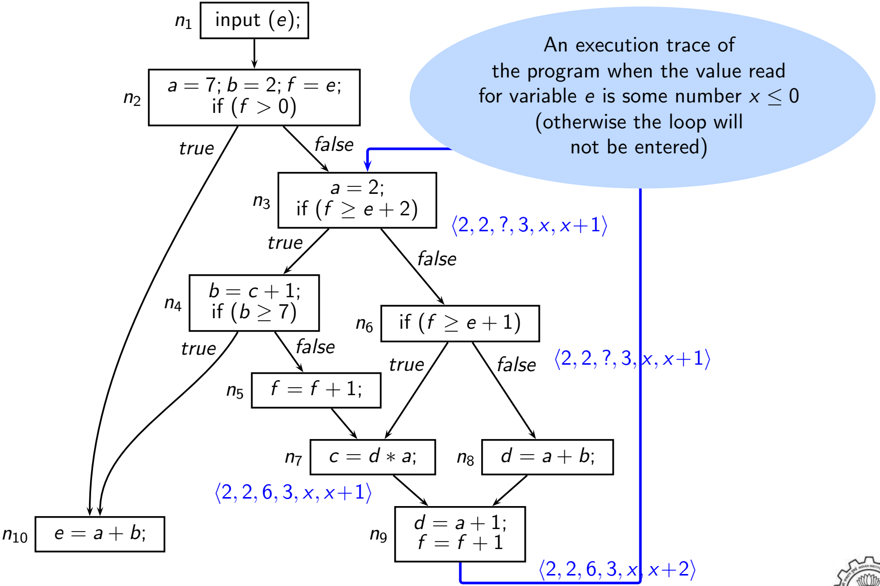
Conditional Constant Propagation



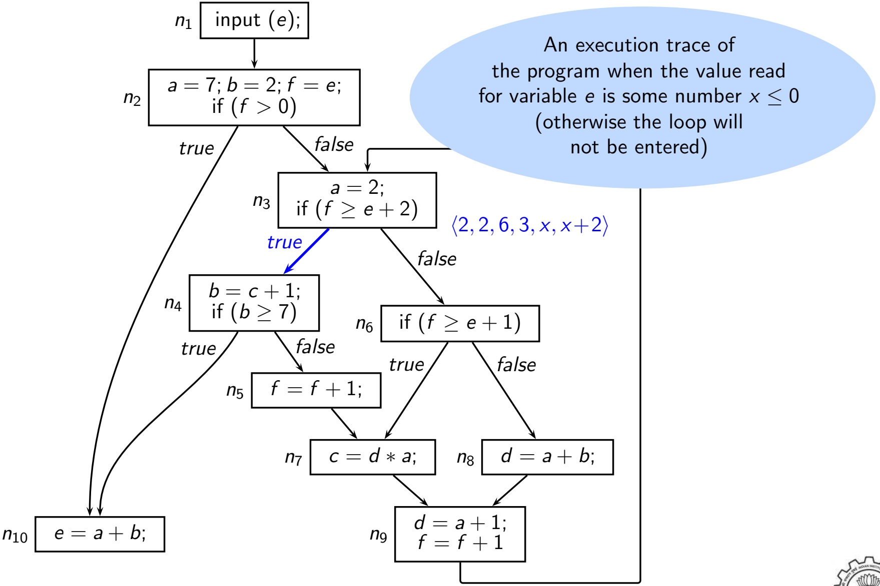
Conditional Constant Propagation



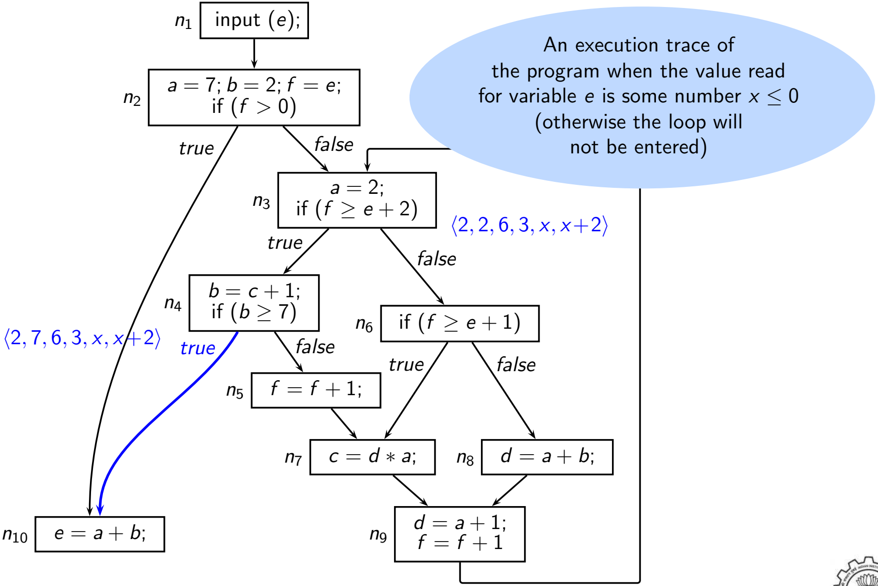
Conditional Constant Propagation



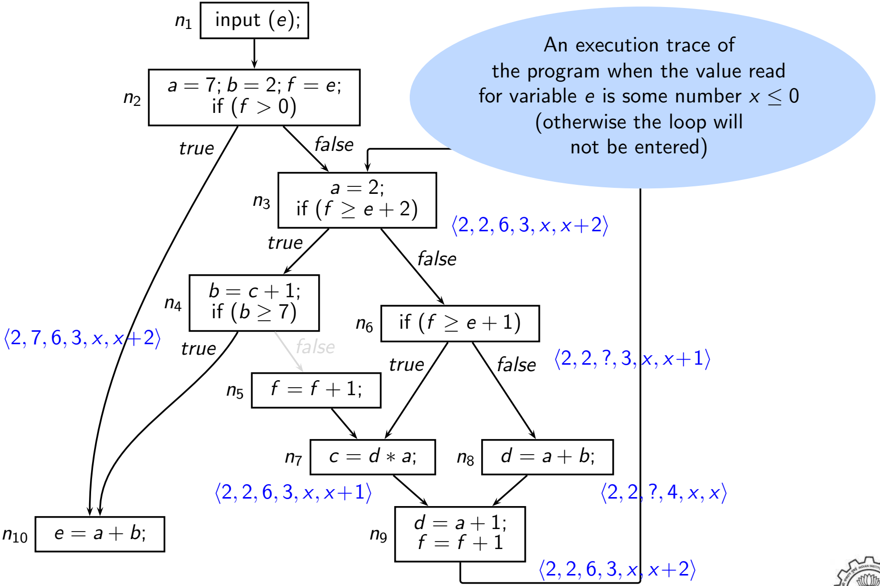
Conditional Constant Propagation



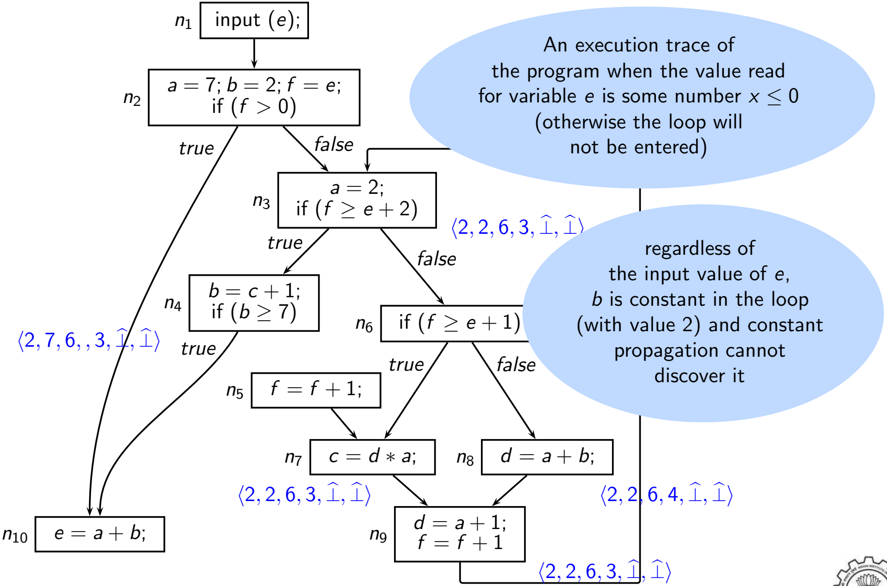
Conditional Constant Propagation



Conditional Constant Propagation



Conditional Constant Propagation



Lattice for Conditional Constant Propagation

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

- 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

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

Data Flow Equations for Conditional Constant Propagation

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

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

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

- $label(m \rightarrow n)$ is T or F if edge $m \rightarrow n$ is a conditional branch
  Otherwise $label(m \rightarrow n)$ is T
- $evalCond(m, X)$ evaluates the condition in block m using the data flow values in X

Compile Time Evaluation of Conditions using the Data Flow Values

$evalCond(m, X)$	
$\{T, F\}$	Block m does not have a condition, or some variable in the condition is $\widehat{\bot}$ in X
$\{\}$	No variable in the condition in block m is $\widehat{\bot}$ in X, but some variable is $\widehat{\top}$ in X
$\{T\}$	The condition in block m evaluates to T with the data flow values in X
$\{F\}$	The condition in block m evaluates to F with the data flow values in X

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}	$R, \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\top} \rangle$		
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}	$N, \top = \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$		
Out_{n_5}	$N, \top = \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$		
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}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$	
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}	$R, \langle 2, 2, \widehat{\top}, 4, \widehat{\bot}, \widehat{\bot} \rangle$		$R, \langle 2, 2, 6, 4, \widehat{\bot}, \widehat{\bot} \rangle$
In_{n_9}	$R, \langle 2, 2, \widehat{\top}, 4, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle$	
Out_{n_9}	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$	
$In_{n_{10}}$	$R, \langle 7, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, \widehat{\bot}, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
$Out_{n_{10}}$	$R, \langle 7, 2, \widehat{\top}, \widehat{\top}, 9, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, \widehat{\bot}, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$

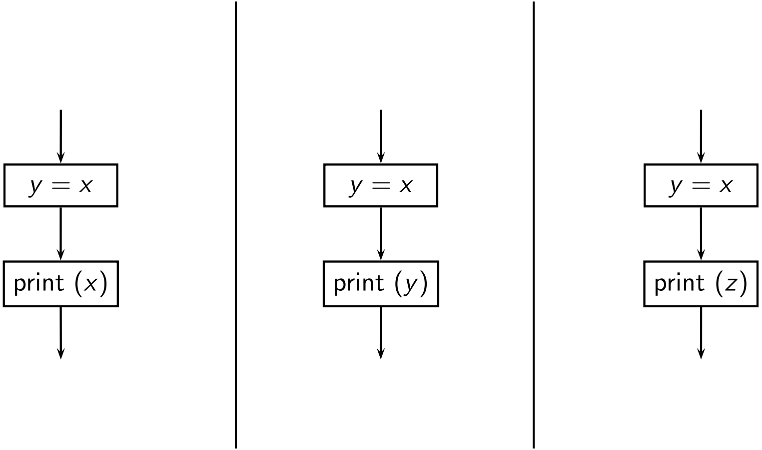
Part 4

Strongly Live Variables Analysis

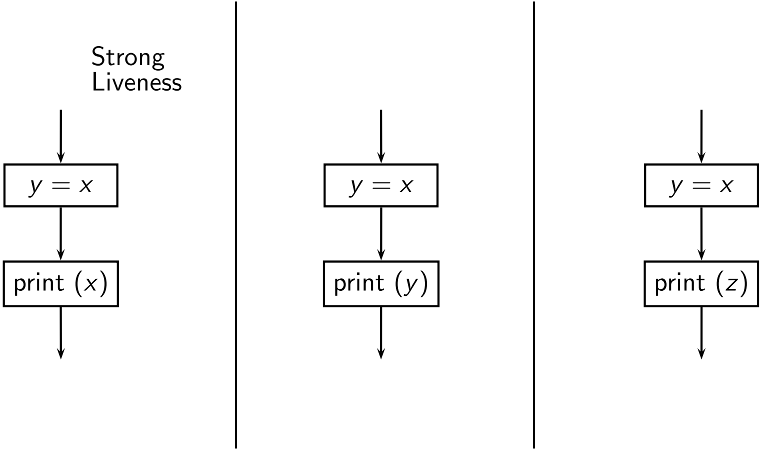
Strongly Live Variables Analysis

- A variable is strongly live if
  - it is used in a statement other than assignment statement, or (same as simple liveness)
  - it is used in an assignment statement defining a variable that is strongly live
    (different from simple liveness)
- Killing: An assignment statement, an input statement, or BI
  (this is same as killing in simple liveness)
- Generation: A direct use or a use for defining values that are strongly live
  (this is different from generation in simple liveness)

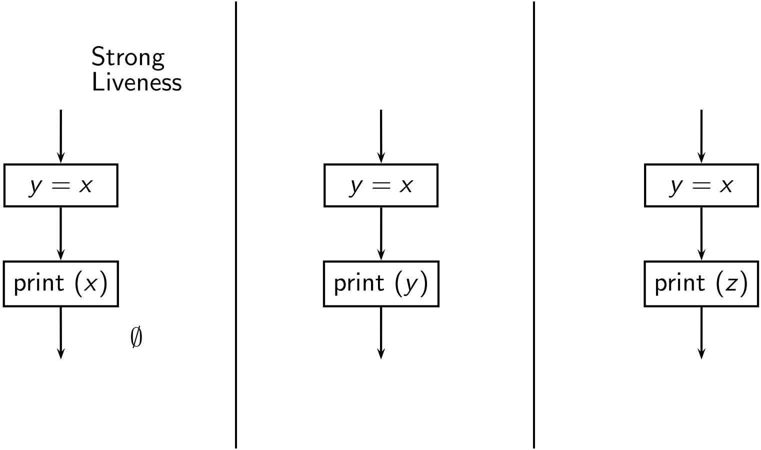
Understanding Strong Liveness



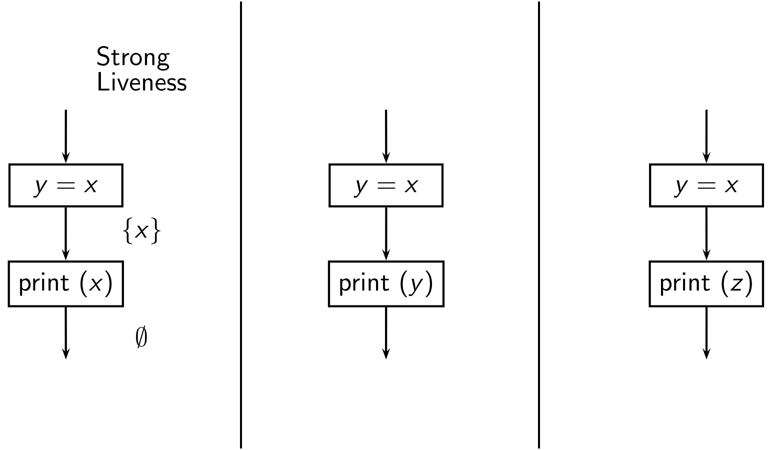
Understanding Strong Liveness



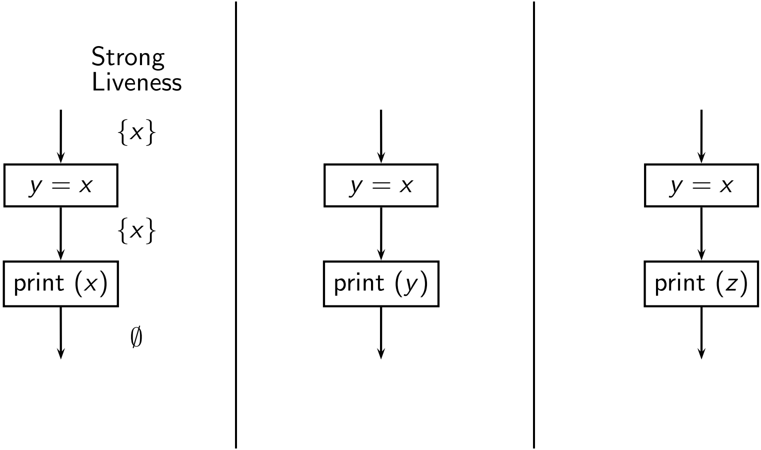
Understanding Strong Liveness



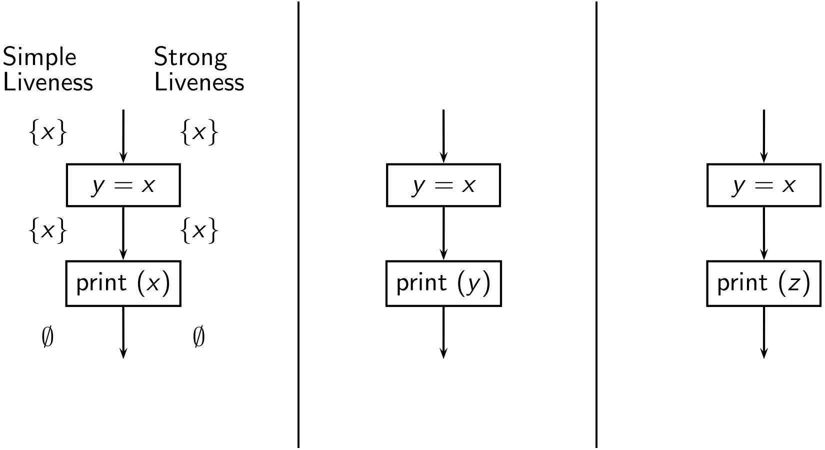
Understanding Strong Liveness



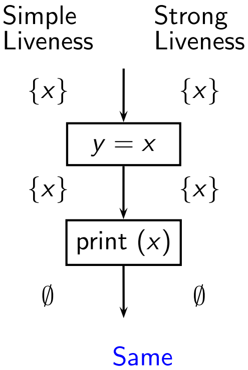
Understanding Strong Liveness

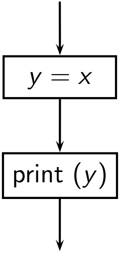


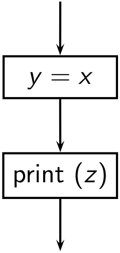
Understanding Strong Liveness



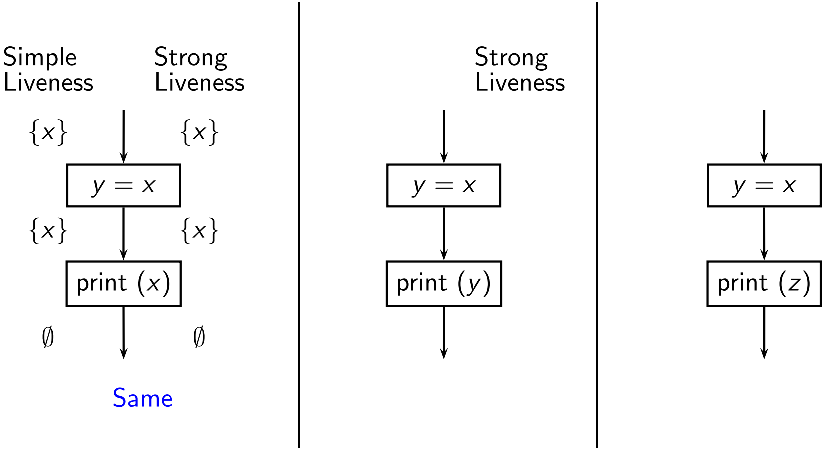
Understanding Strong Liveness



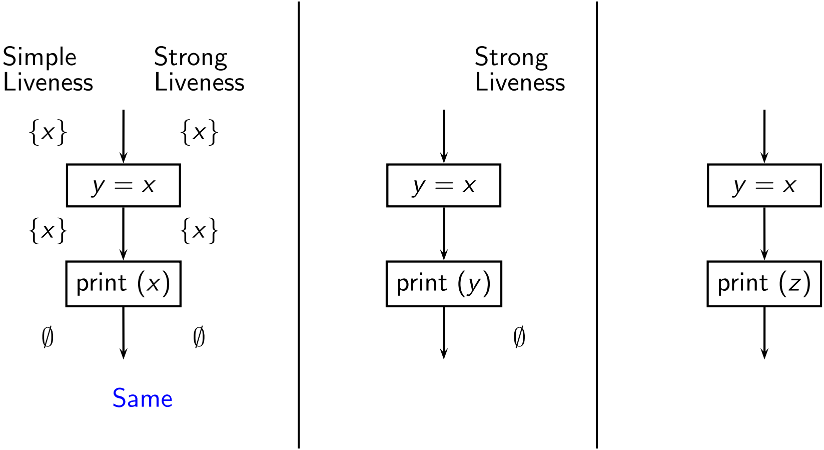




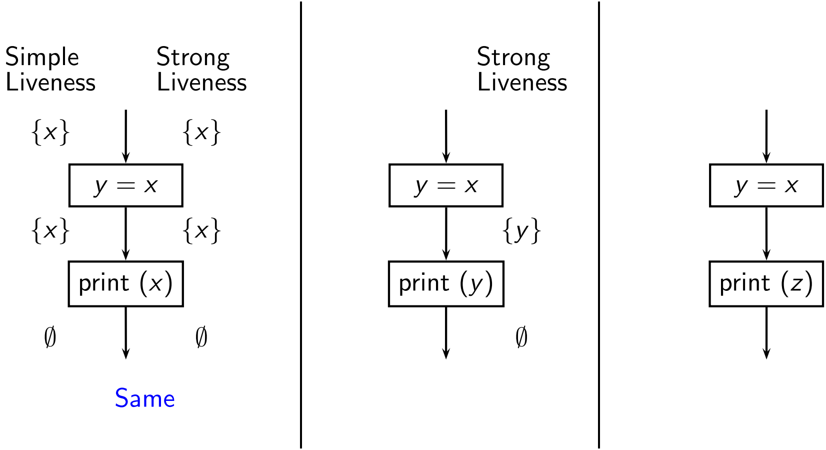
Understanding Strong Liveness



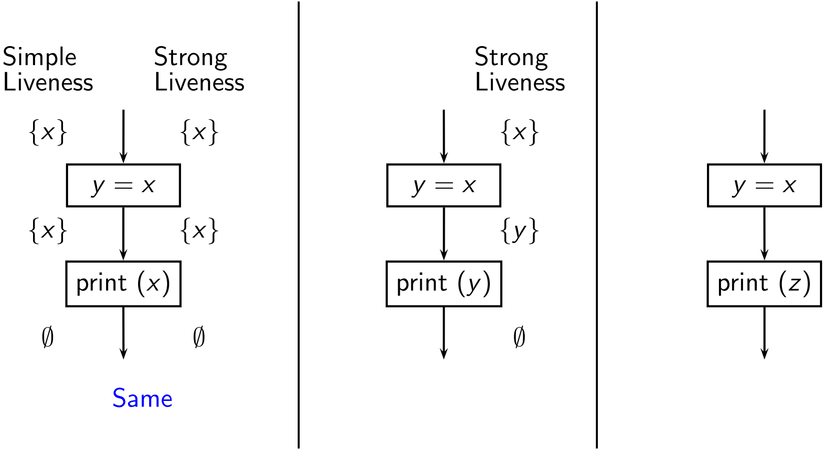
Understanding Strong Liveness



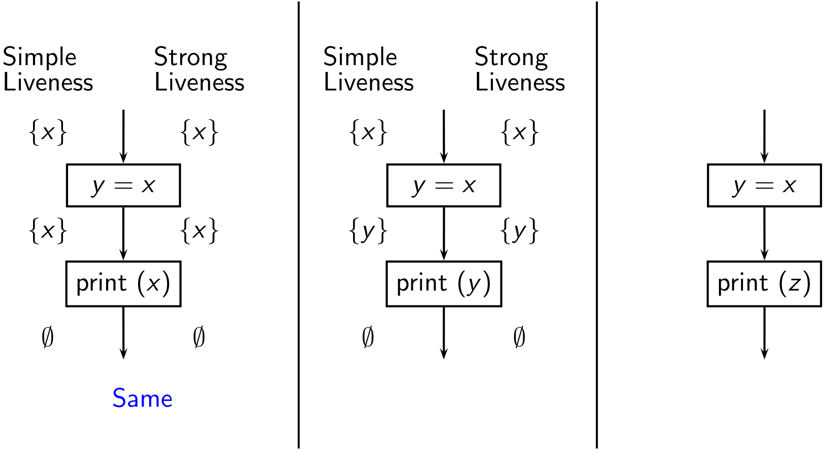
Understanding Strong Liveness



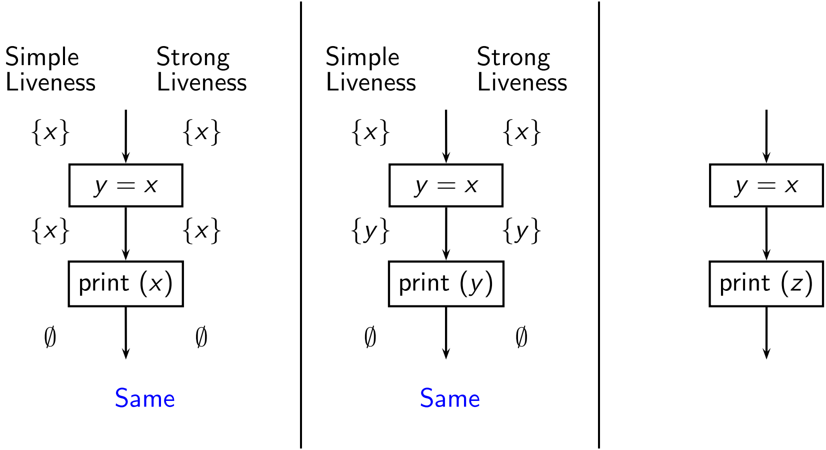
Understanding Strong Liveness



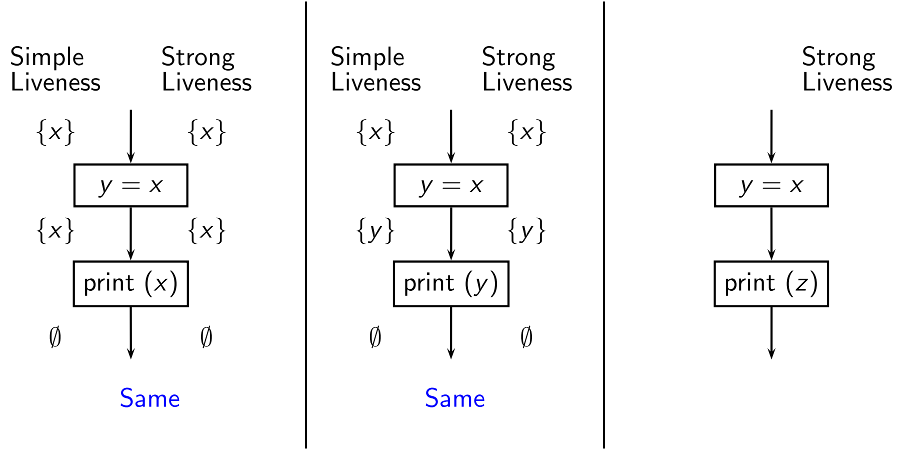
Understanding Strong Liveness



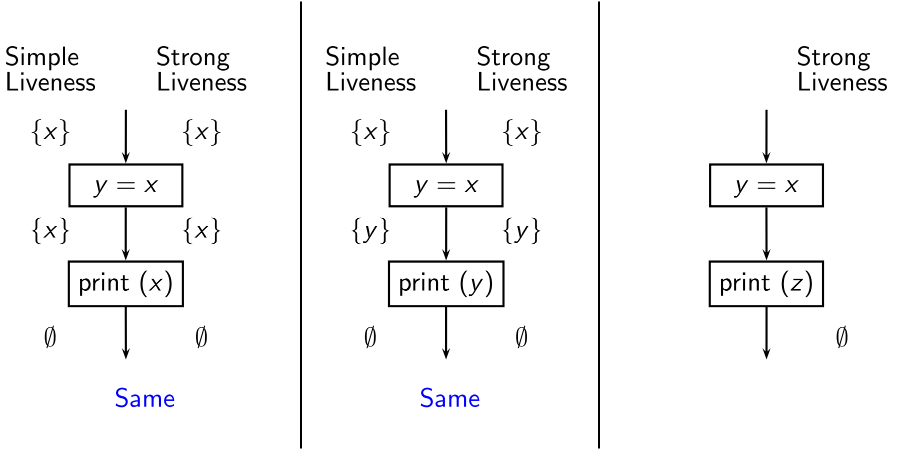
Understanding Strong Liveness



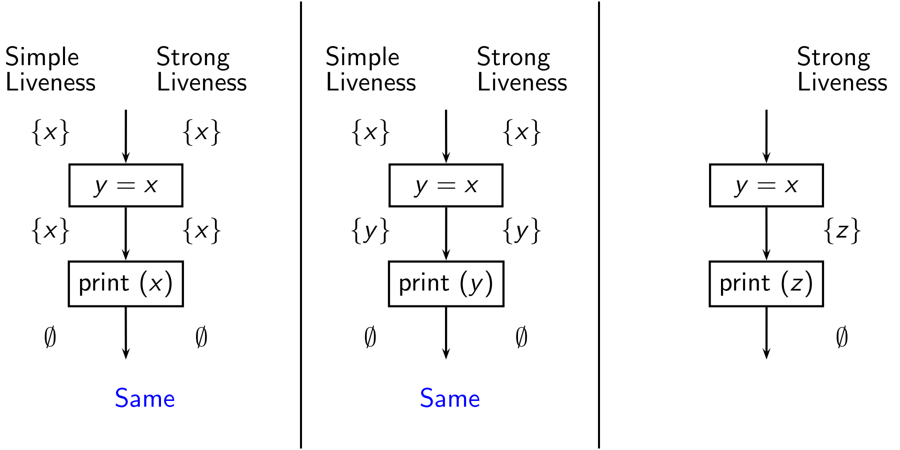
Understanding Strong Liveness



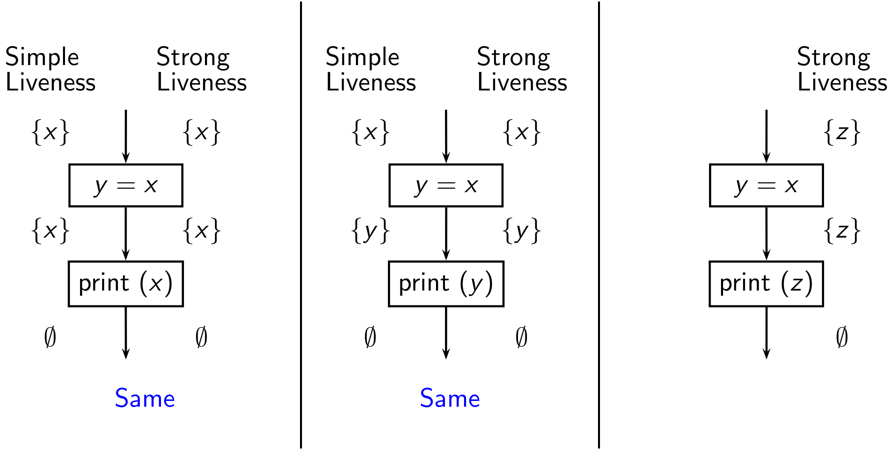
Understanding Strong Liveness



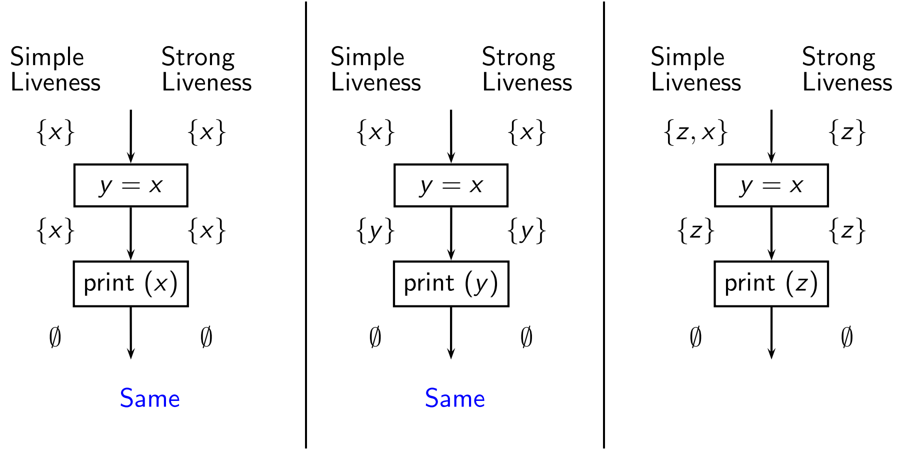
Understanding Strong Liveness



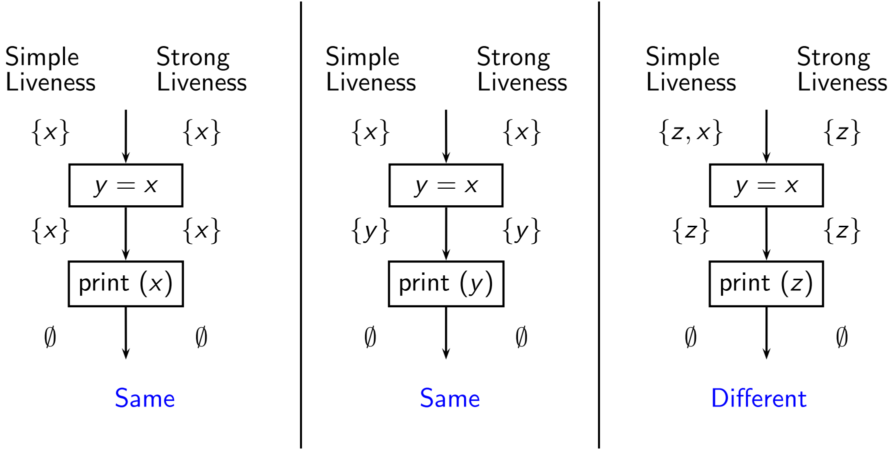
Understanding Strong Liveness



Understanding Strong Liveness



Understanding Strong Liveness



Live Variables Analysis: Simple and Strong Liveness

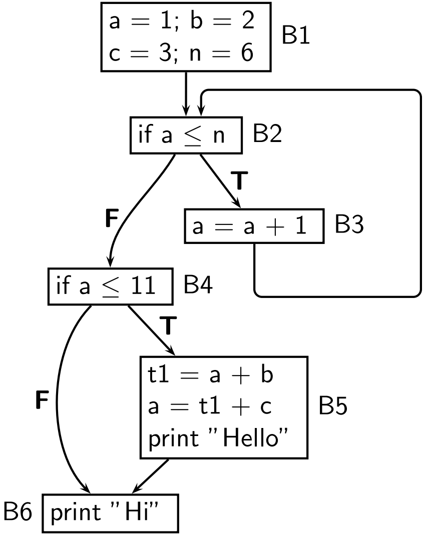
- A variable is live at a program point if its current value is likely to be used later

Live Variables Analysis: Simple and Strong Liveness

- A variable is live at a program point if its current value is likely to be used later
- We want to compute the smallest set of variables that are live

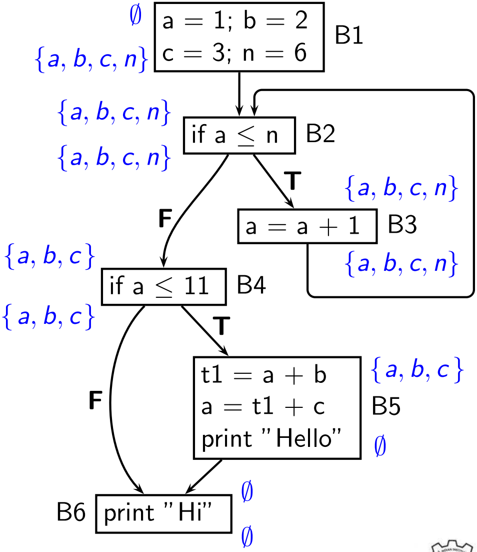
Live Variables Analysis: Simple and Strong Liveness

- A variable is live at a program point if its current value is likely to be used later
- We want to compute the smallest set of variables that are live



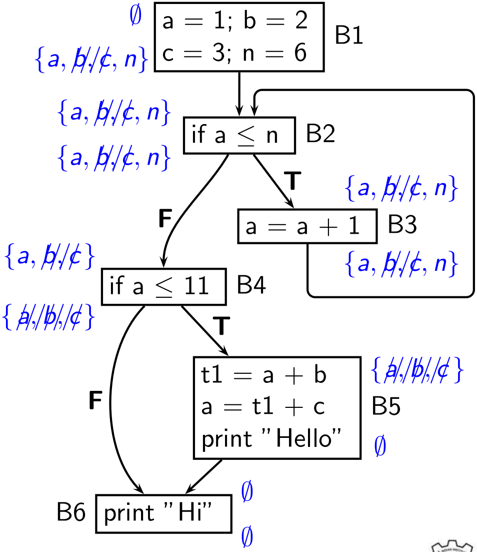
Live Variables Analysis: Simple and Strong Liveness

- A variable is live at a program point if its current value is likely to be used later
- We want to compute the smallest set of variables that are live
- Simple liveness considers every use of a variable as useful



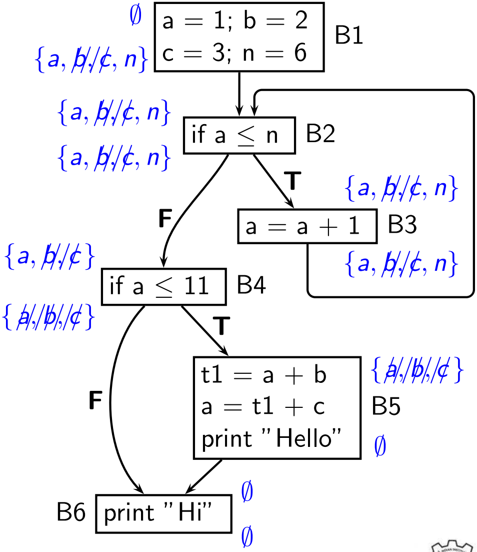
Live Variables Analysis: Simple and Strong Liveness

- A variable is live at a program point if its current value is likely to be used later
- We want to compute the smallest set of variables that are live
- Simple liveness considers every use of a variable as useful
- Strong liveness checks the liveness of the result before declaring the operands to be live



Live Variables Analysis: Simple and Strong Liveness

- A variable is live at a program point if its current value is likely to be used later
- We want to compute the smallest set of variables that are live
- Simple liveness considers every use of a variable as useful
- Strong liveness checks the liveness of the result before declaring the operands to be live
- Strong liveness is more precise than simple liveness



Data Flow Equations for Strongly Live Variables Analysis

$$In_n = f_n(Out_n)$$

$$Out_n = \begin{cases} BI & n \text{ is } End \\ \bigcup\limits_{s \in succ(n)} In_s & \text{otherwise} \end{cases}$$

where,

$$f_n(X) = \begin{cases} (X - \{y\}) \cup (Opd(e) \cap \mathbb{V}\text{ar}) & n \text{ is } y = e, e \in \mathbb{E}\text{xpr}, \;\; y \in X \\ X - \{y\} & n \text{ is } input(y) \\ X \cup \{y\} & n \text{ is } use(y) \\ X & \text{otherwise} \end{cases}$$

Data Flow Equations for Strongly Live Variables Analysis

$$In_n = f_n(Out_n)$$

$$Out_n = \begin{cases} BI & n \text{ is } End \\ \bigcup\limits_{s \in succ(n)} In_s & \text{otherwise} \end{cases}$$

where,

$$f_n(X) = \begin{cases} (X - \{y\}) \cup (Opd(e) \cap \mathbb{V}\text{ar}) & n \text{ is } y = e, e \in \mathbb{E}\text{xpr}, \boxed{y \in X} \\ X - \{y\} & n \text{ is } input(y) \\ X \cup \{y\} & n \text{ is } use(y) \\ X & \text{otherwise} \end{cases}$$

If y is not strongly live, the assignment is skipped using the "otherwise" clause

Properties of Strongly Live Variable Analysis

- What is $\widehat{L}$ for strongly live variables analysis?
- Is strongly live variables analysis a bit vector framework?
- Is strongly live variables analysis a separable framework?
- Is strongly live variables analysis distributive? Monotonic?

Properties of Strongly Live Variable Analysis

- What is $\widehat{L}$ for strongly live variables analysis?
  - $\widehat{L} = \{0, 1\}, 1 \sqsubseteq 0$
- Is strongly live variables analysis a bit vector framework?
- Is strongly live variables analysis a separable framework?
- Is strongly live variables analysis distributive? Monotonic?

Properties of Strongly Live Variable Analysis

- What is $\widehat{L}$ for strongly live variables analysis?
  - $\widehat{L} = \{0, 1\}, 1 \sqsubseteq 0$
- Is strongly live variables analysis a bit vector framework?
  - No because data flow equations cannot be defined only in terms of bit vector operations
- Is strongly live variables analysis a separable framework?
- Is strongly live variables analysis distributive? Monotonic?

Properties of Strongly Live Variable Analysis

- What is $\widehat{L}$ for strongly live variables analysis?
  - $\widehat{L} = \{0, 1\}, 1 \sqsubseteq 0$
- Is strongly live variables analysis a bit vector framework?
  - No because data flow equations cannot be defined only in terms of bit vector operations
- Is strongly live variables analysis a separable framework?
  - No, because strong liveness of variables occurring in RHS of an assignment may depend on the variable occurring in LHS
- Is strongly live variables analysis distributive? Monotonic?

Properties of Strongly Live Variable Analysis

- What is $\widehat{L}$ for strongly live variables analysis?
  - $\widehat{L} = \{0, 1\}, 1 \sqsubseteq 0$
- Is strongly live variables analysis a bit vector framework?
  - No because data flow equations cannot be defined only in terms of bit vector operations
- Is strongly live variables analysis a separable framework?
  - No, because strong liveness of variables occurring in RHS of an assignment may depend on the variable occurring in LHS
- Is strongly live variables analysis distributive? Monotonic?
  - Distributive, and hence monotonic

Distributivity of Strongly Live Variables Analysis (1)

We need to prove that

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

Distributivity of Strongly Live Variables Analysis (1)

We need to prove that

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

- Intuitively,
  - The value does not depend on the argument X
  - Incomparable results cannot be produced
    (A fixed set of variable are excluded or included)

Distributivity of Strongly Live Variables Analysis (1)

We need to prove that

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

- Intuitively,
  - The value does not depend on the argument X
  - Incomparable results cannot be produced
    (A fixed set of variable are excluded or included)
- Formally,
  - We prove it for $input(y)$, $use(y)$, $y = e$, and empty statements independently

Distributivity of Strongly Live Variables Analysis (2)

- For *input*(y) statement:
- For *use*(y) statement:
- For empty statement:

Distributivity of Strongly Live Variables Analysis (2)

- For *input*(y) statement: $$\begin{aligned} f_n(X_1 \cup X_2) &= (X_1 \cup X_2) - \{y\} \\ &= (X_1 - \{y\}) \cup (X_2 - \{y\}) \\ &= f_n(X_1) \cup f_n(X_2) \end{aligned}$$
- For *use*(y) statement:
- For empty statement:

Distributivity of Strongly Live Variables Analysis (2)

- For *input*(y) statement: $$\begin{aligned} f_n(X_1 \cup X_2) &= (X_1 \cup X_2) - \{y\} \\ &= (X_1 - \{y\}) \cup (X_2 - \{y\}) \\ &= f_n(X_1) \cup f_n(X_2) \end{aligned}$$
- For *use*(y) statement: $$\begin{aligned} f_n(X_1 \cup X_2) &= (X_1 \cup X_2) \cup \{y\} \\ &= (X_1 \cup \{y\}) \cup (X_2 \cup \{y\}) \\ &= f_n(X_1) \cup f_n(X_2) \end{aligned}$$
- For empty statement:

Distributivity of Strongly Live Variables Analysis (2)

- For *input*(y) statement: $$\begin{aligned} f_n(X_1 \cup X_2) &= (X_1 \cup X_2) - \{y\} \\ &= (X_1 - \{y\}) \cup (X_2 - \{y\}) \\ &= f_n(X_1) \cup f_n(X_2) \end{aligned}$$
- For *use*(y) statement: $$\begin{aligned} f_n(X_1 \cup X_2) &= (X_1 \cup X_2) \cup \{y\} \\ &= (X_1 \cup \{y\}) \cup (X_2 \cup \{y\}) \\ &= f_n(X_1) \cup f_n(X_2) \end{aligned}$$
- For empty statement: $f_n(X_1 \cup X_2) = X_1 \cup X_2 \quad = \quad f_n(X_1) \cup f_n(X_2)$

Distributivity of Strongly Live Variables Analysis (3)

For $y = e$ statement: Let $Y = Opd(e) \cap \mathbb{V}\text{ar}$. There are three cases:

- $y \in X_1, y \in X_2$.

- $y \in X_1, y \notin X_2$.

- $y \notin X_1, y \notin X_2$.

Distributivity of Strongly Live Variables Analysis (3)

For $y = e$ statement: Let $Y = Opd(e) \cap \mathbb{V}\text{ar}$. There are three cases:

- $y \in X_1, y \in X_2$.

$$\begin{aligned} f_n(X_1 \cup X_2) &= ((X_1 \cup X_2) - \{y\}) \cup Y \\ &= (X_1 - \{y\}) \cup (X_2 - \{y\}) \cup Y \\ &= ((X_1 - \{y\}) \cup Y) \cup ((X_2 - \{y\}) \cup Y) \\ &= f_n(X_1) \cup f_n(X_2) \end{aligned}$$

- $y \in X_1, y \notin X_2$.

- $y \notin X_1, y \notin X_2$.

Distributivity of Strongly Live Variables Analysis (3)

For $y = e$ statement: Let $Y = Opd(e) \cap \mathbb{V}\text{ar}$. There are three cases:

- $y \in X_1, y \in X_2$.

$$\begin{aligned} f_n(X_1 \cup X_2) &= ((X_1 \cup X_2) - \{y\}) \cup Y \\ &= (X_1 - \{y\}) \cup (X_2 - \{y\}) \cup Y \\ &= ((X_1 - \{y\}) \cup Y) \cup ((X_2 - \{y\}) \cup Y) \\ &= f_n(X_1) \cup f_n(X_2) \end{aligned}$$

- $y \in X_1, y \notin X_2$.

$$\begin{aligned} f_n(X_1 \cup X_2) &= ((X_1 \cup X_2) - \{y\}) \cup Y \\ &= ((X_1 - \{y\}) \cup Y) \cup (X_2) && (\because y \notin X_2) \\ &= f_n(X_1) \cup f_n(X_2) && y \notin X_2 \Rightarrow f_n(X_2) \text{ is identity} \end{aligned}$$

- $y \notin X_1, y \notin X_2$.

Distributivity of Strongly Live Variables Analysis (3)

For $y = e$ statement: Let $Y = Opd(e) \cap \mathbb{V}\text{ar}$. There are three cases:

- $y \in X_1, y \in X_2$.

$$\begin{aligned} f_n(X_1 \cup X_2) &= ((X_1 \cup X_2) - \{y\}) \cup Y \\ &= (X_1 - \{y\}) \cup (X_2 - \{y\}) \cup Y \\ &= ((X_1 - \{y\}) \cup Y) \cup ((X_2 - \{y\}) \cup Y) \\ &= f_n(X_1) \cup f_n(X_2) \end{aligned}$$

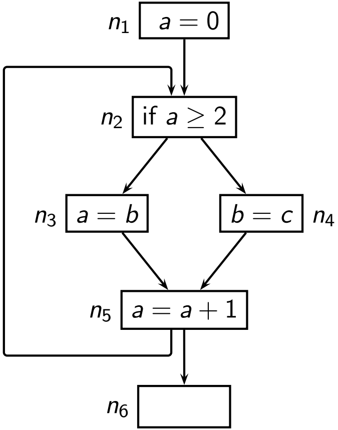
- $y \in X_1, y \notin X_2$.

$$\begin{aligned} f_n(X_1 \cup X_2) &= ((X_1 \cup X_2) - \{y\}) \cup Y \\ &= ((X_1 - \{y\}) \cup Y) \cup (X_2) && (\because y \notin X_2) \\ &= f_n(X_1) \cup f_n(X_2) && y \notin X_2 \Rightarrow f_n(X_2) \text{ is identity} \end{aligned}$$

- $y \notin X_1, y \notin X_2$.

$$f_n(X_1 \cup X_2) = X_1 \cup X_2 \quad = \quad f_n(X_1) \cup f_n(X_2)$$

Tutorial Problem for strongly Live Variables Analysis



Result of Strongly Live Variables Analysis

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

Tutorial Problem: Strongly May-Must Liveness Analysis?

- Instead of viewing liveness information as
  - a map $\mathbb{V}\text{ar} \rightarrow \{0, 1\}$ with the lattice $\{0, 1\}$,

  view it as
  - a map $\mathbb{V}\text{ar} \rightarrow \widehat{L}$ where $\widehat{L}$ is the May-Must Lattice
- Write the data flow equations
- Prove that the flow functions are distributive

Part 5

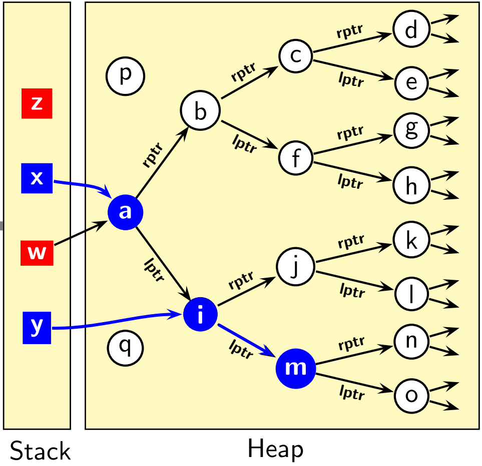
Heap Reference Analysis

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
```

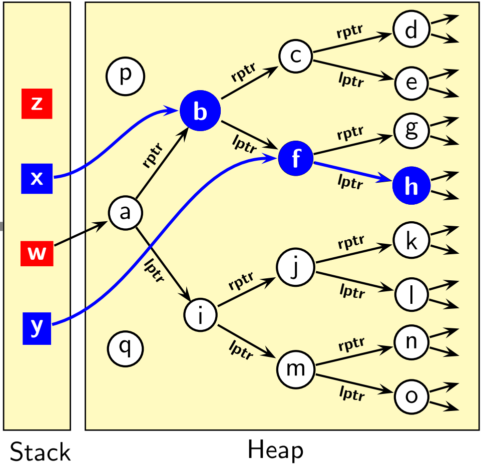


Motivating Example for Heap Liveness Analysis

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
```

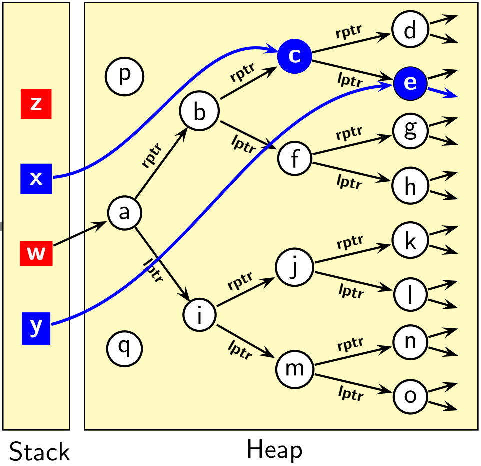


Motivating Example for Heap Liveness Analysis

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

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

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

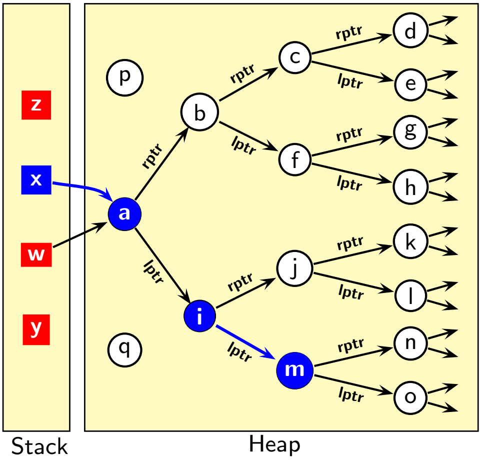
6 y = y.lptr

y.lptr = y.rptr = null

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

x = y = z = null

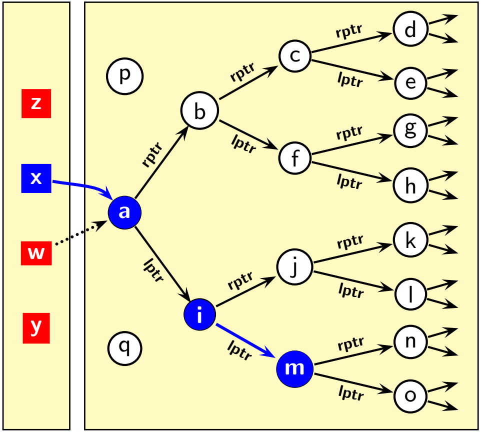
While loop is not executed even once



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

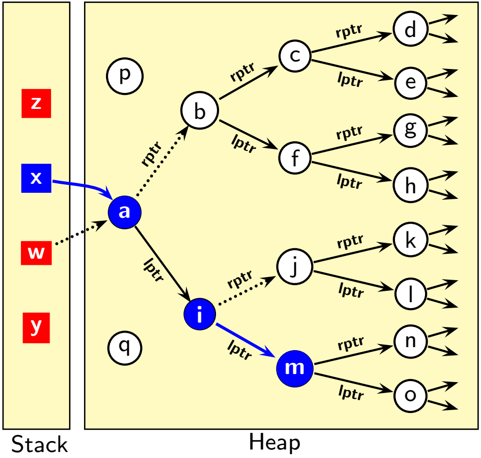
While loop is not executed even once



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

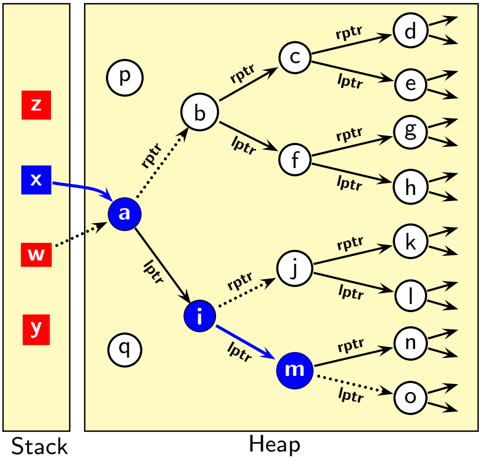
While loop is not executed even once



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
```

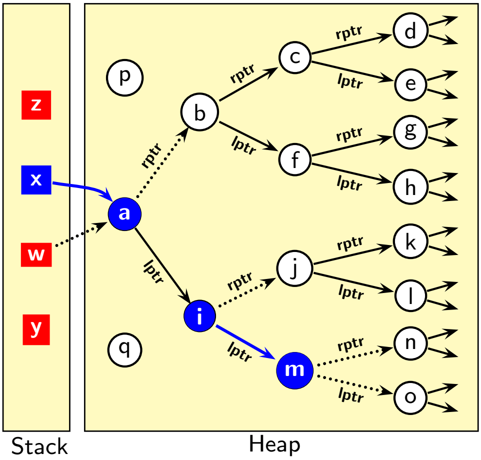
While loop is not executed even once



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
```

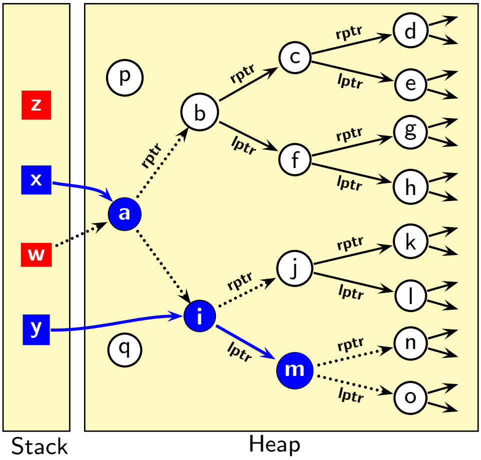
While loop is not executed even once



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
```

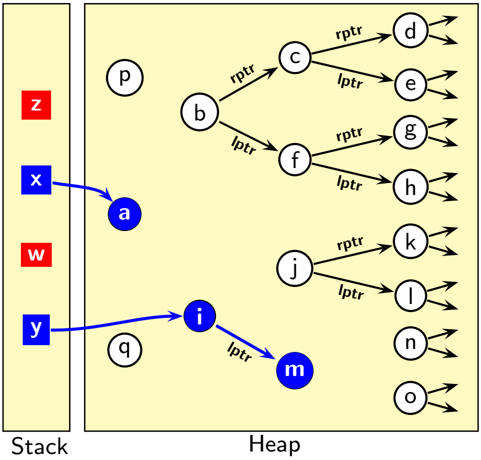
While loop is not executed even once



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
```

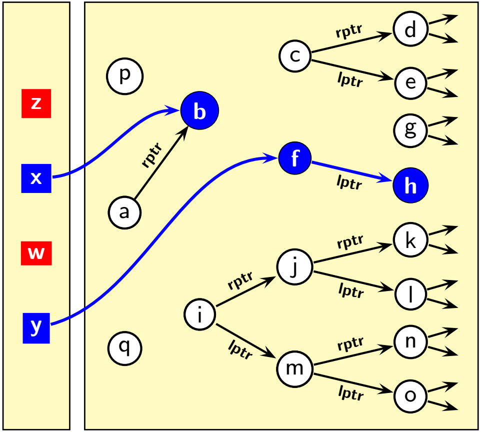
While loop is not executed even once



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

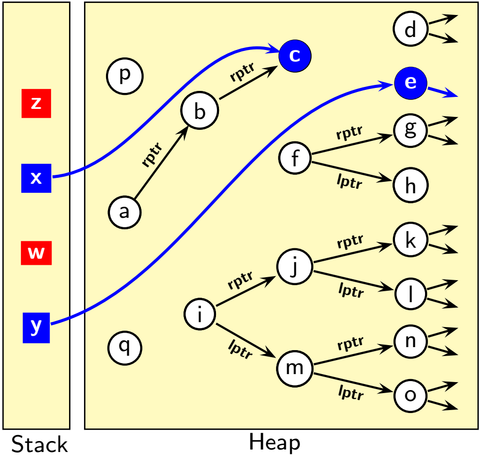
While loop is executed once



Our Solution

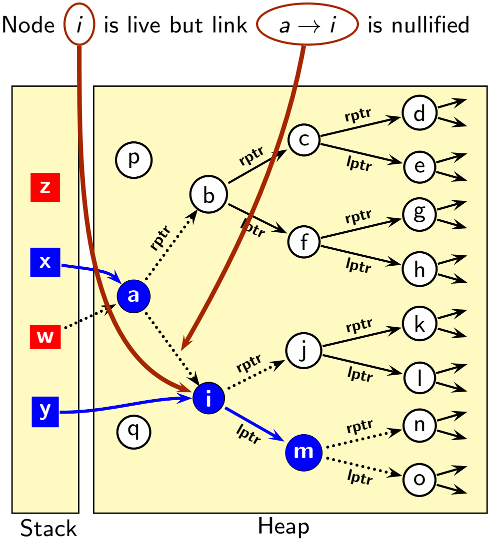
```
   y = z = null
1  w = x
   w = null
2  while (x.data < max)
   {      x.lptr = null
3         x = x.rptr       }
   x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null
4  y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
5  z = New class_of_z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

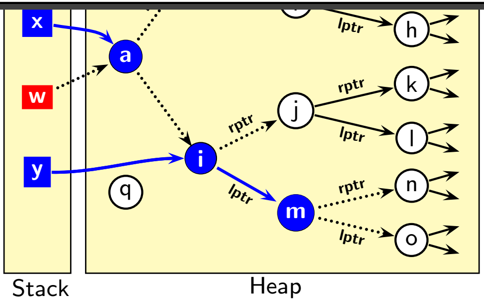
```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr       }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z =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

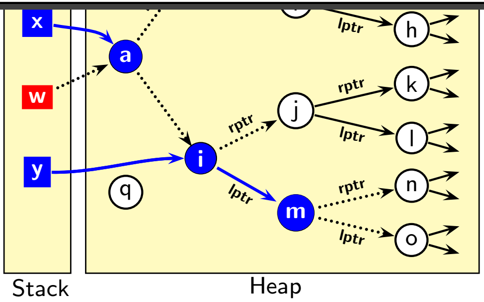
- The memory address that *x* holds when the execution reaches a given program point is not an invariant of program execution



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
```

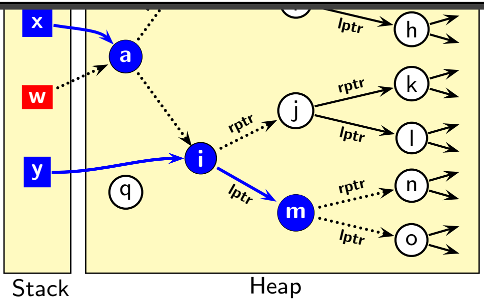
- The memory address that x holds when the execution reaches a given program point is not an invariant of program execution
- Whether we dereference lptr out of x or rptr out of x at a given program point is an invariant of program execution



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
```

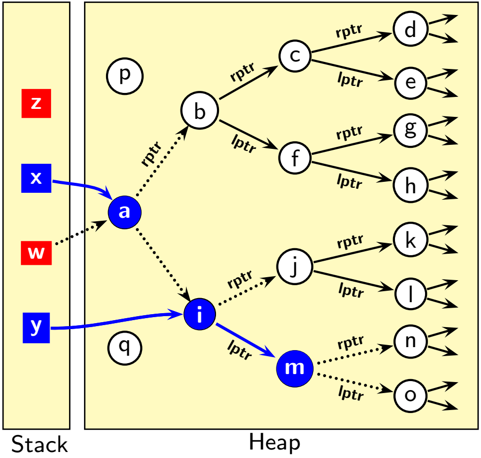
- The memory address that x holds when the execution reaches a given program point is not an invariant of program execution
- Whether we dereference lptr out of x or rptr out of x at a given program point is an invariant of program execution
- *A static analysis can discover only invariants*



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

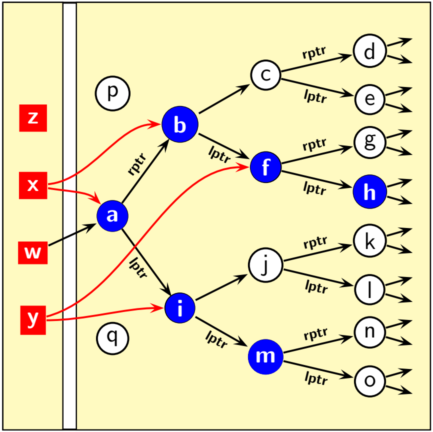
  - *Alias Analysis*. Identify all possible ways of accessing a link
  - *Liveness Analysis*. For each program point, identify "dead" links (i.e. links which are not accessed after that program point)
  - *Availability and Anticipability Analyses*. Dead links should be reachable for making null assignment.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



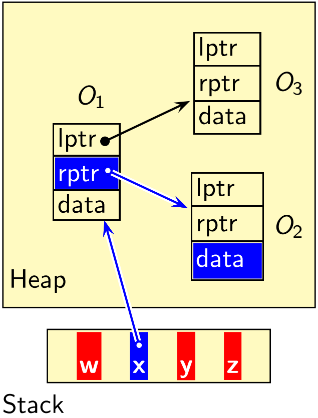
- Root variables : x, y, z
- Field names : rptr, lptr
- Access path : x➔rptr➔lptr
  Semantically, sequence of "links"
- Frontier : name of the last link
- Live access path : If the link corresponding to its frontier is used in future

What Makes a Link Live?

Assuming that a statement must be executed, 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	$x, x\to$rptr
if (x.rptr.data $<$ sum)	x, O_1, O_2	$x, x\to$rptr

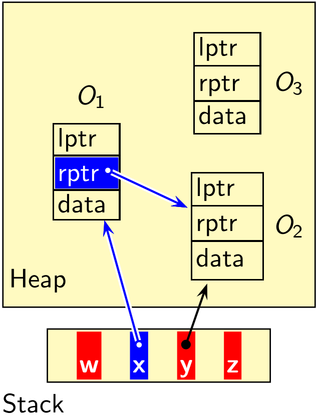


What Makes a Link Live?

Assuming that a statement must be executed, 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.rptr

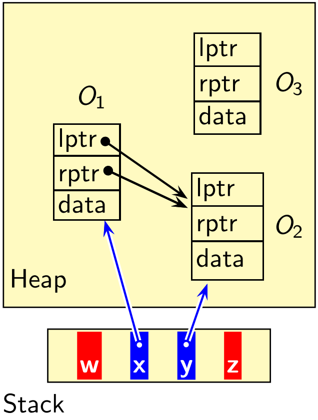


What Makes a Link Live?

Assuming that a statement must be executed, 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	$x, x.\text{rptr}$
$x.\text{lptr} = y$	x, O_1, y	x, y

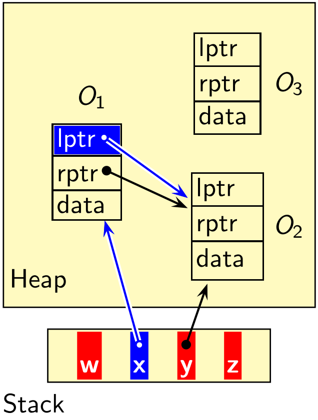


What Makes a Link Live?

Assuming that a statement must be executed, 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

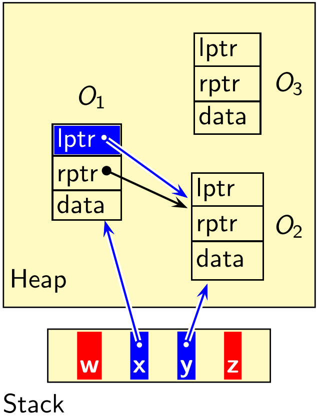


What Makes a Link Live?

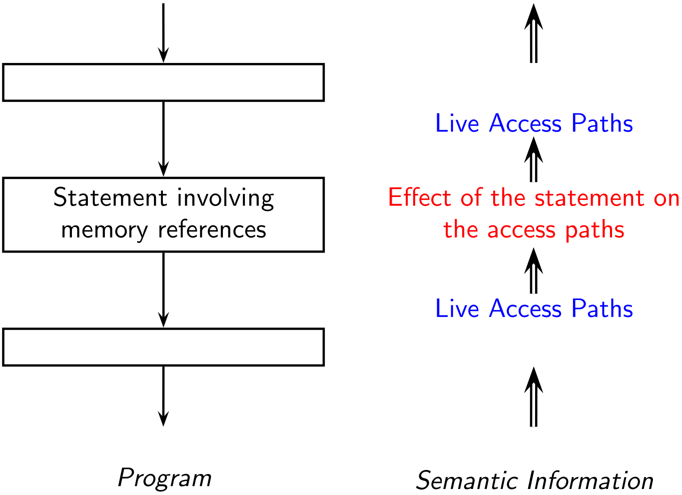
Assuming that a statement must be executed, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

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

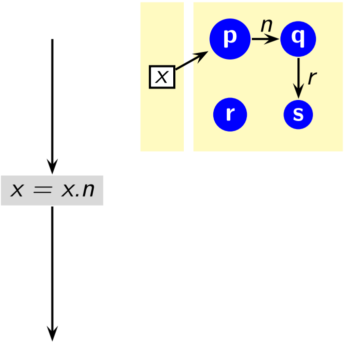
Example	Objects read	Live access paths
if (x.lptr == null)	x, O_1	$x, x\rightarrow$lptr
if ($y == x$.lptr)	x, O_1, y	$x, x\rightarrow$lptr$, y$



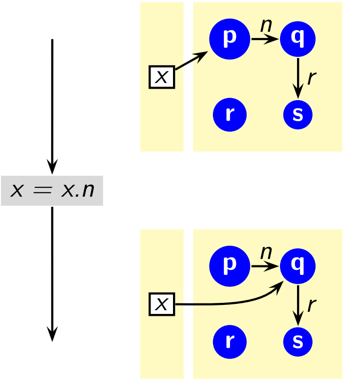
Liveness Analysis



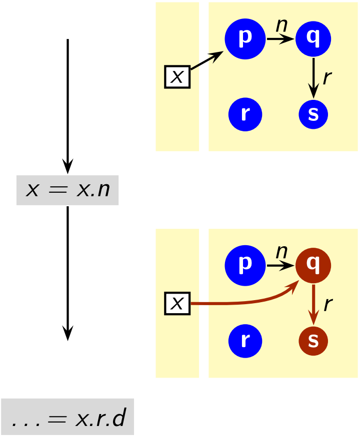
Key Idea #2 : Transfer of Access Paths



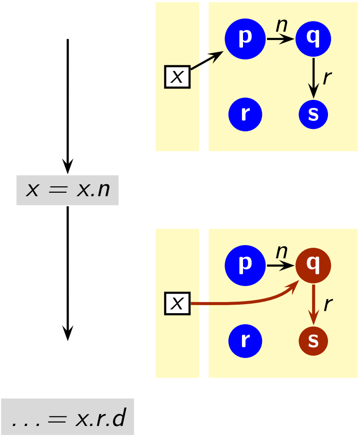
Key Idea #2 : Transfer of Access Paths



Key Idea #2 : Transfer of Access Paths



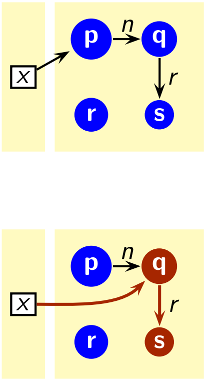
Key Idea #2 : Transfer of Access Paths



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

Key Idea #2 : Transfer of Access Paths

$x = x.n$



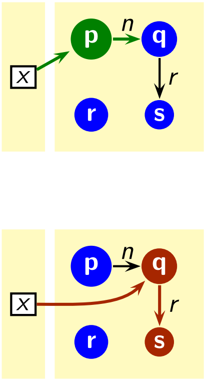
$\ldots = x.r.d$

Analysis

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

Key Idea #2 : Transfer of Access Paths

$x = x.n$

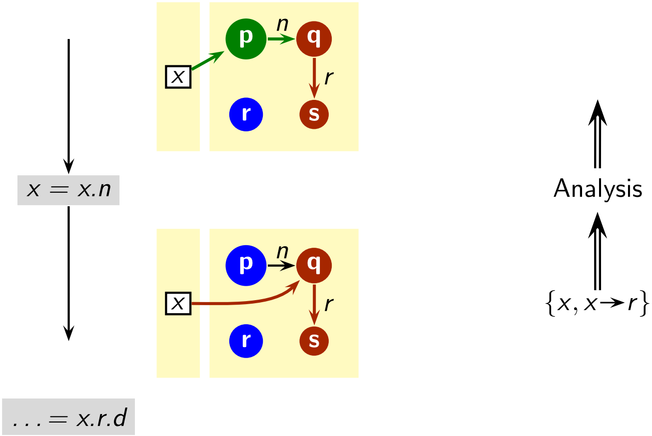


Analysis

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

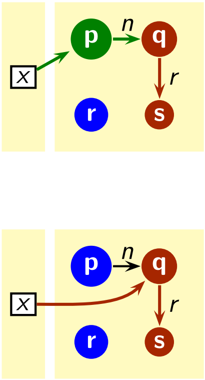
$\ldots = x.r.d$

Key Idea #2 : Transfer of Access Paths



Key Idea #2 : Transfer of Access Paths

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



$x = x.n$

$\ldots = x.r.d$

$\Uparrow$

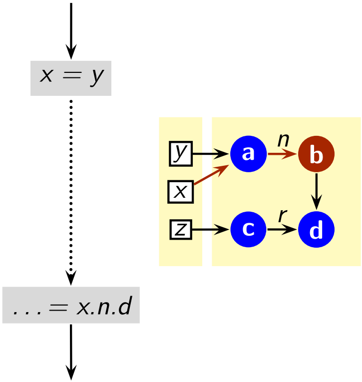
Analysis

$\Uparrow$

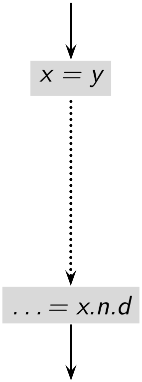
$\{x, x\rightarrow r\}$

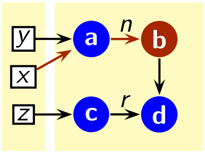
x after the assignment is same as $x\rightarrow n$ before the assignment

Key Idea #3 : Liveness Closure Under Link Aliasing



Key Idea #3 : Liveness Closure Under Link Aliasing



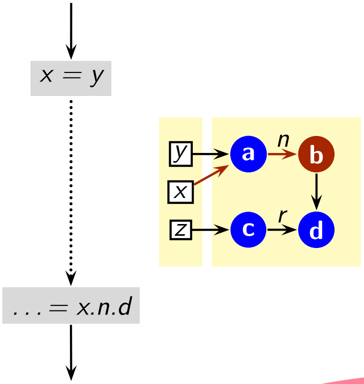


x and y are node aliases

$x.n$ 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



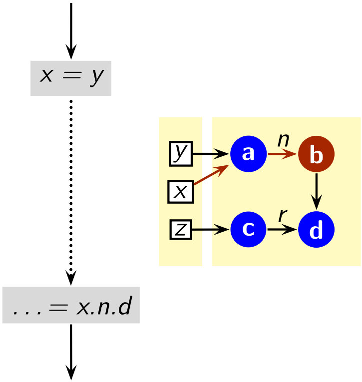
x and y are node aliases

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

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

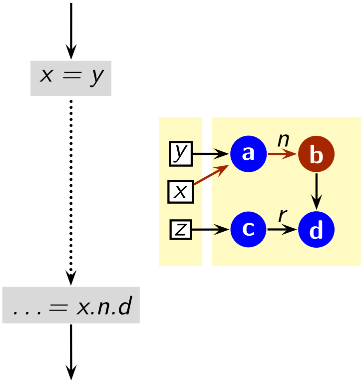
Nullifying $y \rightarrow n$ will have the side effect of nullifying $x \rightarrow n$

Explicit and Implicit Liveness



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

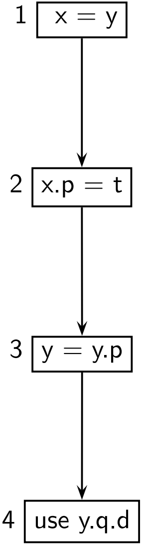
Explicit and Implicit Liveness



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

$y \rightarrow n$ is implicitly live
$x \rightarrow n$ is explicitly live

Key Idea #4: Aliasing is Required with Explicit Liveness



Key Idea #4: Aliasing is Required with Explicit Liveness

Explicit Liveness

1 x = y

$\{x, y,\ y{\rightarrow}p, y{\rightarrow}p{\rightarrow}q\ \}$

2 x.p = t

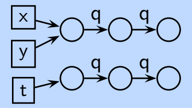
$\{y, y{\rightarrow}p, y{\rightarrow}p{\rightarrow}q\}$

3 y = y.p

$\{y, y{\rightarrow}q\}$

4 use y.q.d

Key Idea #4: Aliasing is Required with Explicit Liveness



Explicit Liveness

1 x = y

$\{x, y,\ y\rightarrow p, y\rightarrow p\rightarrow q\ \}$

2 x.p = t

$\{y, y\rightarrow p, y\rightarrow p\rightarrow q\}$

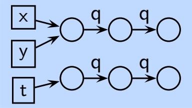
3 y = y.p

$\{y, y\rightarrow q\}$

4 use y.q.d

Key Idea #4: Aliasing is Required with Explicit Liveness

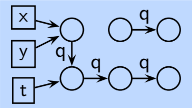
Explicit Liveness



1 $\boxed{x = y}$

$\{x, y,\ y{\rightarrow}p, y{\rightarrow}p{\rightarrow}q\ \}$

2 $\boxed{x.p = t}$



$\{y, y{\rightarrow}p, y{\rightarrow}p{\rightarrow}q\}$

3 $\boxed{y = y.p}$

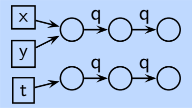
$\{y, y{\rightarrow}q\}$

4 $\boxed{\text{use } y.q.d}$

Key Idea #4: Aliasing is Required with Explicit Liveness

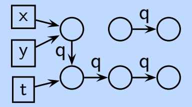
Explicit Liveness

1 x = y



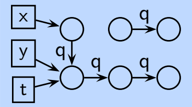
$\{x, y,\ y\rightarrow p, y\rightarrow p\rightarrow q\ \}$

2 x.p = t



$\{y, y\rightarrow p, y\rightarrow p\rightarrow q\}$

3 y = y.p



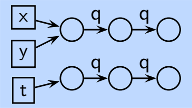
$\{y, y\rightarrow q\}$

4 use y.q.d

Key Idea #4: Aliasing is Required with Explicit Liveness

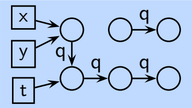
Explicit Liveness

1 x = y



$\{x, y,\ y \rightarrow p, y \rightarrow p \rightarrow q\ \}$

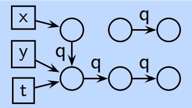
2 x.p = t



$\{y, y \rightarrow p, y \rightarrow p \rightarrow q\}$

Effect of Aliasing
$y \rightarrow p \equiv x \rightarrow p$
$y \rightarrow p \rightarrow q \equiv x \rightarrow p \rightarrow q$

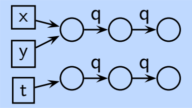
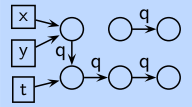
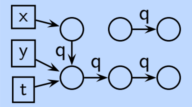
3 y = y.p



$\{y, y \rightarrow q\}$

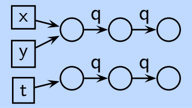
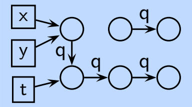
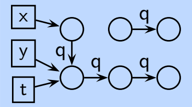
4 use y.q.d

Key Idea #4: Aliasing is Required with Explicit Liveness

	Program	*Explicit Liveness*	*Required Liveness*
	1 x = y		
		$\{x, y,\ y \rightarrow p, y \rightarrow p \rightarrow q\ \}$	$\{x, y,\ t, t \rightarrow q\ \}$
	2 x.p = t		
		$\{y, y \rightarrow p, y \rightarrow p \rightarrow q\}$	$\{y, y \rightarrow p, y \rightarrow p \rightarrow q\}$
	3 y = y.p		
		$\{y, y \rightarrow q\}$	$\{y, y \rightarrow q\}$
	4 use y.q.d		

Effect of Aliasing
$y \rightarrow p \equiv x \rightarrow p$
$y \rightarrow p \rightarrow q \equiv x \rightarrow p \rightarrow q$

Key Idea #4: Aliasing is Required with Explicit Liveness

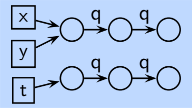
	Program	*Explicit Liveness*	*Required Liveness*
	1 $\boxed{x = y}$		
		$\{x, y, \boxed{y \rightarrow p, y \rightarrow p \rightarrow q}\}$	$\{x, y, \boxed{t, t \rightarrow q}\}$
	2 $\boxed{x.p = t}$		
		$\{y, y \rightarrow p, y \rightarrow p \rightarrow q\}$	$\{y, y \rightarrow p, y \rightarrow p \rightarrow q\}$
	3 $\boxed{y = y.p}$		
		$\{y, y \rightarrow q\}$	$\{y, y \rightarrow q\}$
	4 $\boxed{\text{use } y.q.d}$		

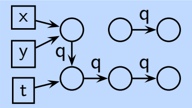
Spurious

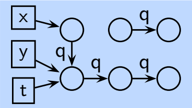
Missing

Effect of Aliasing
$y \rightarrow p \equiv x \rightarrow p$
$y \rightarrow p \rightarrow q \equiv x \rightarrow p \rightarrow q$

Key Idea #4: Aliasing is Required with Explicit Liveness







1 x = y

2 x.p = t

3 y = y.p

4 use y.q.d

	Explicit Liveness	*Required Liveness*
	$\{x, y, y \rightarrow p, y \rightarrow p \rightarrow q\}$	$\{x, y, t, t \rightarrow q\}$
	$\{y, y \rightarrow p, y \rightarrow p \rightarrow q\}$	$\{y, y \rightarrow p, y \rightarrow p \rightarrow q\}$

Spurious

Missing

Effect of Aliasing
$y \rightarrow p \equiv x \rightarrow p$
$y \rightarrow p \rightarrow q \equiv x \rightarrow p \rightarrow q$

The need of link alias closure of LHS

- Transferring liveness to RHS (soundness)
- Killing liveness (precision)

Link alias closure of RHS can be computed later for implicit liveness

Notation for Defining Flow Functions for Explicit Liveness

- Basic entities
  - Variables $u, v \in \mathbb{V}\text{ar}$
  - Pointer variables $w, x, y, z \in \mathbf{P} \subseteq \mathbb{V}\text{ar}$
  - Pointer fields $f, g, h \in pF$
  - Non-pointer fields $a, b, c, d \in npF$
- Additional notation
  - Sequence of pointer fields $\sigma \in pF^*$ (could be ϵ)
  - Access paths $\rho \in \mathbf{P} \times pF^*$
    Example: $\{x, x \rightarrow f, x \rightarrow f \rightarrow g\}$
  - Summarized access paths rooted at x or $x \rightarrow \sigma$ for a given x and σ
    - $x \rightarrow * = \{x \rightarrow \sigma \mid \sigma \in pF^*\}$
    - $x \rightarrow \sigma \rightarrow * = \{x \rightarrow \sigma \rightarrow \sigma' \mid \sigma' \in pF^*\}$

Data Flow Equations for Explicit Liveness Analysis

$$In_n = \big(Out_n - \mathsf{Kill}_n(Out_n)\big) \;\cup\; \mathsf{Gen}_n(Out_n)$$

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

Flow Functions for Explicit Liveness Analysis

Let A denote May Aliases at the exit of node n

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
$x = y$	$\{y \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x = y.f$	$\{y \rightarrow f \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x.f = y$	$\left\{y \rightarrow \sigma \;\middle|\; z \rightarrow f \rightarrow \sigma \in X, z \in A(x)\right\}$	$\bigcup_{z \in Must(A)(x)} z \rightarrow f \rightarrow *$
$x = new$	$\emptyset$	$x \rightarrow *$
$x = null$	$\emptyset$	$x \rightarrow *$
other	$\emptyset$	$\emptyset$

Flow Functions for Explicit Liveness Analysis

Let A denote May Aliases at the exit of node n

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
$x = y$	$\{y \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x = y.f$	$\{y \rightarrow f \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x.f = y$	$\left\{y \rightarrow \sigma \;\middle\vert\; \boxed{z \rightarrow f \rightarrow \sigma \in X, z \in A(x)}\right\}$	$\bigcup_{z \in Must(A)(x)} z \rightarrow f \rightarrow *$
$x = new$	$\emptyset$	$x \rightarrow *$
$x = null$	$\emptyset$	$x \rightarrow *$
other	$\emptyset$	$\emptyset$

May link aliasing for soundness

Flow Functions for Explicit Liveness Analysis

Let A denote May Aliases at the exit of node n

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
$x = y$	$\{y \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x = y.f$	$\{y \rightarrow f \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x.f = y$	$\left\{y \rightarrow \sigma \;\middle\vert\; \boxed{z \rightarrow f \rightarrow \sigma \in X, z \in A(x)}\right\}$	$\boxed{\bigcup_{z \in Must(A)(x)} z \rightarrow f \rightarrow *}$
$x = new$	$\emptyset$	$x \rightarrow *$
$x = null$	$\emptyset$	$x \rightarrow *$
other	$\emptyset$	$\emptyset$

May link aliasing for soundness

Must link aliasing for precision

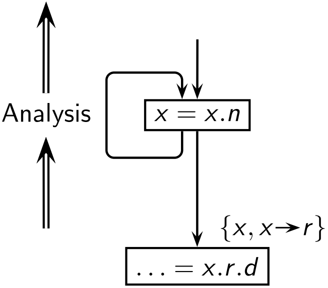
Flow Functions for Explicit Liveness Analysis

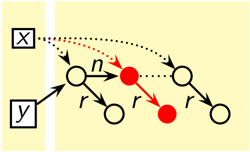
Let A denote May Aliases at the exit of node n

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
x		
x		
x		$*$
x		
x		
o		

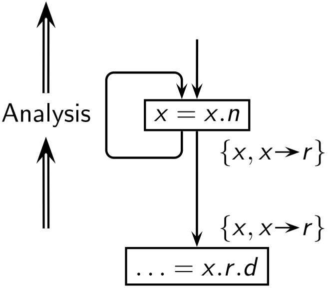
- Why is $y \notin \text{Gen}_n(X)$ for $x.f = y$ when $x \notin X$?
  If $x \notin Out_n$, we can do dead code elimination
- Why is $y \notin \text{Gen}_n(X)$ for $x = y.f$ when $x \rightarrow \sigma \notin X$?
  If $x \rightarrow \sigma \notin Out_n$, we can do dead code elimination
- Why is $x \notin \text{Gen}_n(X)$ for $x.f = y$?
  - If $x \rightarrow f \rightarrow \sigma \notin Out_n$, we can do dead code elimination
  - If $\exists\, x \rightarrow f \rightarrow \sigma \in Out_n$, then $\exists\, x \in Out_n$
    It will not be killed, so no need of $x \in \text{Gen}_n$

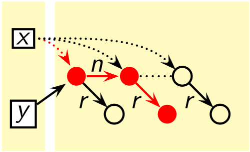
Computing Explicit Liveness Using Sets of Access Paths

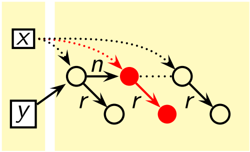




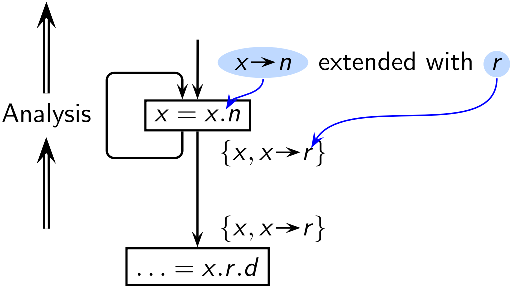
Computing Explicit Liveness Using Sets of Access Paths

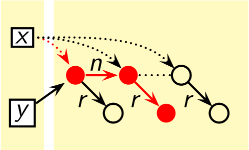


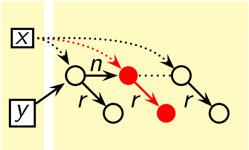




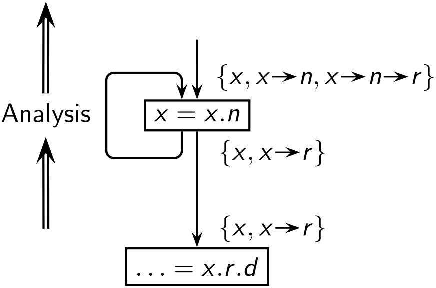
Computing Explicit Liveness Using Sets of Access Paths

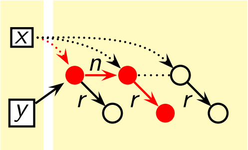


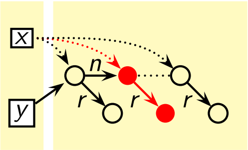




Computing Explicit Liveness Using Sets of Access Paths

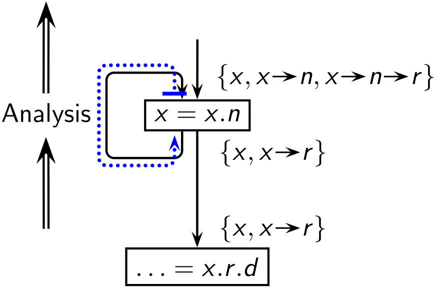






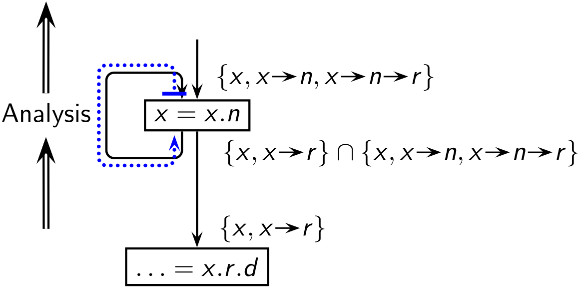
Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An *All Paths* problem



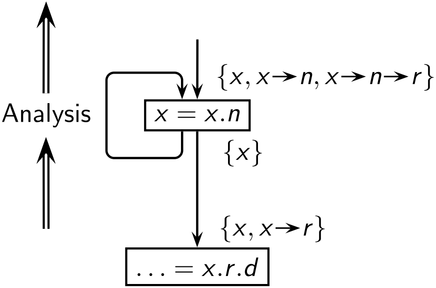
Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An *All Paths* problem



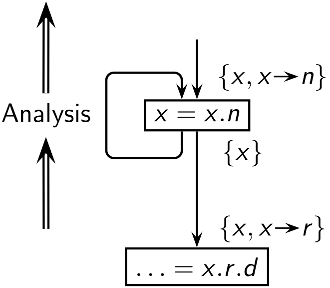
Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An *All Paths* problem



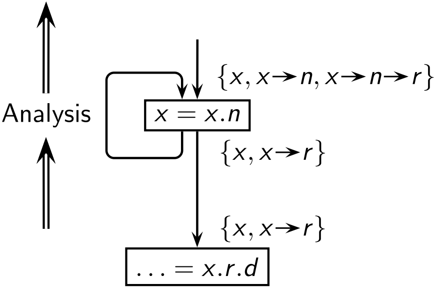
Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An *All Paths* problem



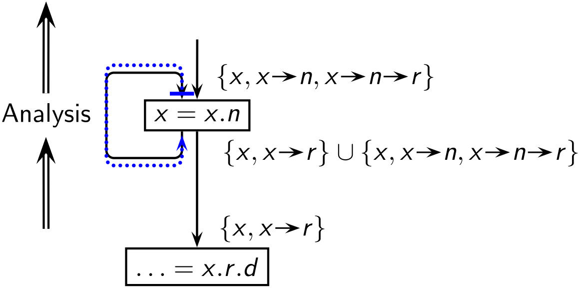
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



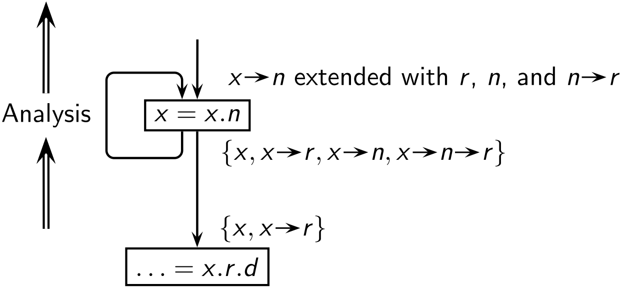
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



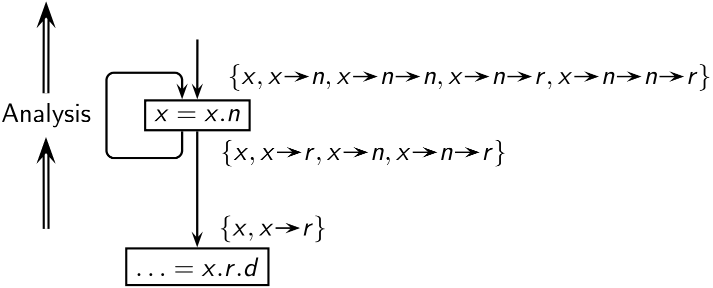
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



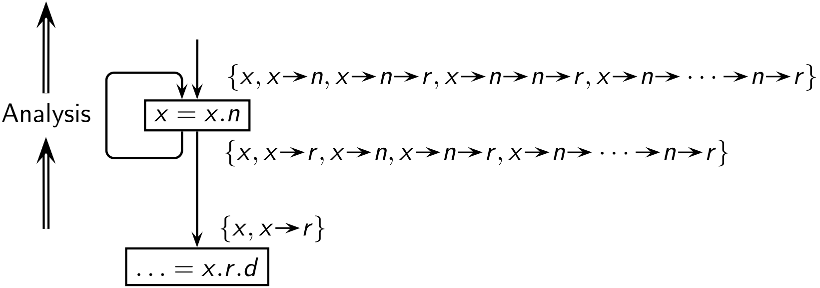
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



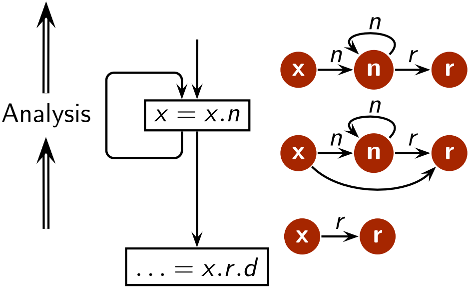
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



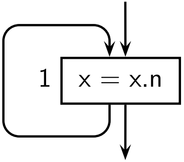
Infinite Number of Unbounded Access Paths

Key Idea #5: Using Graphs as Data Flow Values



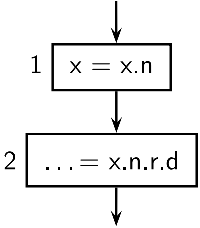
Finite Number of Bounded Structures

Key Idea #6 : Include Program Point in Graphs



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

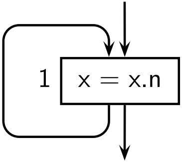
Different occurrences of n's in an access path are **Indistinguishable**



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

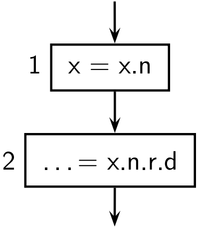
Different occurrences of n's in an access path are **Distinct**

Key Idea #6 : Include Program Point in Graphs



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

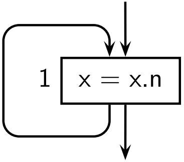
Different occurrences of n's in an access path are
Indistinguishable



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

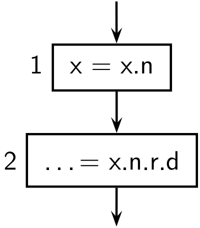
Different occurrences of n's in an access path are
Distinct
(pattern of subsequent dereferences could be distinct)

Key Idea #6 : Include Program Point in Graphs



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

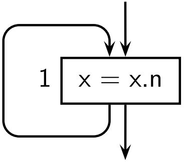
Different occurrences of n's in an access path are
Indistinguishable
(pattern of subsequent dereferences remains same)



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

Different occurrences of n's in an access path are
Distinct
(pattern of subsequent dereferences could be distinct)

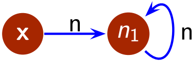
Key Idea #6 : Include Program Point in Graphs

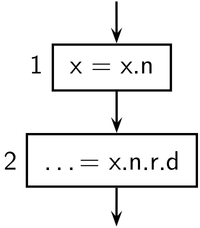


$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow n, \ldots\}$

Different occurrences of n's in an access path are
Indistinguishable
(pattern of subsequent dereferences remains same)

Access Graph :

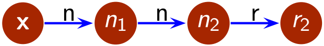




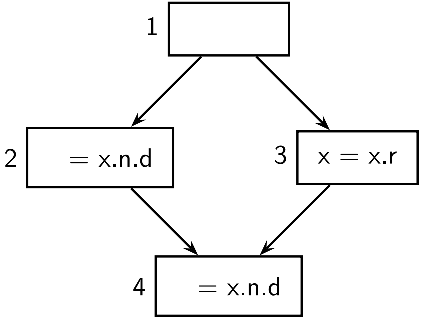
$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow r\}$

Different occurrences of n's in an access path are
Distinct
(pattern of subsequent dereferences could be distinct)

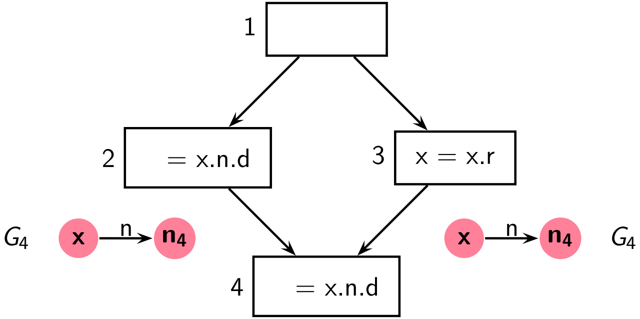
Access Graph :



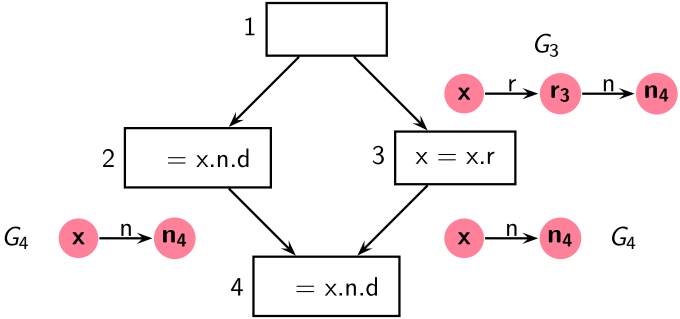
Inclusion of Program Point Facilitates Summarization



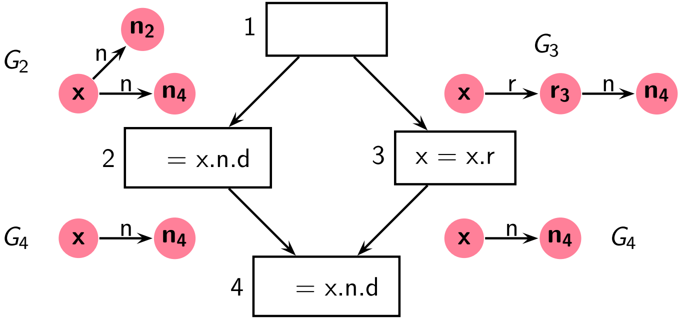
Inclusion of Program Point Facilitates Summarization



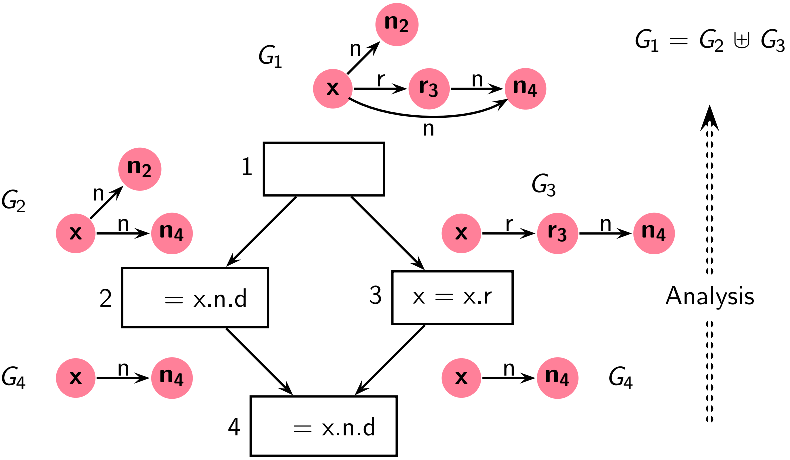
Inclusion of Program Point Facilitates Summarization



Inclusion of Program Point Facilitates Summarization

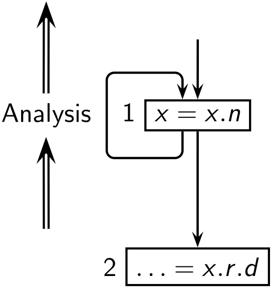


Inclusion of Program Point Facilitates Summarization



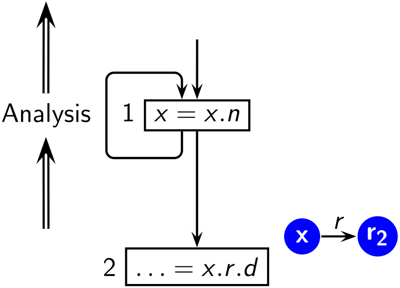
Inclusion of Program Point Facilitates Summarization

Iteration #1

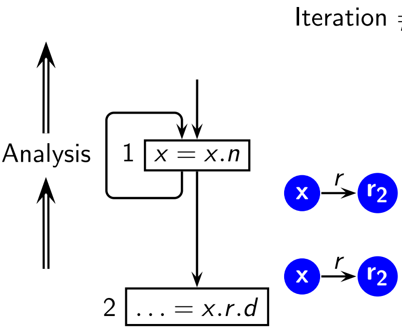


Inclusion of Program Point Facilitates Summarization

Iteration #1

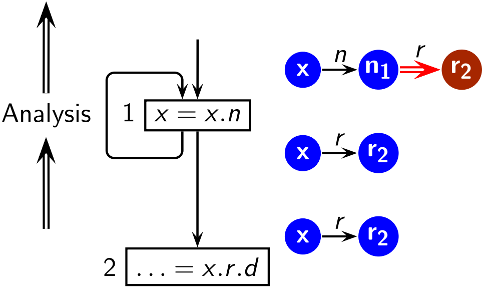


Inclusion of Program Point Facilitates Summarization



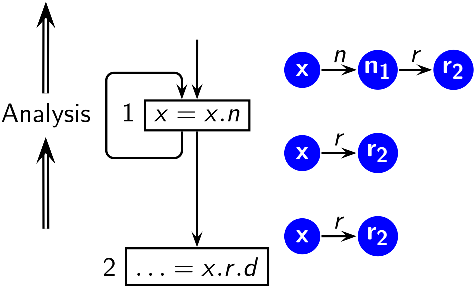
Inclusion of Program Point Facilitates Summarization

Iteration #1

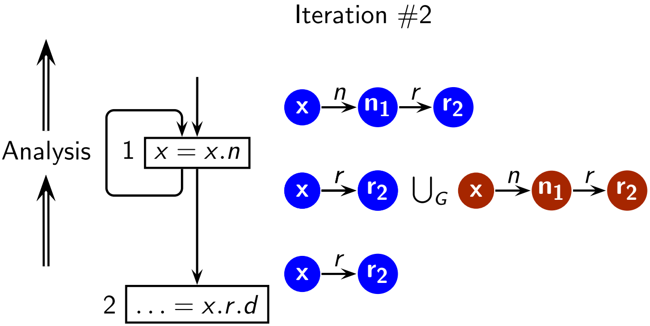


Inclusion of Program Point Facilitates Summarization

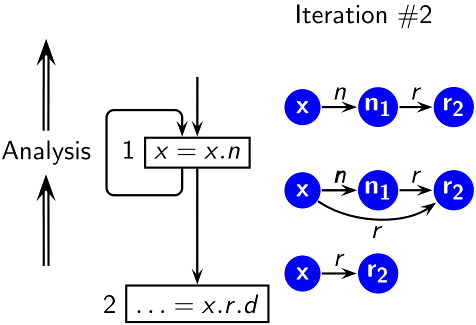
Iteration #1



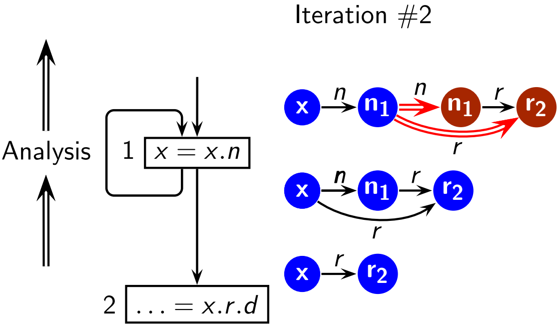
Inclusion of Program Point Facilitates Summarization



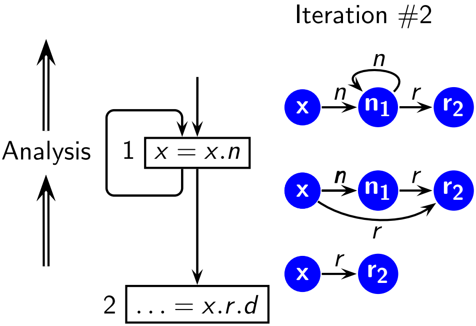
Inclusion of Program Point Facilitates Summarization



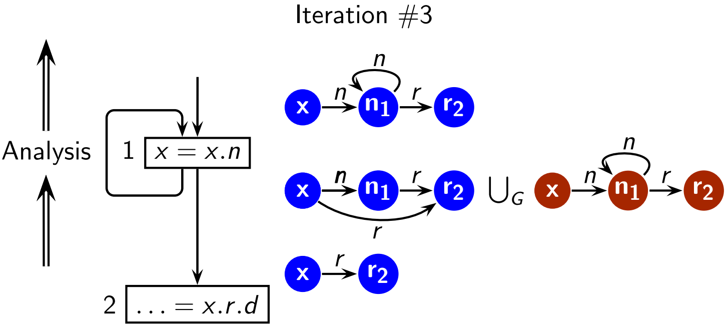
Inclusion of Program Point Facilitates Summarization



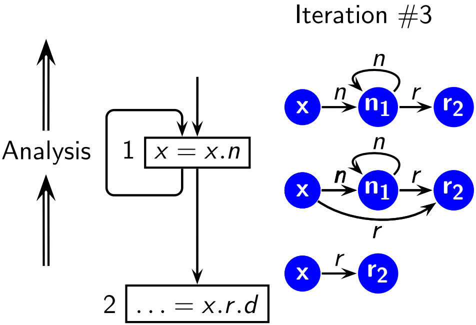
Inclusion of Program Point Facilitates Summarization



Inclusion of Program Point Facilitates Summarization

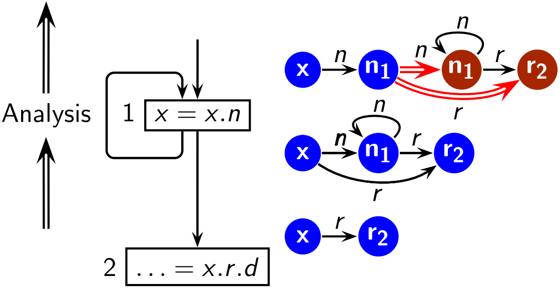


Inclusion of Program Point Facilitates Summarization

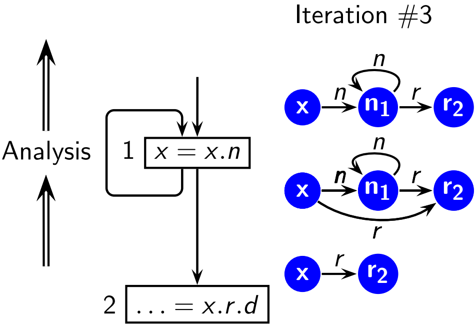


Inclusion of Program Point Facilitates Summarization

Iteration #3



Inclusion of Program Point Facilitates Summarization



Access Graph and Memory Graph

Program Fragment

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

$\downarrow$

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

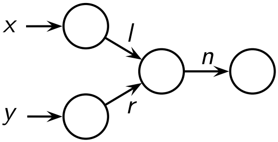
Access Graph and Memory Graph

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

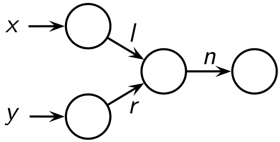
Program Fragment

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

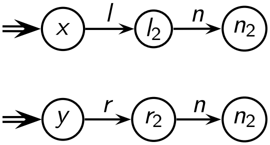
$\downarrow$

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

Memory Graph



Access Graphs



Access Graph and Memory Graph

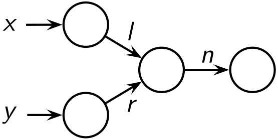
Program Fragment

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

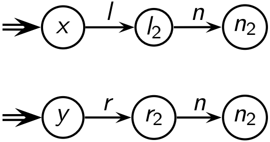
$\downarrow$

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

Memory Graph

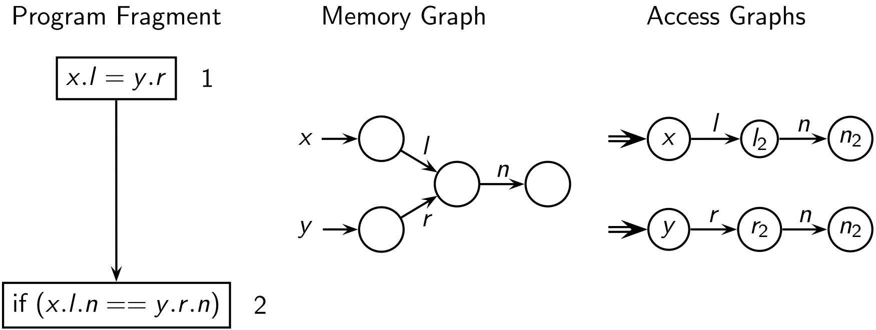


Access Graphs



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

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.

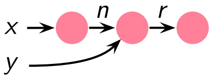
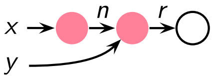
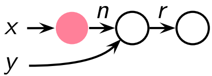
Access Graph Operations

- *Union*. $G \uplus G'$
- *Path Removal*

  $G \ominus R$ removes those access paths in G which have $\rho \in R$ as a prefix
- *Factorization* $(/)$
- *Extension*

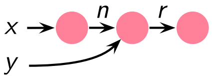
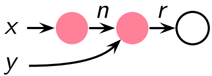
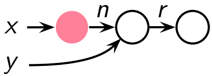
Defining Factorization

Given statement $x.n = y$, what should be the result of transfer?

Live AP	Memory Graph	Transfer	Remainder
$x \to n \to r$		$y \to r$	r (LHS is contained in the live access path)
$x \to n$		y	ϵ (LHS is contained in the live access path)
x		no transfer	?? (LHS is not contained in the live access path)

Defining Factorization

Given statement $x.n = y$, what should be the result of transfer?

Live AP	Memory Graph	Transfer	Remainder
$x \rightarrow n \rightarrow r$		$y \rightarrow r$	r (LHS is contained in the live access path)
$x \rightarrow n$		y	ϵ (LHS is contained in the live access path)
x		no transfer	?? (LHS is not contained in the live access path) Quotient is empty So no remainder

Semantics of Access Graph Operations

- $P(G)$ is the set of all paths in graph G
- $P(G, M)$ is the set of paths in G terminaing on nodes in M
- S is the set of remainder graphs
- $P(S)$ is the set of all paths in all remainder graphs in S

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

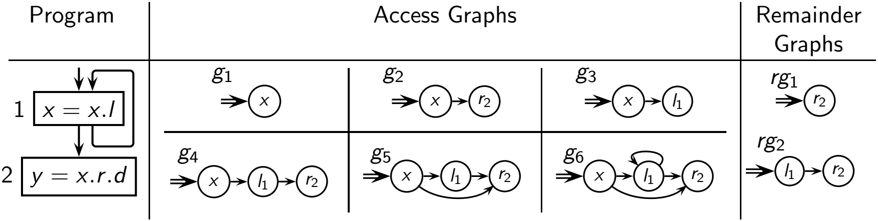
Semantics of Access Graph Operations

- $P\,(G)$ is the set of all paths in graph G
- $P\,(G, M)$ is the set of paths in G terminaing on nodes in M
- S is the set of remainder graphs
- $P\,(S)$ is the set of all paths in all remainder graphs in S

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

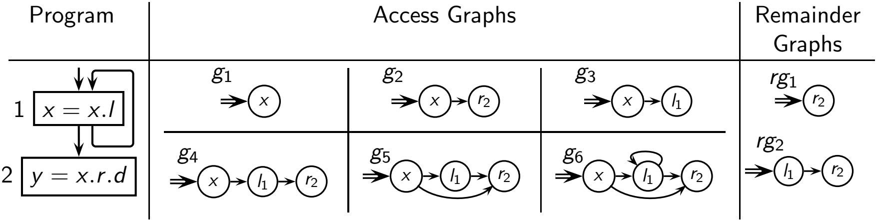
σ represents remainder

Access Graph Operations: Examples



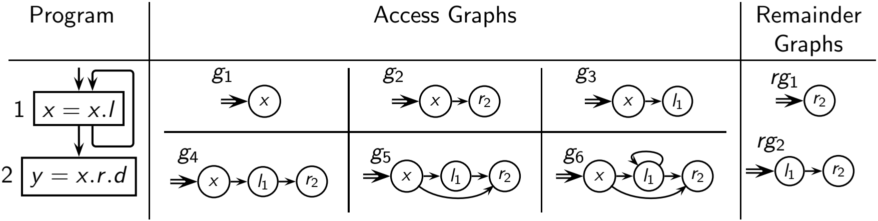
Union	Path Removal	Factorisation	Extension

Access Graph Operations: Examples



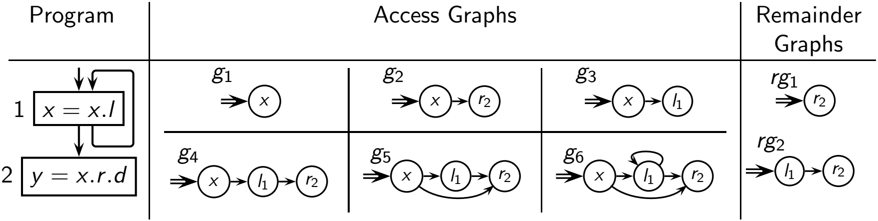
Union	Path Removal	Factorisation	Extension
$g_3 \uplus g_4 = g_4$ $g_2 \uplus g_4 = g_5$ $g_5 \uplus g_4 = g_5$ $g_5 \uplus g_6 = g_6$			

Access Graph Operations: Examples



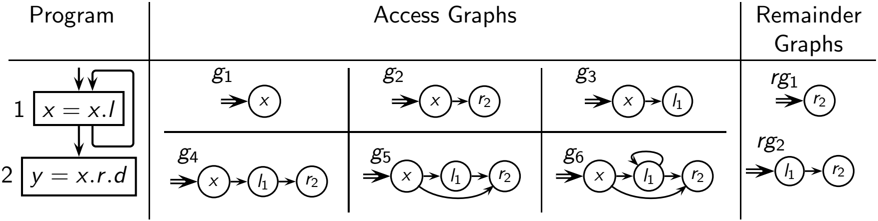
Union	Path Removal	Factorisation	Extension
$g_3 \uplus g_4 = g_4$ $g_2 \uplus g_4 = g_5$ $g_5 \uplus g_4 = g_5$ $g_5 \uplus g_6 = g_6$	$g_6 \ominus \{x \rightarrow l\} = g_2$ $g_5 \ominus \{x\} = \mathcal{E}_G$ $g_4 \ominus \{x \rightarrow r\} = g_4$ $g_4 \ominus \{x \rightarrow l\} = g_1$		

Access Graph Operations: Examples



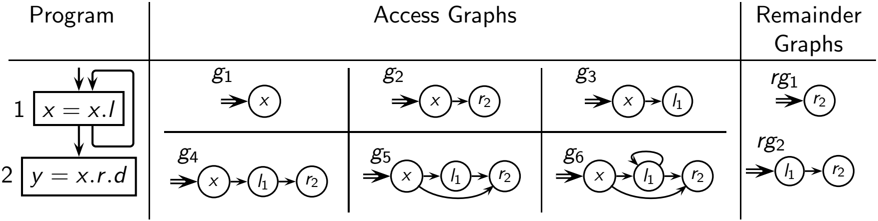
Union	Path Removal	Factorisation	Extension
$g_3 \uplus g_4 = g_4$	$g_6 \ominus \{x \rightarrow l\} = g_2$	$g_2/x = \{rg_1\}$	
$g_2 \uplus g_4 = g_5$	$g_5 \ominus \{x\} = \mathcal{E}_G$	$g_5/x = \{rg_1, rg_2\}$	
$g_5 \uplus g_4 = g_5$	$g_4 \ominus \{x \rightarrow r\} = g_4$	$g_5/x \rightarrow r = \{\epsilon_{RG}\}$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

Access Graph Operations: Examples



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

Access Graph Operations: Examples



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

Remainder is empty

Quotient is empty

Data Flow Equations for Explicit Liveness Analysis: Access Graphs Version

$$In_n = \left(Out_n \ominus \mathsf{Kill}_n(Out_n)\right) \uplus \mathsf{Gen}_n(Out_n)$$

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

- In_n, Out_n, and Gen_n are access graphs
- Kill_n is a set of access paths

Flow Functions for Explicit Liveness Analysis: Access Paths Version

Let A denote May Aliases at the exit of node n

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
$x = y$	$\{y \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x = y.f$	$\{y \rightarrow f \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x.f = y$	$\left\{ y \rightarrow \sigma \mid z \rightarrow f \rightarrow \sigma \in X, z \in A(x) \right\}$	$\bigcup\limits_{z \in Must(A)(x)} z \rightarrow f \rightarrow *$
$x = new$	$\emptyset$	$x \rightarrow *$
$x = null$	$\emptyset$	$x \rightarrow *$
other	$\emptyset$	$\emptyset$

Flow Functions for Explicit Liveness Analysis: Access Paths Version

Let A denote May Aliases at the exit of node n

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
$x = y$	$\{y \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x = y.f$	$\{y \rightarrow f \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x.f = y$	$\left\{y \rightarrow \sigma \;\middle\vert\; \boxed{z \rightarrow f \rightarrow \sigma \in X, z \in A(x)}\right\}$	$\bigcup_{z \in Must(A)(x)} z \rightarrow f \rightarrow *$
$x = new$	$\emptyset$	$x \rightarrow *$
$x = null$	$\emptyset$	$x \rightarrow *$
other	$\emptyset$	$\emptyset$

May link aliasing for soundness

Flow Functions for Explicit Liveness Analysis: Access Paths Version

Let A denote May Aliases at the exit of node n

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
$x = y$	$\{y \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x = y.f$	$\{y \rightarrow f \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x.f = y$	$\left\{y \rightarrow \sigma \;\middle\vert\; \boxed{z \rightarrow f \rightarrow \sigma \in X, z \in A(x)}\right\}$	$\boxed{\bigcup_{z \in Must(A)(x)} z \rightarrow f \rightarrow *}$
$x = new$	$\emptyset$	$x \rightarrow *$
$x = null$	$\emptyset$	$x \rightarrow *$
other	$\emptyset$	$\emptyset$

May link aliasing for soundness

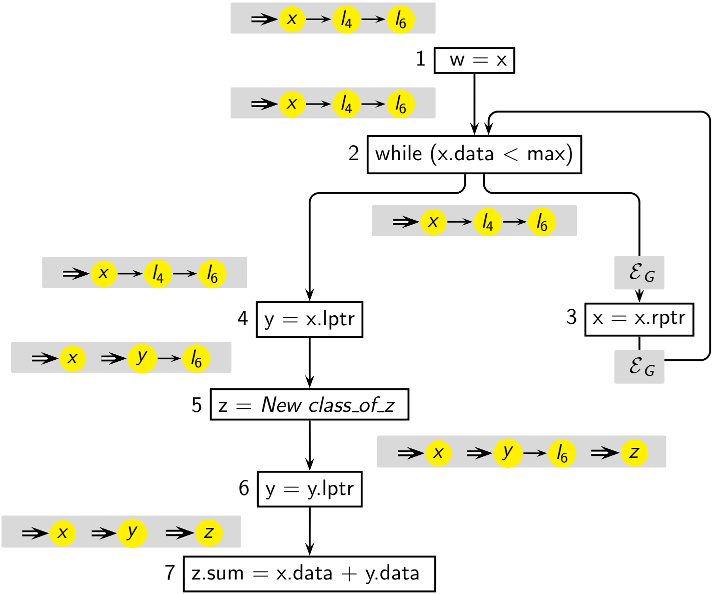
Must link aliasing for precision

Flow Functions for Explicit Liveness Analysis: Access Graphs Version

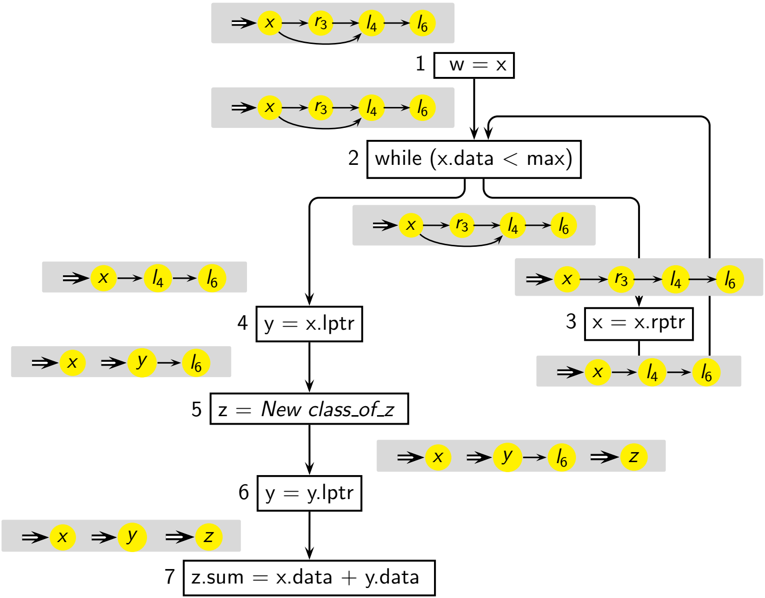
- A denotes May Aliases at the exit of node n
- $mkGraph(\rho)$ creates an access graph for access path ρ

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
$x = y$	$mkGraph(y)\#(X/x)$	$\{x\}$
$x = y.f$	$mkGraph(y \rightarrow f)\#(X/x)$	$\{x\}$
$x.f = y$	$mkGraph(y)\#\left(\bigcup_{z \in A(x)} (X/(z \rightarrow f))\right)$	$\{z \rightarrow f \mid z \in Must(A)(x)\}$
$x = new$	$\emptyset$	$\{x\}$
$x = null$	$\emptyset$	$\{x\}$
other	$\emptyset$	$\emptyset$

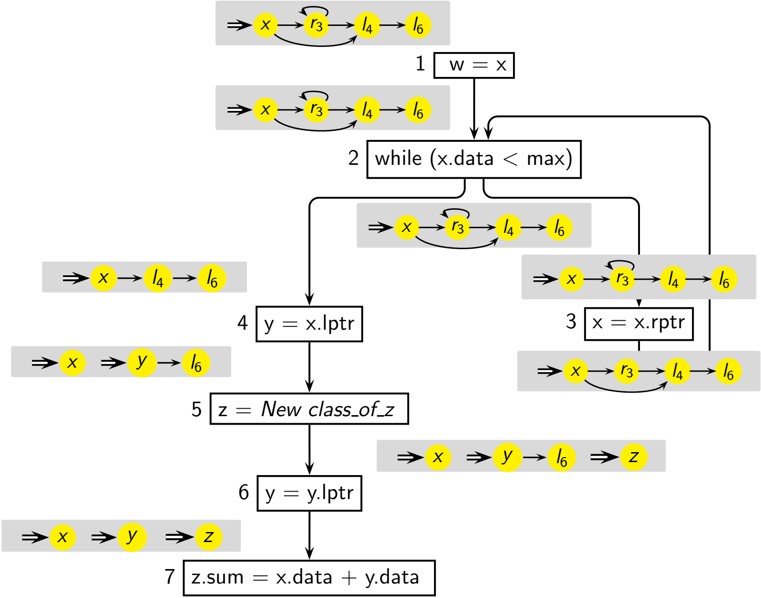
Liveness Analysis of Example Program: Ist Iteration



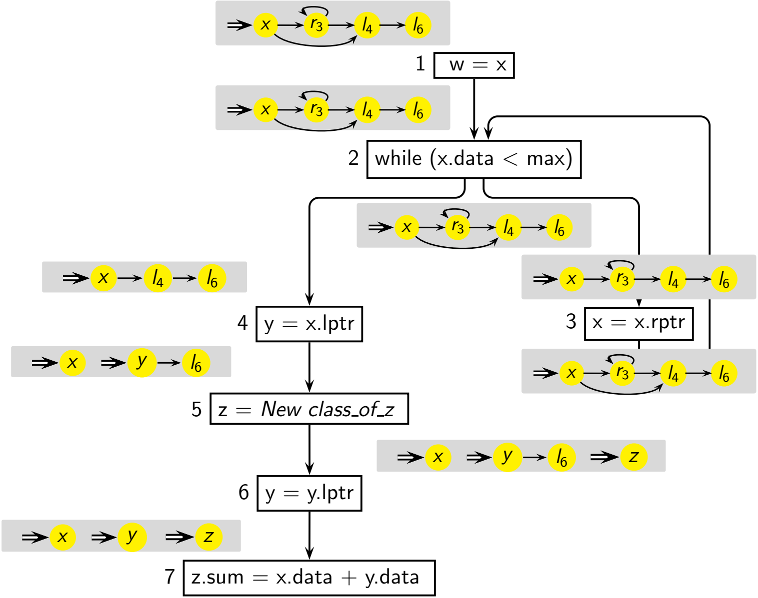
Liveness Analysis of Example Program: 2nd Iteration



Liveness Analysis of Example Program: 3rd Iteration



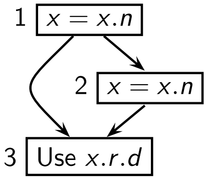
Liveness Analysis of Example Program: 4th Iteration



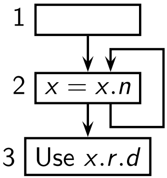
Tutorial Problem for Explicit Liveness (1)

Construct access graphs at the entry of block 1 for the following programs

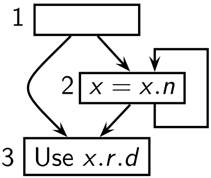
A



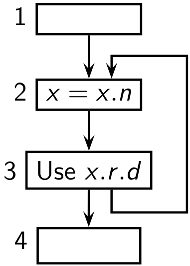
B



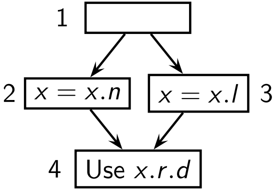
C



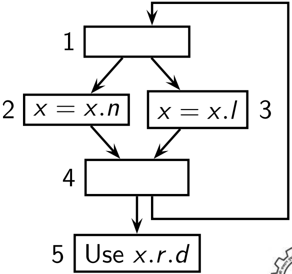
D



E

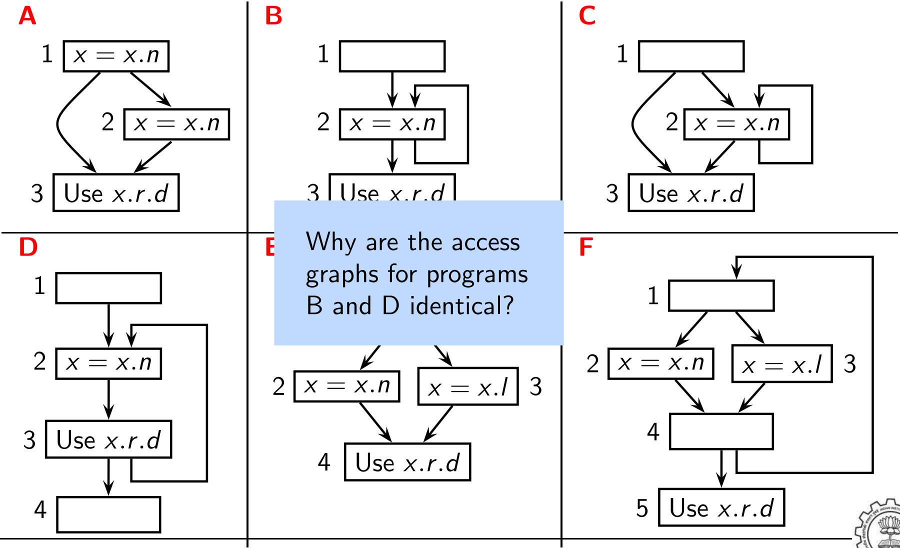


F



Tutorial Problem for Explicit Liveness (1)

Construct access graphs at the entry of block 1 for the following programs

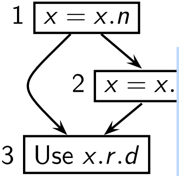


Why are the access graphs for programs B and D identical?

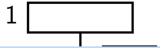
Tutorial Problem for Explicit Liveness (1)

Construct access graphs at the entry of block 1 for the following programs

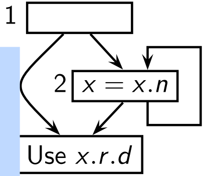
A



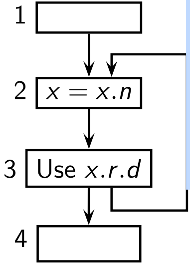
B

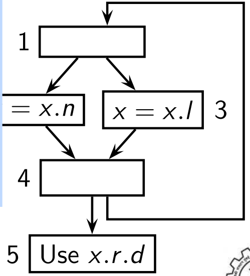


C



D



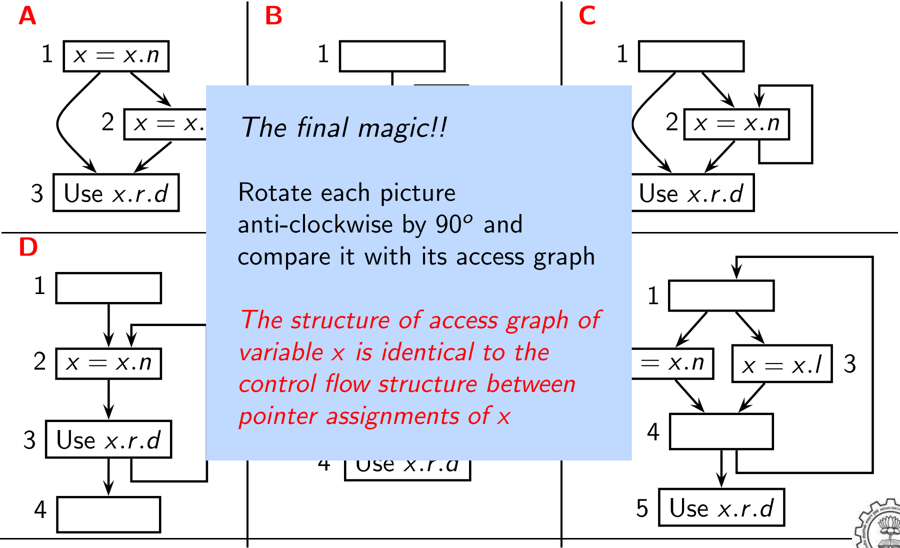


The final magic!!

Rotate each picture anti-clockwise by 90^o and compare it with its access graph

Tutorial Problem for Explicit Liveness (1)

Construct access graphs at the entry of block 1 for the following programs



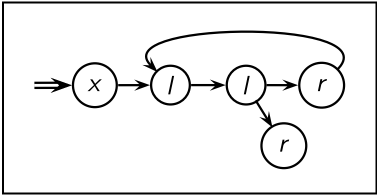
The final magic!!

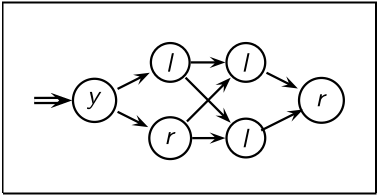
Rotate each picture anti-clockwise by 90^o and compare it with its access graph

The structure of access graph of variable x is identical to the control flow structure between pointer assignments of x

Tutorial Problem for Explicit Liveness (2)

- Unfortunately the student who constructed these access graphs forgot to attach statement numbers as subscripts to node labels and has misplaced the programs which gave rise to these graphs
- Please help her by constructing CFGs for which these access graphs represent explicit liveness at some program point in the CFGs

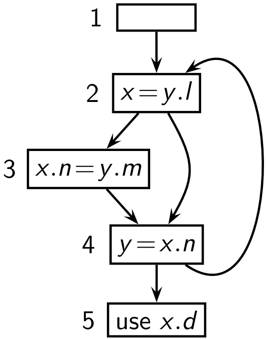




Tutorial Problem for Explicit Liveness (3)

- Compute explicit liveness for the program.
- Are the following access paths live at node 1? Show the corresponding execution sequence of statements

P1 : $y \rightarrow m \rightarrow l$
P2 : $y \rightarrow l \rightarrow n \rightarrow m$
P3 : $y \rightarrow l \rightarrow n \rightarrow l$
P4 : $y \rightarrow n \rightarrow l \rightarrow n$



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 $\rho \rightarrow f =$ 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 $\rho \rightarrow f$ = 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.

Transfer in Availability and Anticipability Analysis

The essential idea of the transfer of access paths remains same

- Transfer in Availability Analysis is from the RHS to the LHS

$\rho_r \rightarrow \sigma$ available here

$\boxed{\rho_l = \rho_r}$

$\rho_l \rightarrow \sigma$ available here

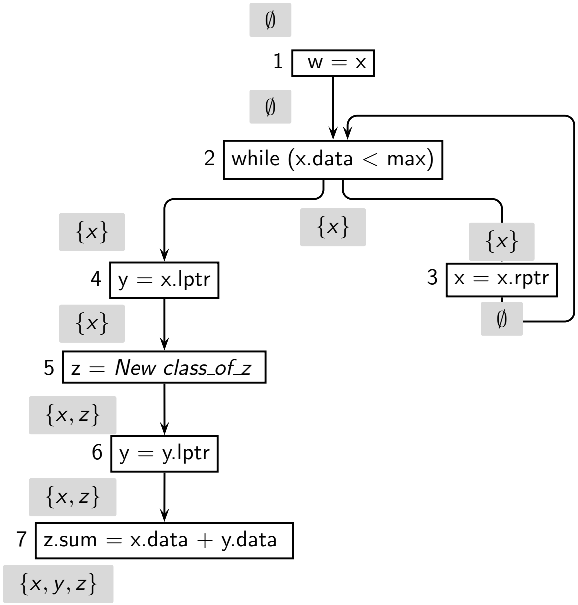
- Transfer in Anticipability Analysis is from the LHS to the RHS

$\rho_r \rightarrow \sigma$ anticipable here

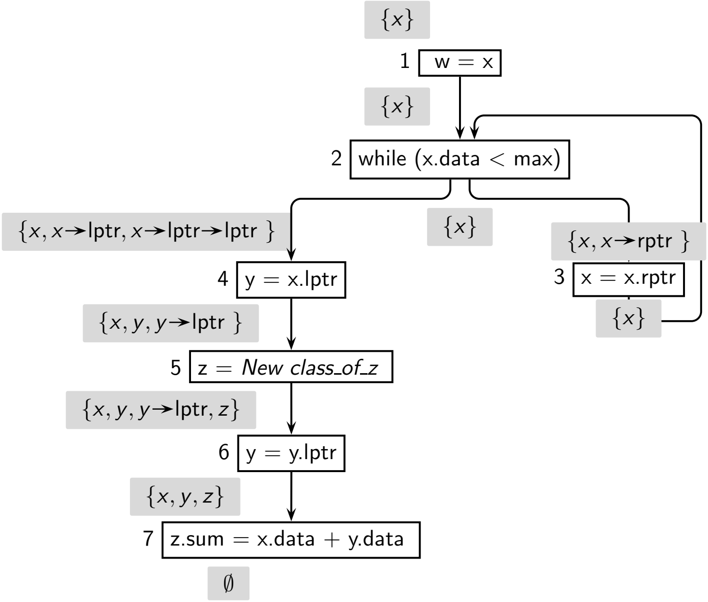
$\boxed{\rho_l = \rho_r}$

$\rho_l \rightarrow \sigma$ anticipable here

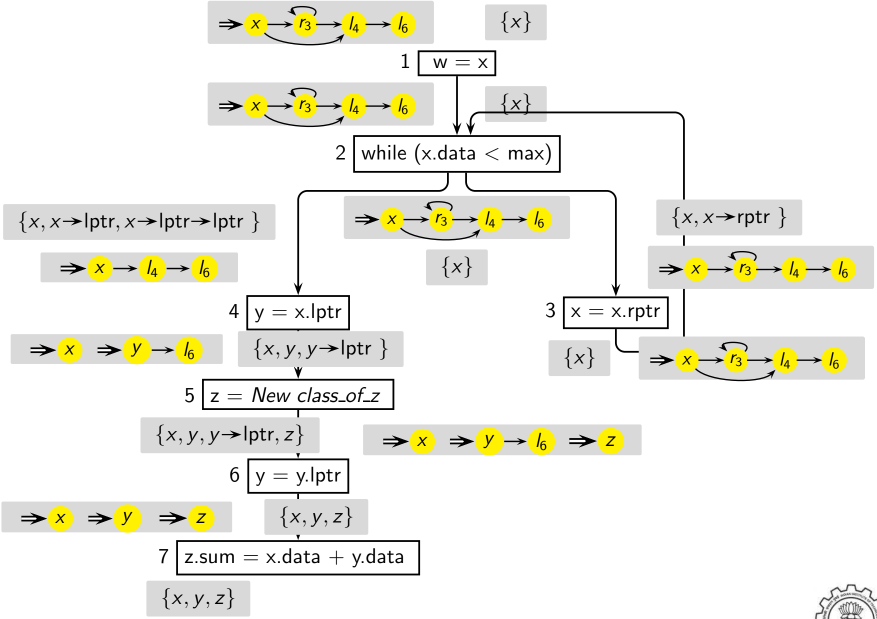
Availability Analysis of Example Program



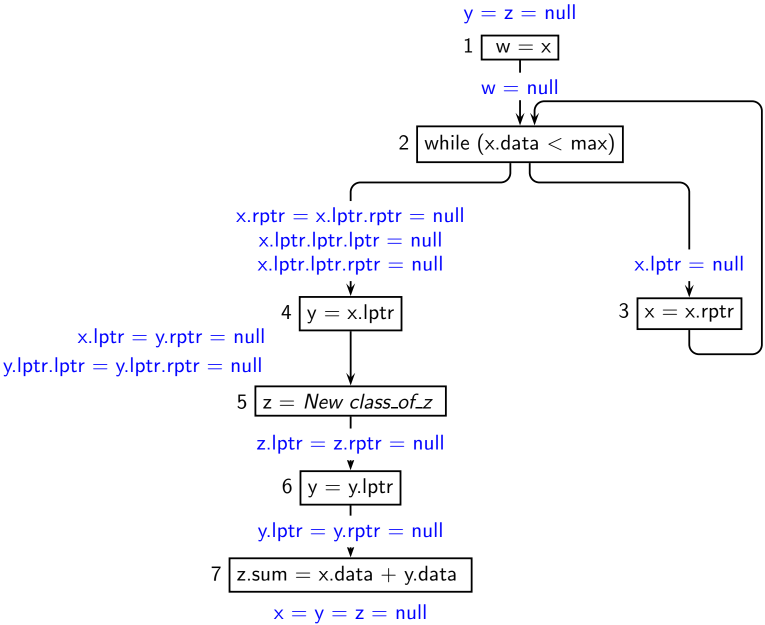
Anticipability Analysis of Example Program



Live and Accessible Paths



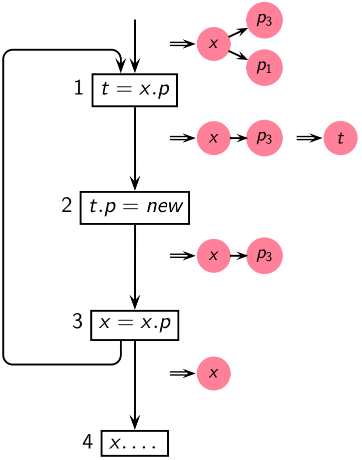
Creating null Assignments from Live and Accessible Paths



The Resulting Program

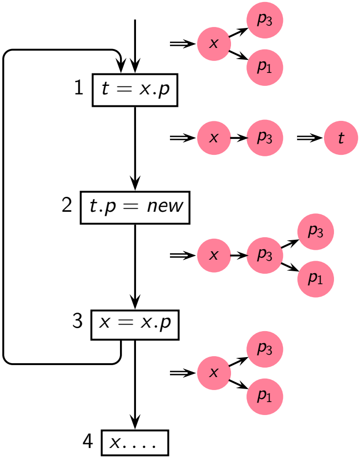
```
                                  y = z = null
w = x
                                  w = null
while (x.data < max)
{                                 x.lptr = null
    x = x.rptr       }
                                  x.rptr = x.lptr.rptr = null
                                  x.lptr.lptr.lptr = 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
```

Overapproximation Caused by Our Summarization



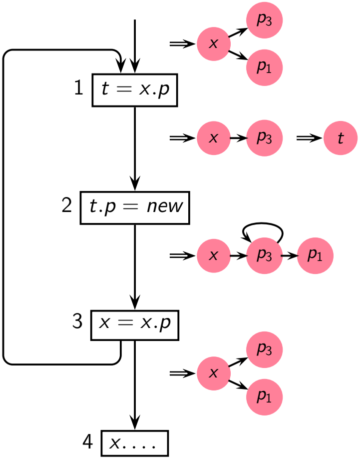
- The program allocates $x \rightarrow p$ in one iteration and uses it in the next

Overapproximation Caused by Our Summarization



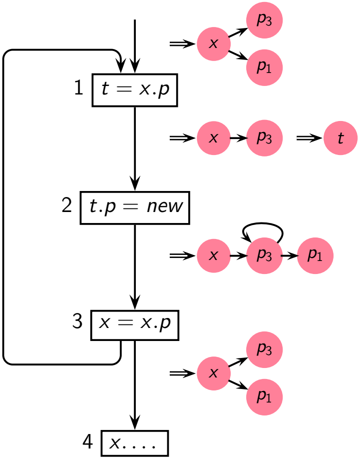
- The program allocates $x \rightarrow p$ in one iteration and uses it in the next

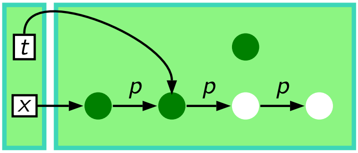
Overapproximation Caused by Our Summarization



- The program allocates $x \rightarrow p$ in one iteration and uses it in the next
- *Only* $x \rightarrow p \rightarrow p$ *is live at* Out_2

Overapproximation Caused by Our Summarization

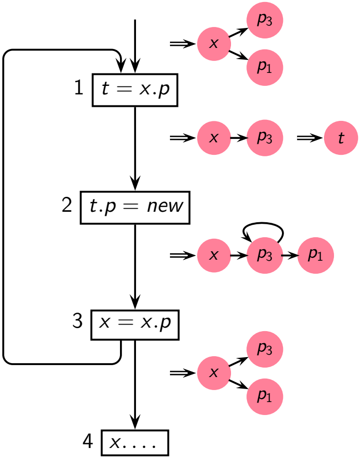


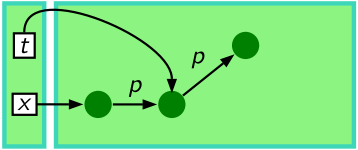


Out_1

- The program allocates $x \rightarrow p$ in one iteration and uses it in the next
- *Only $x \rightarrow p \rightarrow p$ is live at* Out_2

Overapproximation Caused by Our Summarization





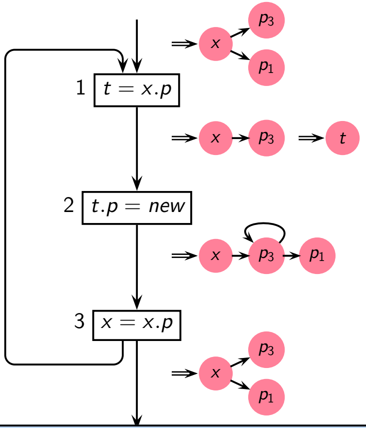
Out_2

- The program allocates $x \rightarrow p$ in one iteration and uses it in the next
- *Only $x \rightarrow p \rightarrow p$ is live at* Out_2
- $x \rightarrow p \rightarrow p$ is live at Out_2

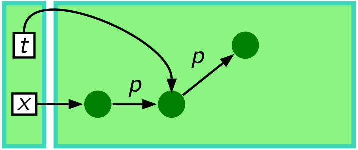
  $x \rightarrow p \rightarrow p \rightarrow p$ is dead at Out_2
- First p used in statement 3

  Second p used in statement 4
- Third p is reallocated

Overapproximation Caused by Our Summarization



Second occurrence of a dereference does not necessarily mean an unbounded number of repetitions!



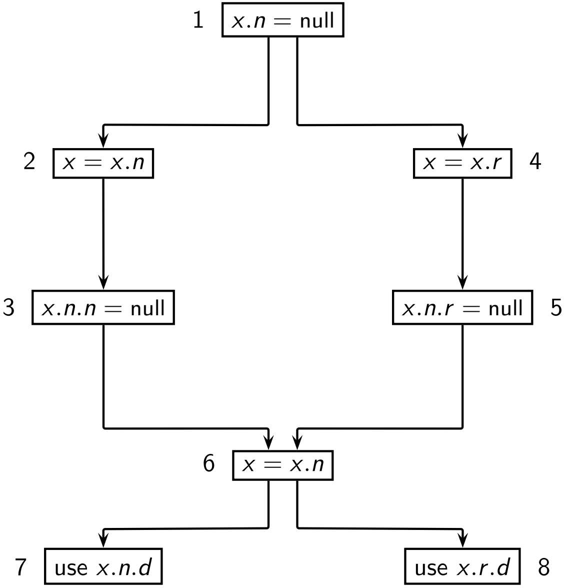
Out_2

- The program allocates $x \rightarrow p$ in one iteration and uses it in the next
- *Only $x \rightarrow p \rightarrow p$ is live at* Out_2
- $x \rightarrow p \rightarrow p$ is live at Out_2

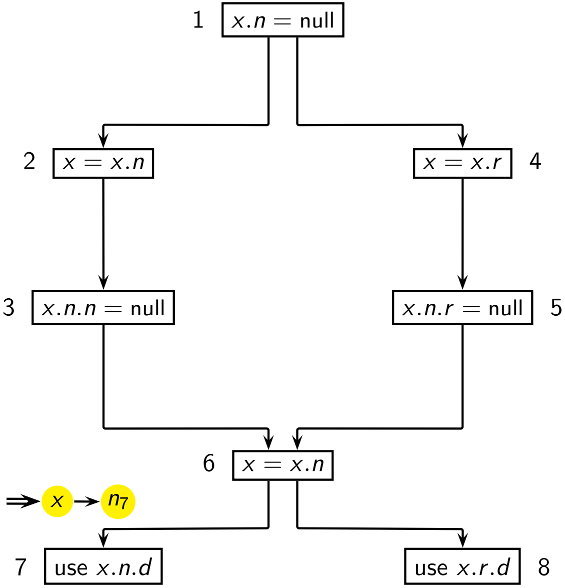
  $x \rightarrow p \rightarrow p \rightarrow p$ is dead at Out_2
- First p used in statement 3

  Second p used in statement 4
- Third p is reallocated

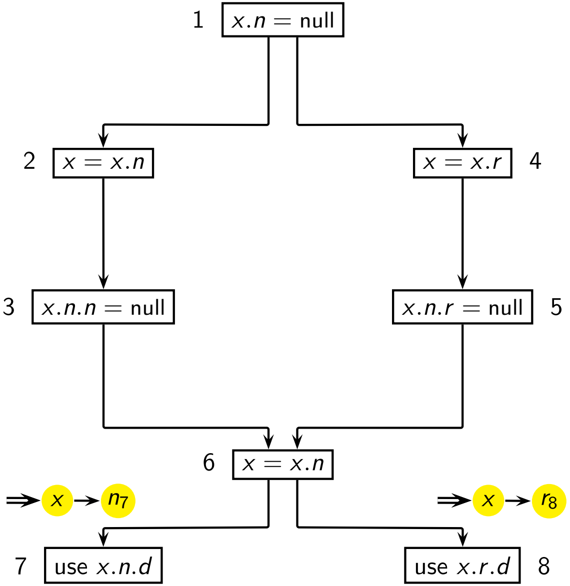
Non-Distributivity of Explicit Liveness Analysis



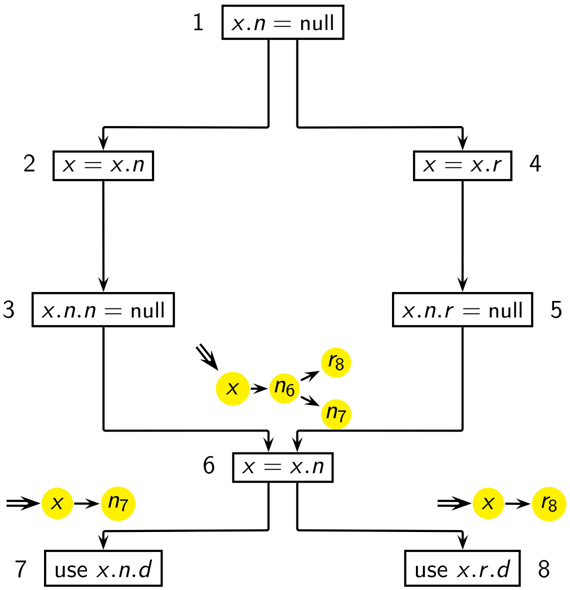
Non-Distributivity of Explicit Liveness Analysis



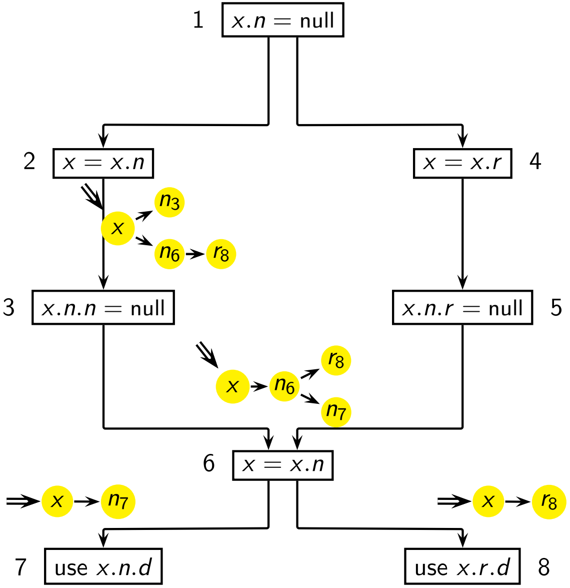
Non-Distributivity of Explicit Liveness Analysis



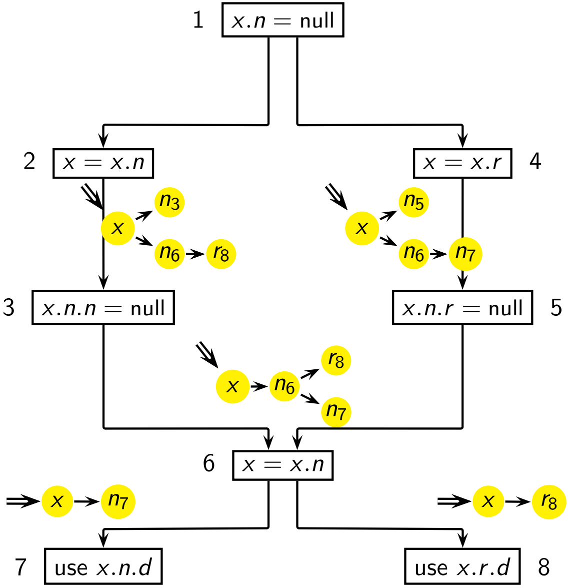
Non-Distributivity of Explicit Liveness Analysis



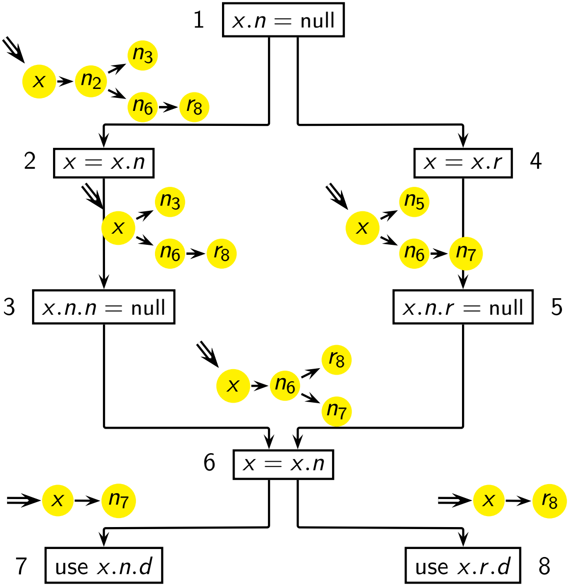
Non-Distributivity of Explicit Liveness Analysis



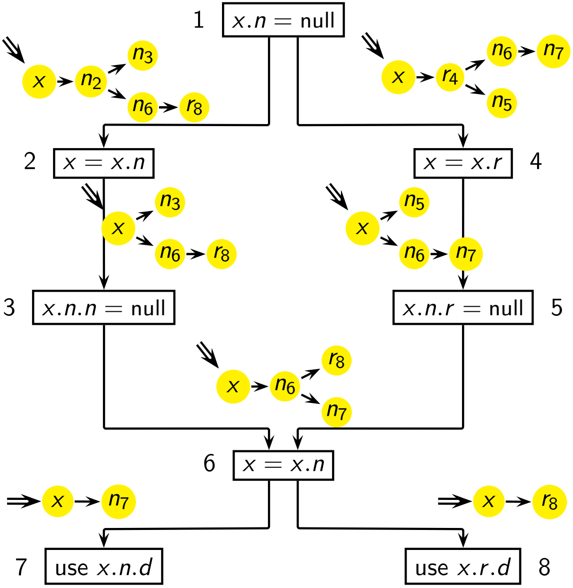
Non-Distributivity of Explicit Liveness Analysis



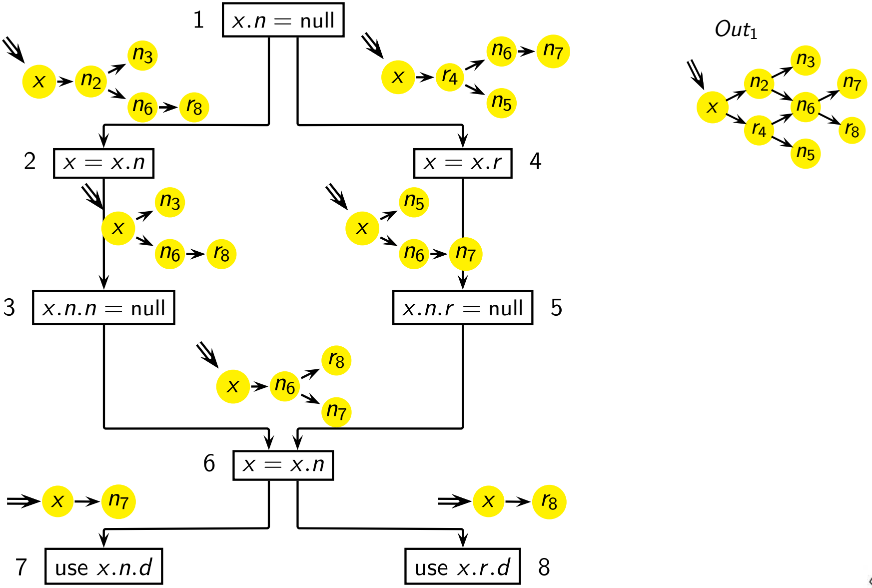
Non-Distributivity of Explicit Liveness Analysis



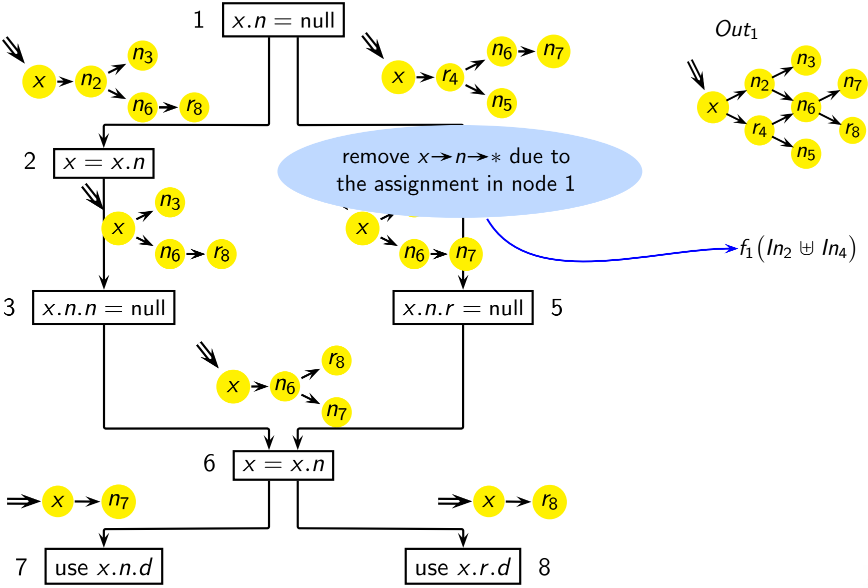
Non-Distributivity of Explicit Liveness Analysis



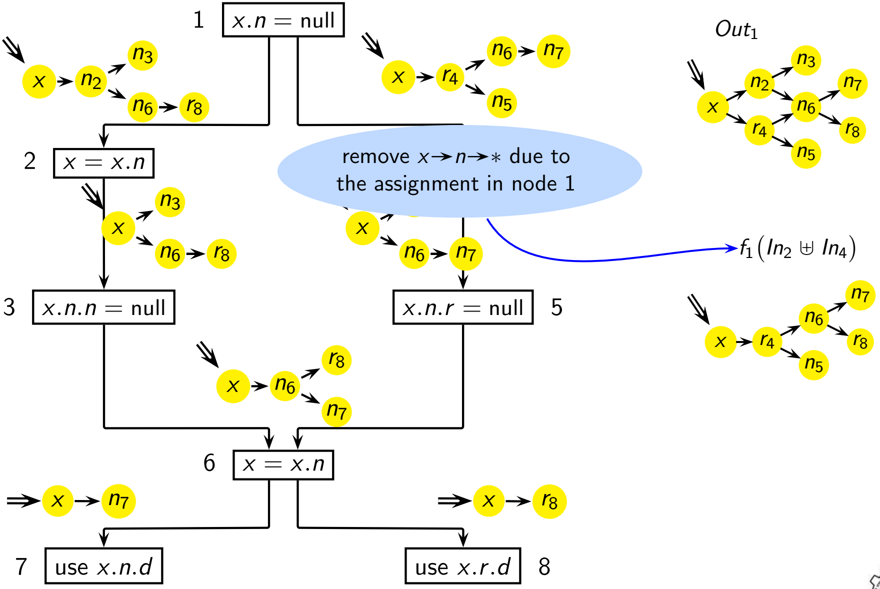
Non-Distributivity of Explicit Liveness Analysis



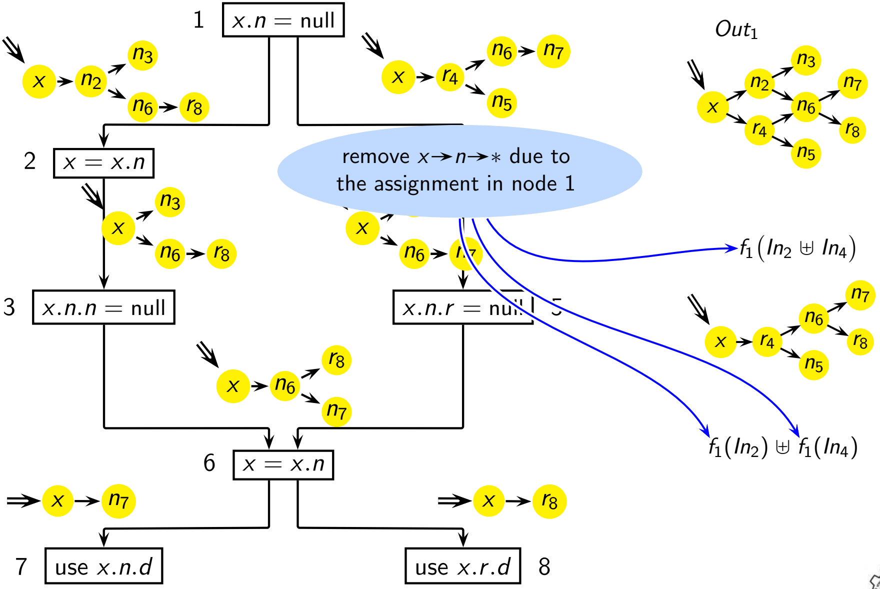
Non-Distributivity of Explicit Liveness Analysis



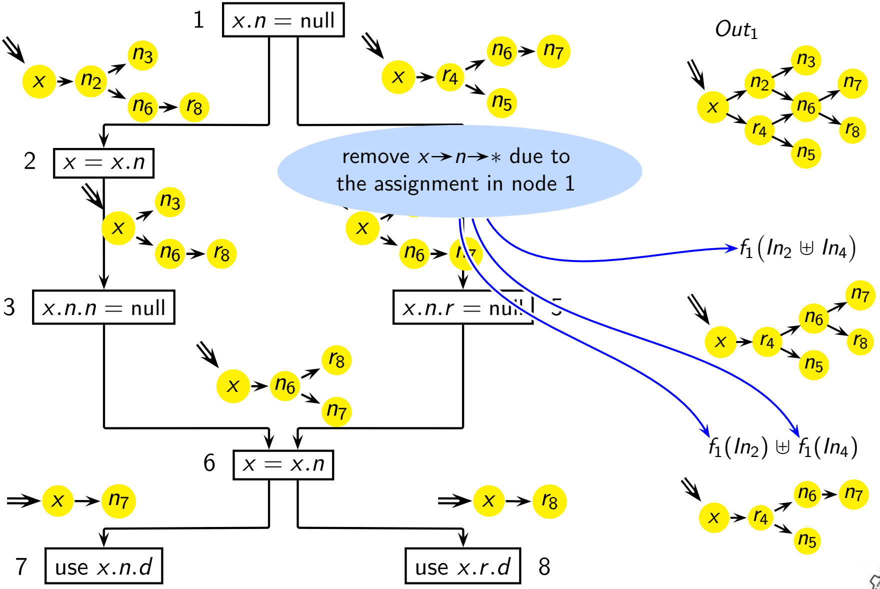
Non-Distributivity of Explicit Liveness Analysis



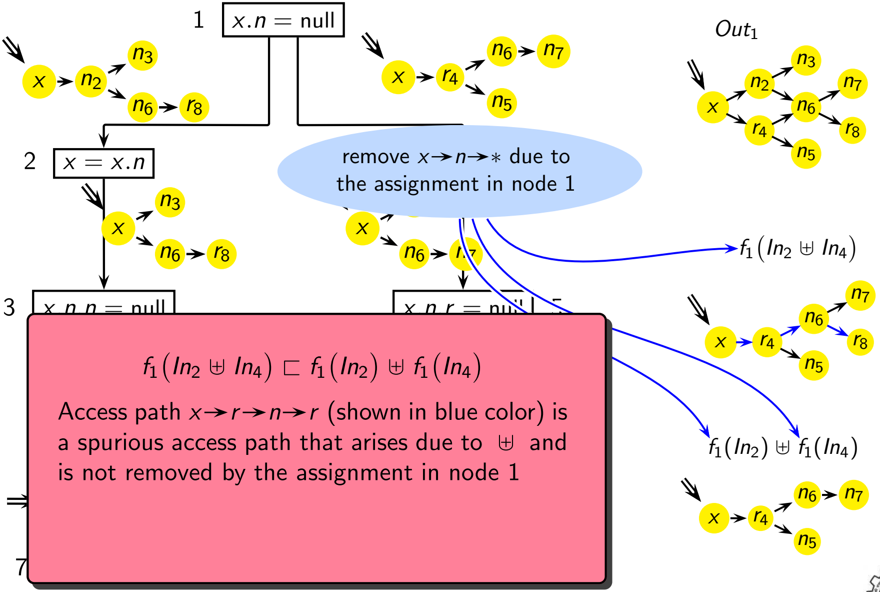
Non-Distributivity of Explicit Liveness Analysis



Non-Distributivity of Explicit Liveness Analysis



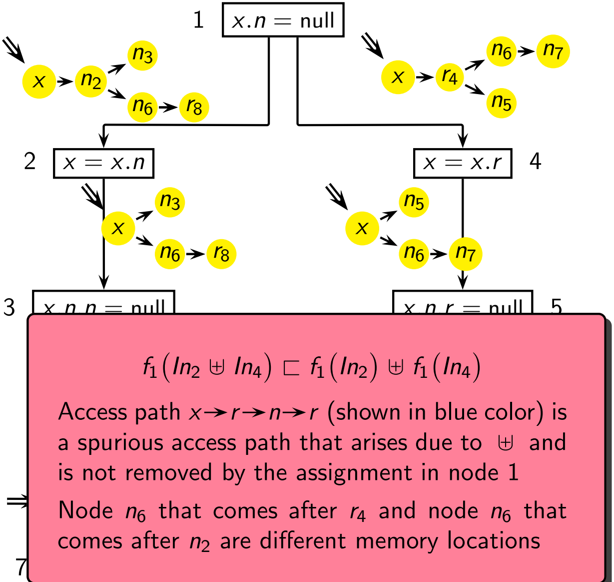
Non-Distributivity of Explicit Liveness Analysis

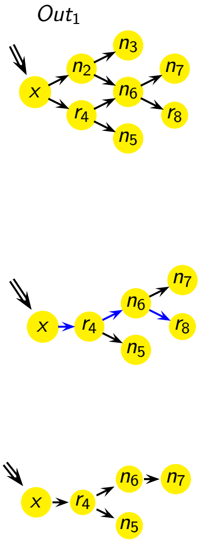


$$f_1\left(In_2 \uplus In_4\right) \sqsubset f_1\left(In_2\right) \uplus f_1\left(In_4\right)$$

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

Non-Distributivity of Explicit Liveness Analysis





$$f_1\left(In_2 \uplus In_4\right) \sqsubset f_1\left(In_2\right) \uplus f_1\left(In_4\right)$$

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

Node n_6 that comes after r_4 and node n_6 that comes after n_2 are different memory locations

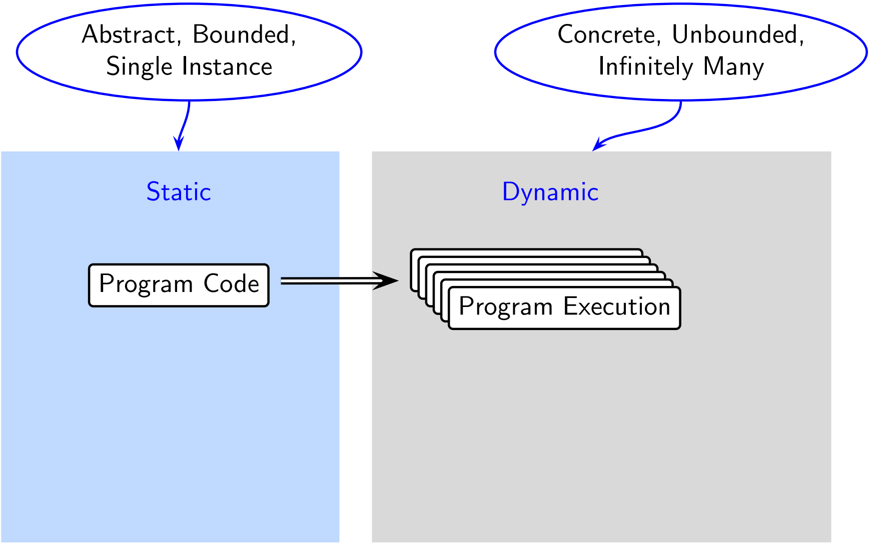
Issues Not Covered

- Precision of information
  - Cyclic Data Structures
  - Eliminating Redundant null Assignments
- Properties of Data Flow Analysis:
  Monotonicity, Boundedness, Complexity
- Interprocedural Analysis
- Extensions for C/C++
- Formulation for functional languages
- Issues that need to be researched: Good alias analysis of heap

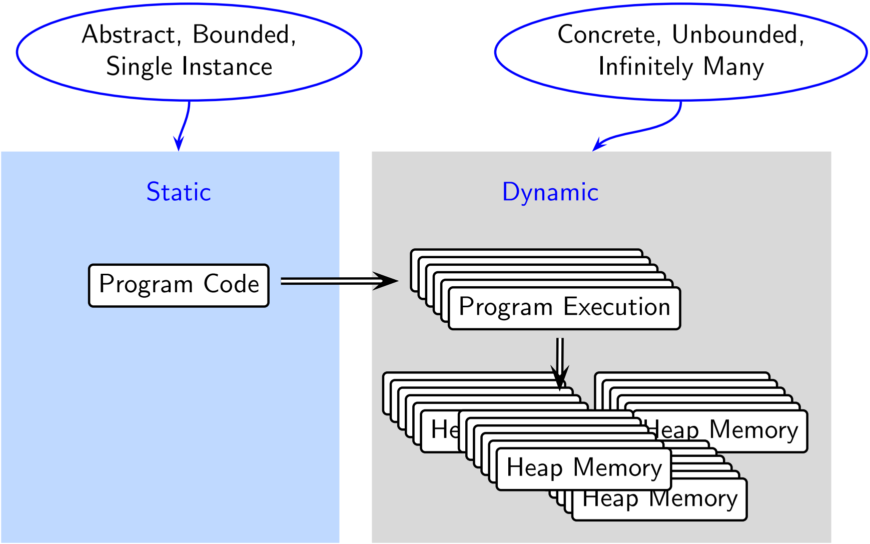
BTW, What is Static Analysis of Heap?

Static	Dynamic

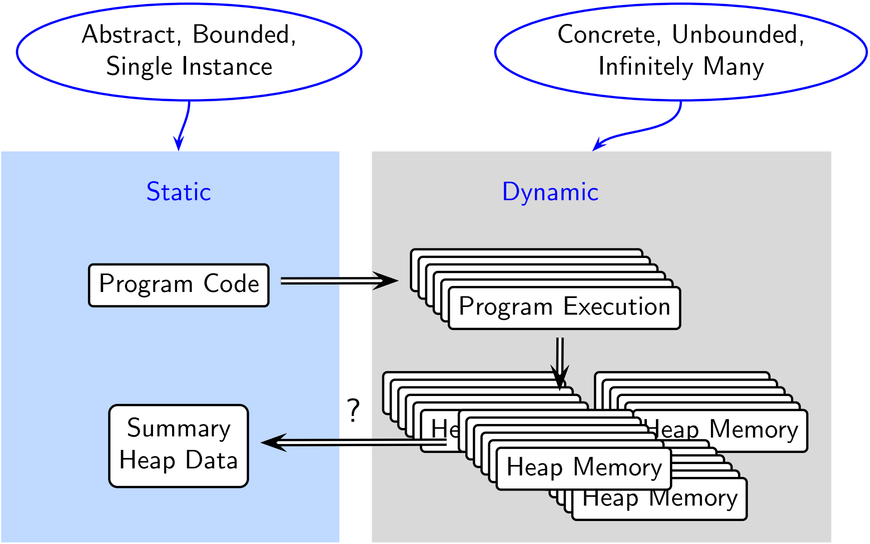
BTW, What is Static Analysis of Heap?



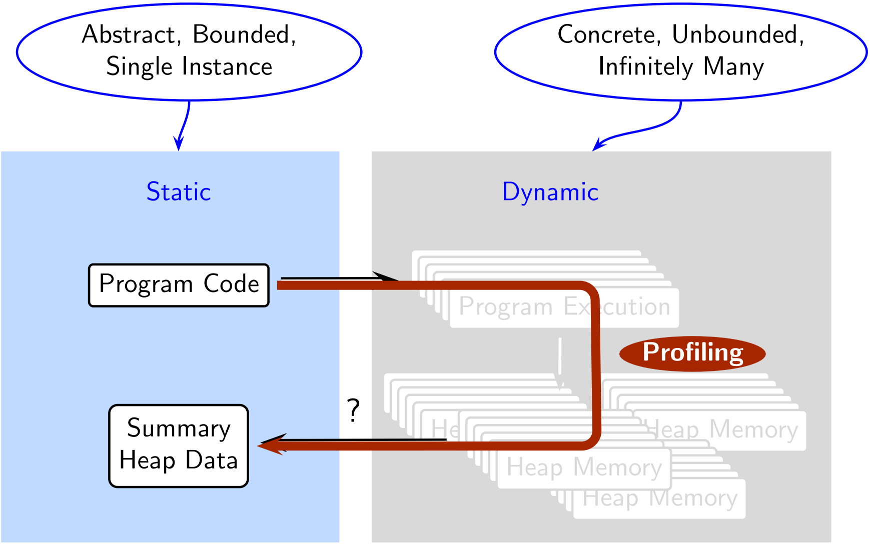
BTW, What is Static Analysis of Heap?



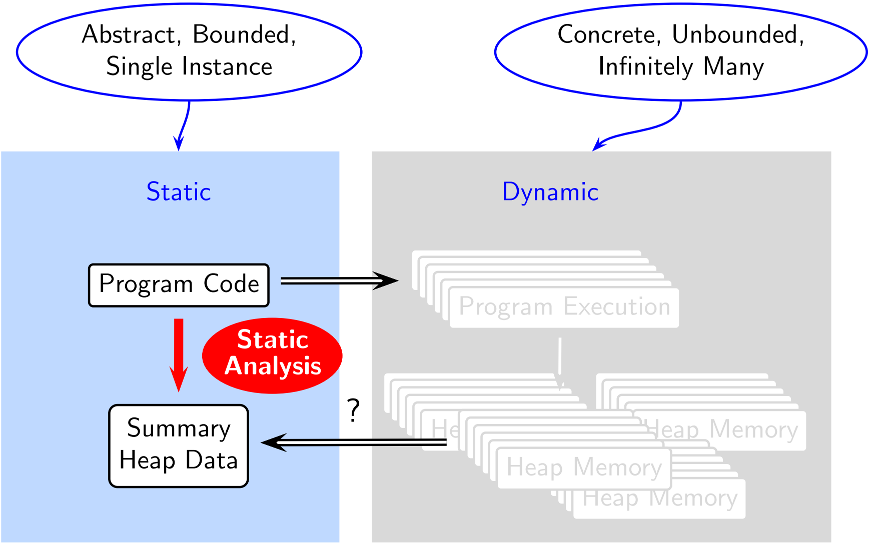
BTW, What is Static Analysis of Heap?



BTW, What is Static Analysis of Heap?



BTW, What is Static Analysis of Heap?



Conclusions

- Unbounded information can be summarized using interesting insights
  - Contrary to popular perception, heap structure is not arbitrary

    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