General Data Flow Frameworks

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

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

September 2017

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.

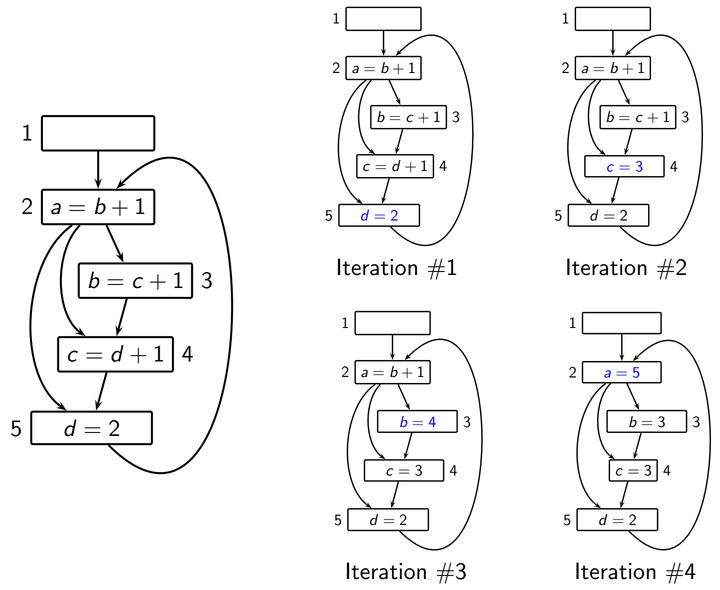
Outline

- 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?



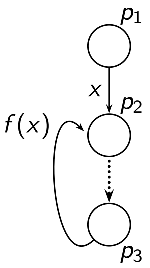
Iteration #1

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

- 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 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 ≡ 2-bounded frameworks (eg. bit vector frameworks)

  Both these conditions must be satisfied
  - *Separability*

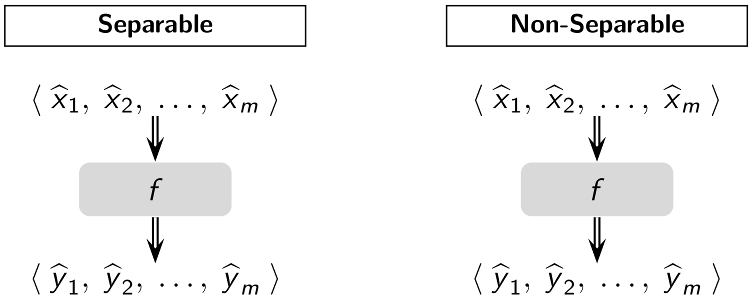
    Data flow values of different entities are independent
  - *Constant or Identity Flow Functions*

    Flow functions for an entity are either constant or identity
- Non-fast frameworks

  At least one of the above conditions is violated

Separability

$f : L \to L$ 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

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

f

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

Non-Separable

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

f

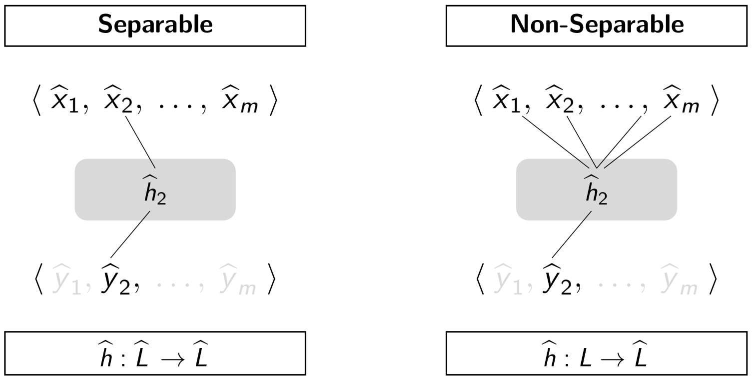
$\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$

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

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

$\widehat{h}_2$

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

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

Non-Separable

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

$\widehat{h}_2$

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

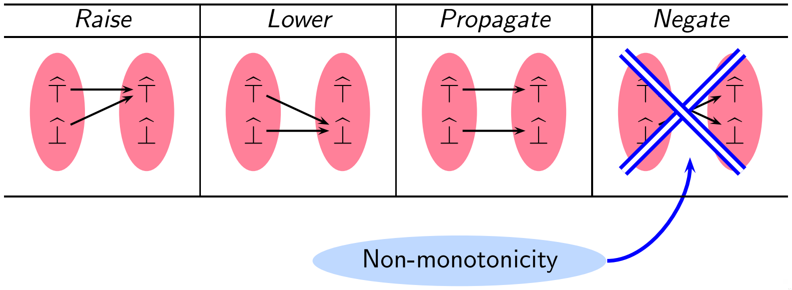
$\widehat{h} : L \to \widehat{L}$

Example: All bit vector frameworks

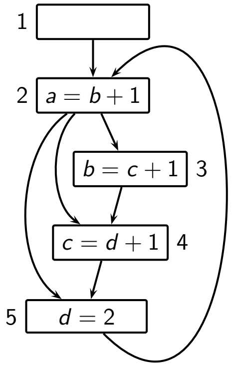
Example: Constant Propagation

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:



Larger Values of Loop Closure Bounds



Composite flow function for the loop is

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

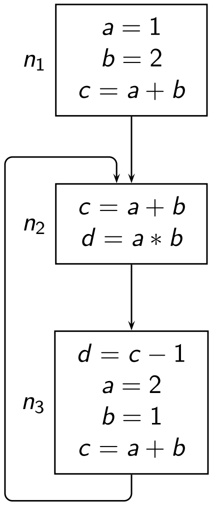
f is not 2-bounded because:

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

Part 3

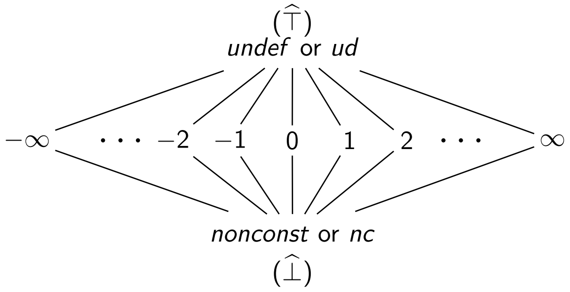
Constant Propagation

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



$(\widehat{\top})$

undef or *ud*

$-\infty \quad \cdots \quad -2 \quad -1 \quad 0 \quad 1 \quad 2 \quad \cdots \quad \infty$

nonconst or *nc*

$(\widehat{\bot})$

$\widehat{\sqcap}$	$\langle v, ud\rangle$	$\langle v, nc\rangle$	$\langle v, c_1\rangle$
$\langle v, ud\rangle$	$\langle v, ud\rangle$	$\langle v, nc\rangle$	$\langle v, c_1\rangle$
$\langle v, nc\rangle$	$\langle v, nc\rangle$	$\langle v, nc\rangle$	$\langle v, nc\rangle$
$\langle v, c_2\rangle$	$\langle v, c_2\rangle$	$\langle v, nc\rangle$	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

$$
\begin{array}{rrcl}
 & (x, v_1) \sqsubseteq (y, v_2) & \Leftrightarrow & x = y \wedge v_1 \widehat{\sqsubseteq} v_2 \\
OR & x \mapsto v_1 \sqsubseteq y \mapsto v_2 & \Leftrightarrow & x = y \wedge v_1 \widehat{\sqsubseteq} v_2
\end{array}
$$

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

$$
\begin{array}{rrl}
 & X \sqcap Y & = \{(x, v_1 \widehat{\sqcap} v_2) \mid (x, v_1) \in X, (x, v_2) \in Y\} \\
OR & X \sqcap Y & = \{x \mapsto v_1 \widehat{\sqcap} v_2 \mid x \mapsto v_1 \in X, x \mapsto v_2 \in Y\}
\end{array}
$$

Notations for Mappings as Data Flow Values

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

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

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

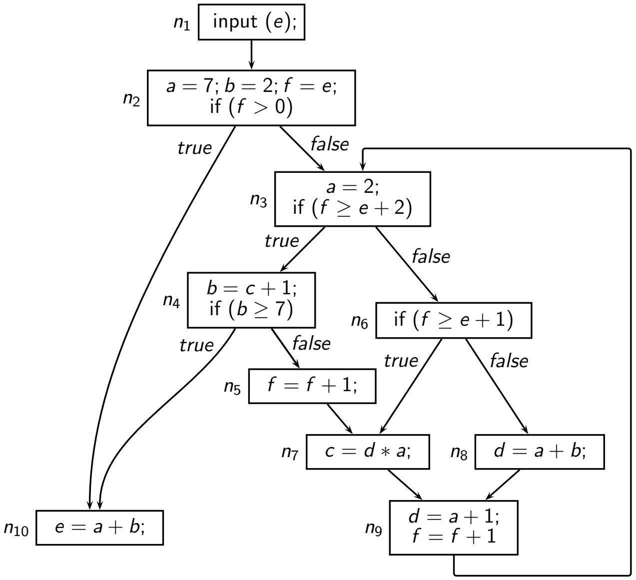
Defining Data Flow Equations for Constant Propagation

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

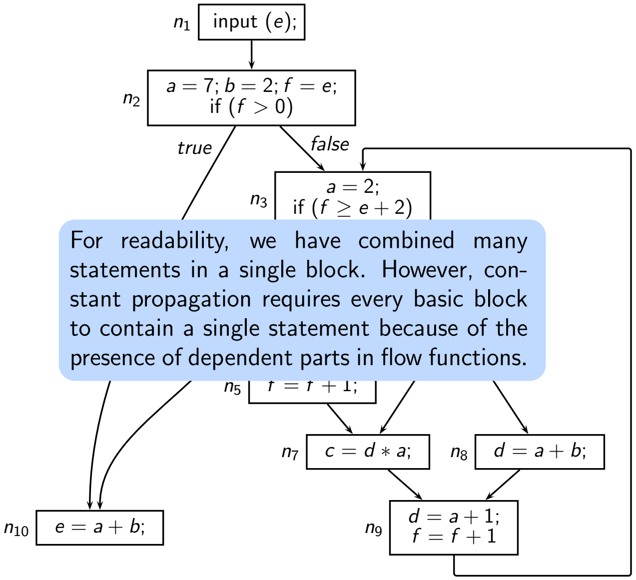
$$
f_n(X) = \begin{cases}
X\,[y \mapsto c] & n \text{ is } y = c, y \in \mathbb{V}\text{ar}, c \in \mathbb{C}\text{onst} \\
X\,[y \mapsto nc] & n \text{ is } input(y), y \in var \\
X\,[y \mapsto X(z)] & n \text{ is } y = z, y \in \mathbb{V}\text{ar}, z \in \mathbb{V}\text{ar} \\
X\,[y \mapsto eval(e,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

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

Example Program for Constant Propagation



Example Program for Constant Propagation

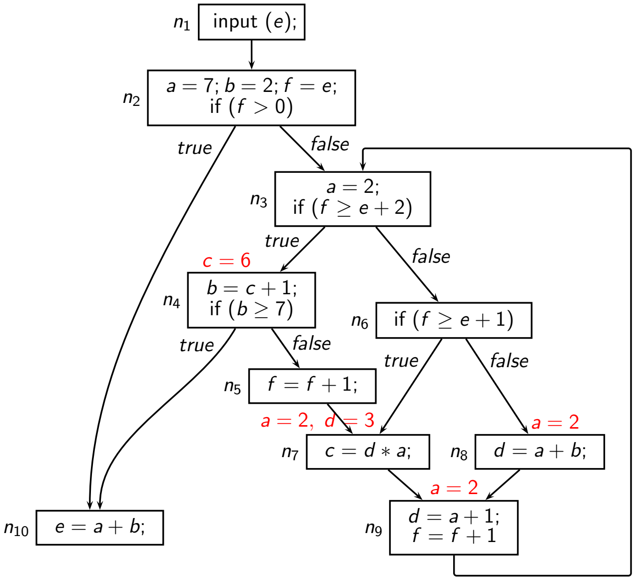


For readability, we have combined many statements in a single block. However, constant propagation requires every basic block to contain a single statement because of the presence of dependent parts in flow functions.

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



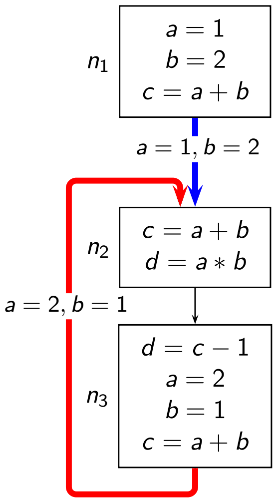
Monotonicity of Constant Propagation

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, C3, and C5
- 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



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

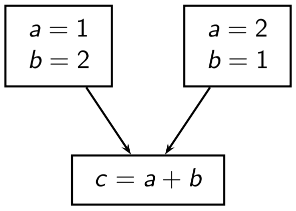
$$\begin{aligned} f(x) \sqcap f(y) &= f(\langle 1, 2, 3, ? \rangle) \sqcap f(\langle 2, 1, 3, 2 \rangle) \\ &= \langle 1, 2, 3, 2 \rangle \sqcap \langle 2, 1, 3, 2 \rangle \\ &= \langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle \end{aligned}$$

- Function application after merging

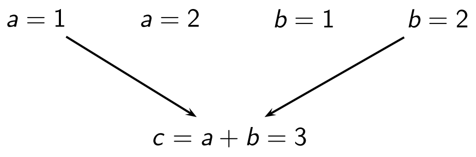
$$\begin{aligned} f(x \sqcap y) &= f(\langle 1, 2, 3, ? \rangle \sqcap \langle 2, 1, 3, 2 \rangle) \\ &= f(\langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle) \\ &= \langle \widehat{\bot}, \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle \end{aligned}$$

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

Why is Constant Propagation Non-Distributive?

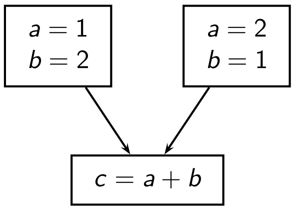


Possible combinations due to merging

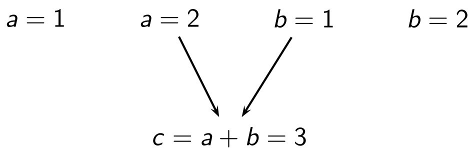


- Correct combination.

Why is Constant Propagation Non-Distributive?



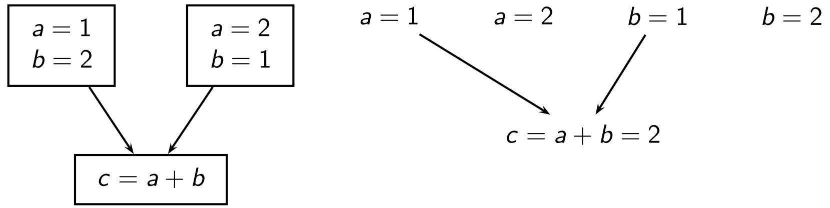
Possible combinations due to merging



- Correct combination.

Why is Constant Propagation Non-Distributive?

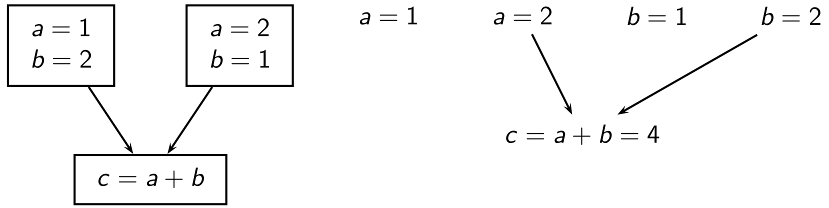
Possible combinations due to merging



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

Why is Constant Propagation Non-Distributive?

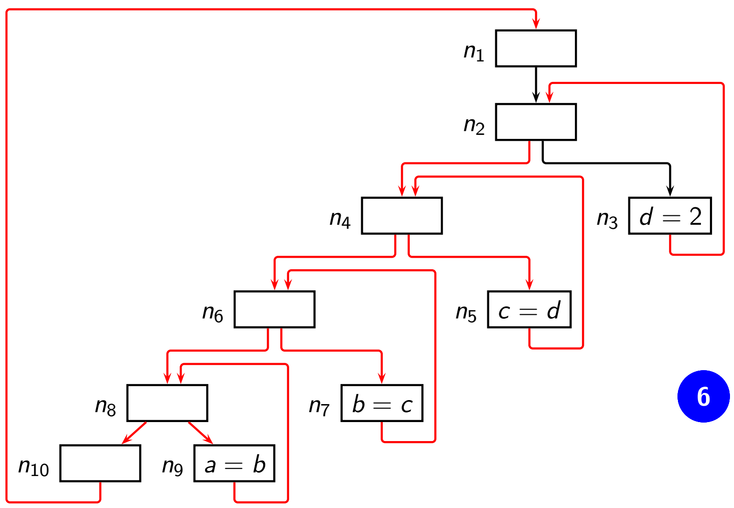
Possible combinations due to merging



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

Tutorial Problem on Constant Propagation

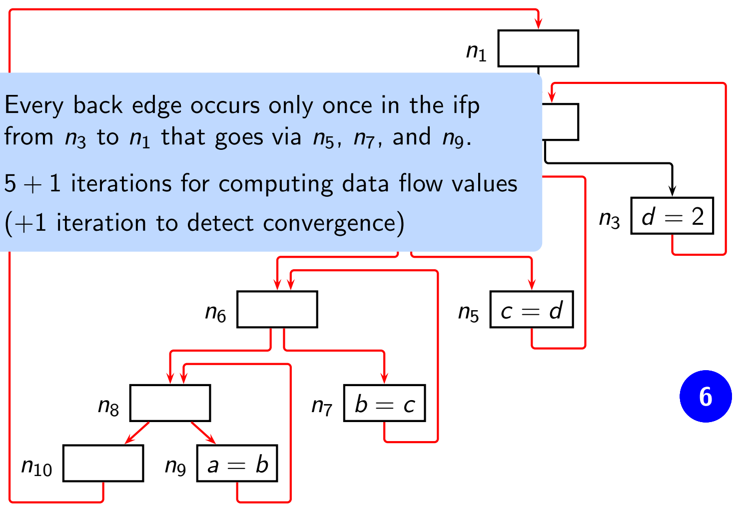
How many iterations do we need?



6

Tutorial Problem on Constant Propagation

How many iterations do we need?

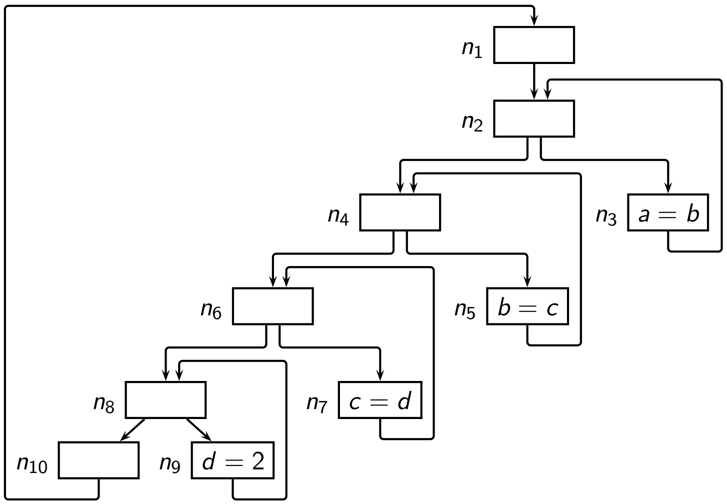


- Every back edge occurs only once in the ifp from n_3 to n_1 that goes via n_5, n_7, and n_9.
- $5 + 1$ iterations for computing data flow values ($+1$ iteration to detect convergence)

6

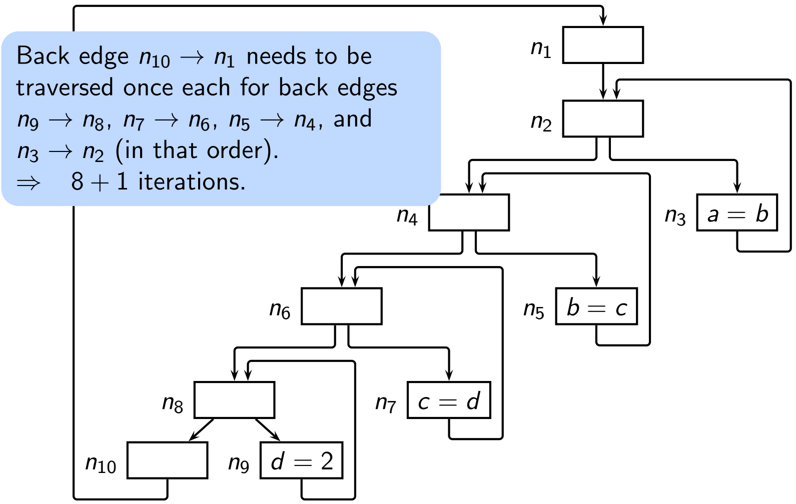
Tutorial Problem on Constant Propagation

And now how many iterations do we need?



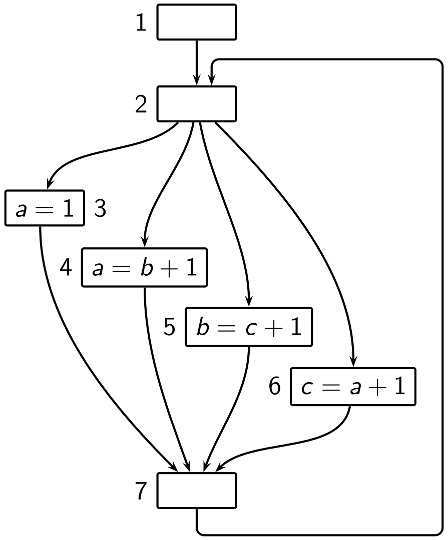
Tutorial Problem on Constant Propagation

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).
$\Rightarrow$ $8 + 1$ iterations.

Boundedness of Constant Propagation

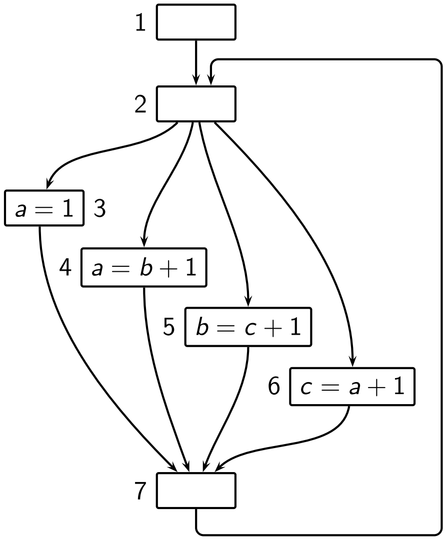


Summary flow function:
(data flow value at node 7)

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

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

Boundedness of Constant Propagation



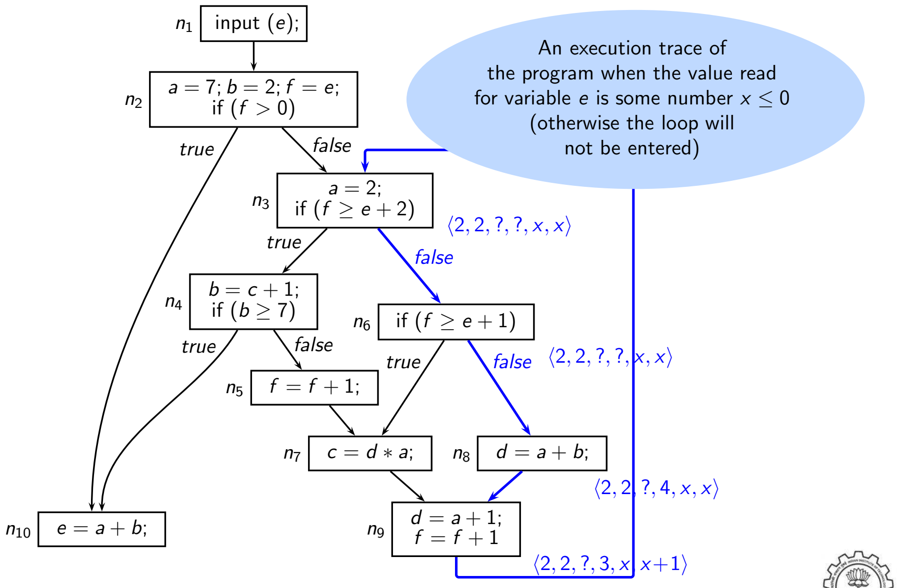
$$f^*(\top) = \prod_{i=0}^{6} f^i(\top)$$

Boundedness of Constant Propagation

The moral of the story:

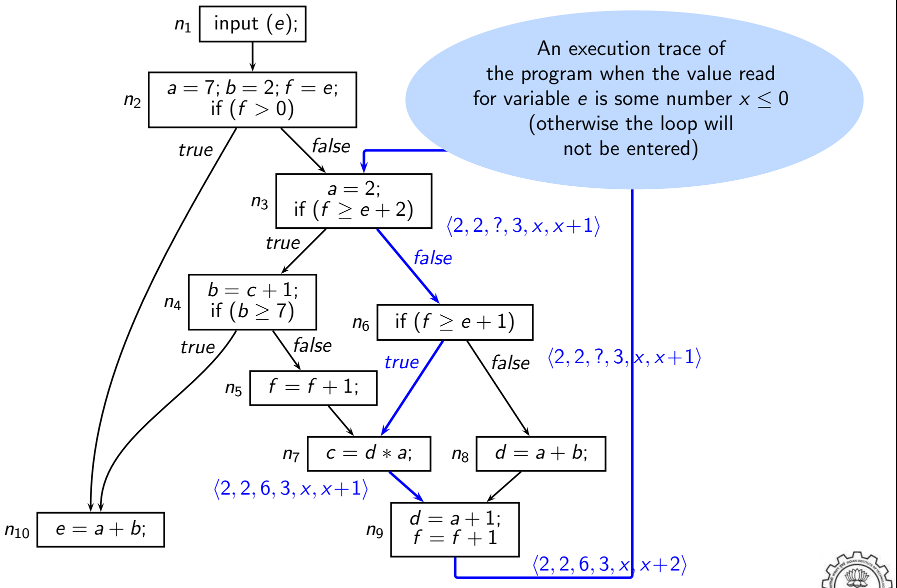
- The data flow value of every variable could change twice
- In the worst case, only one change may happen in every step of a function application
- Maximum number of steps: $2 \times |\mathbb{V}\mathrm{ar}|$
- Boundedness parameter k is $(2 \times |\mathbb{V}\mathrm{ar}|) + 1$

Conditional Constant Propagation



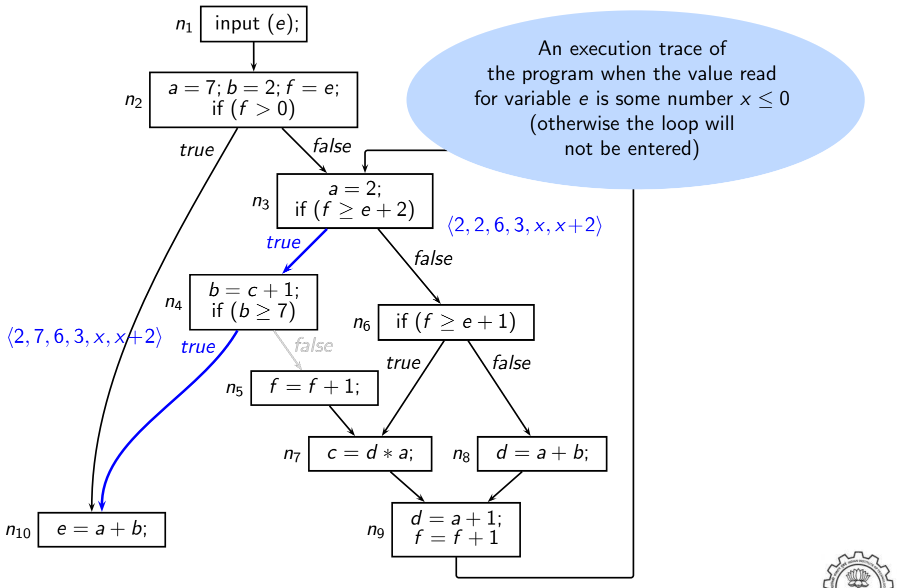
An execution trace of the program when the value read for variable e is some number $x \le 0$ (otherwise the loop will not be entered)

Conditional Constant Propagation



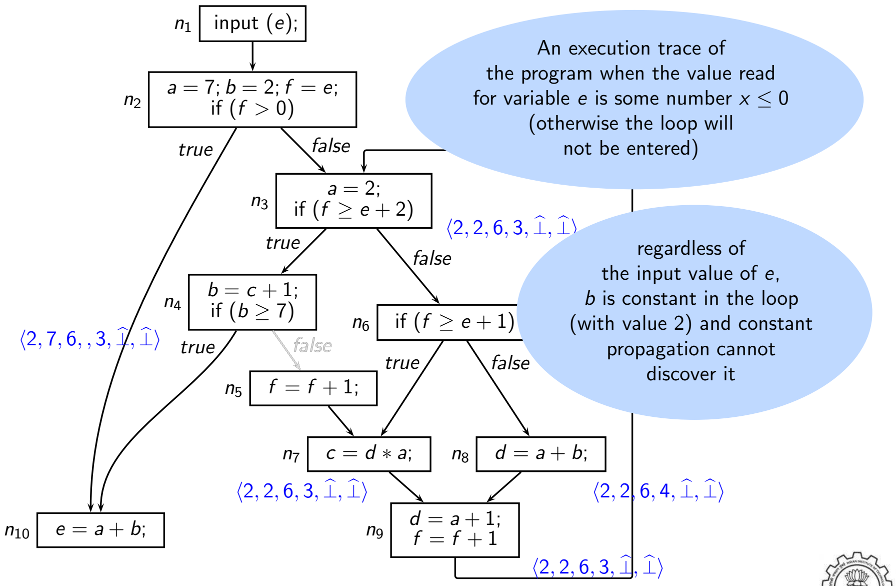
An execution trace of the program when the value read for variable e is some number $x \le 0$ (otherwise the loop will not be entered)

Conditional Constant Propagation



An execution trace of the program when the value read for variable e is some number $x \le 0$ (otherwise the loop will not be entered)

Conditional Constant Propagation

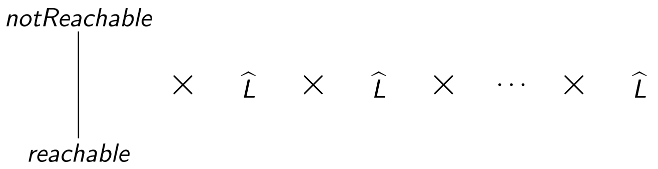


An execution trace of the program when the value read for variable *e* is some number $x \leq 0$ (otherwise the loop will not be entered)

regardless of the input value of *e*, *b* is constant in the loop (with value 2) and constant propagation cannot discover it

---

Lattice for Conditional Constant Propagation



$$\times \quad \widehat{L} \quad \times \quad \widehat{L} \quad \times \quad \cdots \quad \times \quad \widehat{L}$$

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

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

Data Flow Equations for Conditional Constant Propagation

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

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

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

- *label*$(m \to n)$ is T or F if edge $m \to n$ is a conditional branch
  Otherwise *label*$(m \to n)$ 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}	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$	
$In_{n_{10}}$	$R, \langle 7, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, \widehat{\bot}, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
$Out_{n_{10}}$	$R, \langle 7, 2, \widehat{\top}, \widehat{\top}, 9, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, \widehat{\bot}, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$

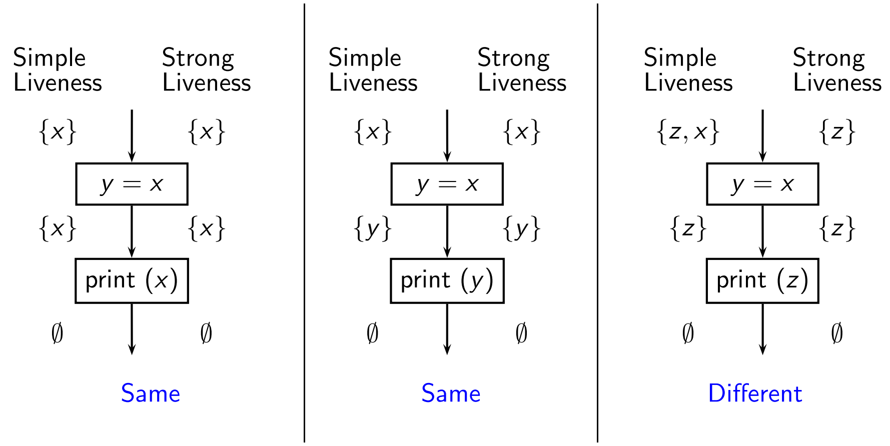
Part 4

Strongly Live Variables Analysis

Strongly Live Variables Analysis

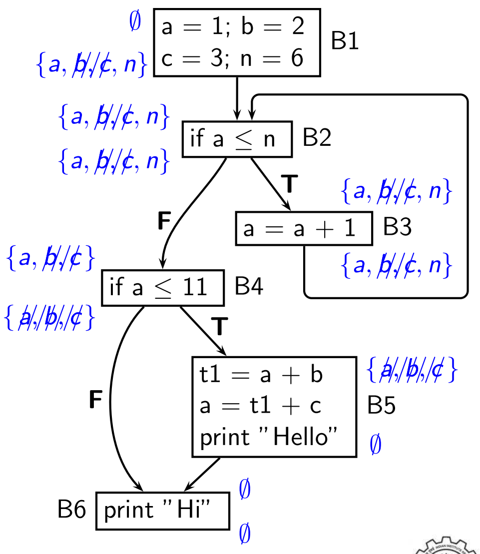
- A variable is strongly live if
  - it is used in a statement other than assignment statement, or (same as simple liveness)
  - it is used in an assignment statement defining a variable that is strongly live (different from simple liveness)
- Killing: An assignment statement, an input statement, or 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



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,

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

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?
  - $\widehat{L} = \{0, 1\}, 1 \sqsubseteq 0$
- 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

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

- 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)

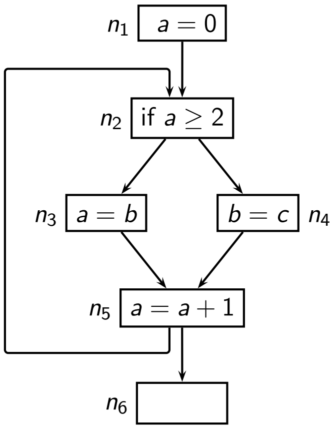
- For *input*(y) statement: $$\begin{aligned} f_n(X_1 \cup X_2) &= (X_1 \cup X_2) - \{y\} \\ &= (X_1 - \{y\}) \cup (X_2 - \{y\}) \\ &= f_n(X_1) \cup f_n(X_2) \end{aligned}$$
- For *use*(y) statement: $$\begin{aligned} f_n(X_1 \cup X_2) &= (X_1 \cup X_2) \cup \{y\} \\ &= (X_1 \cup \{y\}) \cup (X_2 \cup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

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

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 $\mathbb{V}\text{ar} \to \{0,1\}$ with the lattice $\{0,1\}$,

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

Part 5

Pointer Analyses

An Outline of Pointer Analysis Coverage

- The larger perspective
- Comparing Points-to and Alias information
- Flow Insensitive Points-to Analysis
- Flow Sensitive Points-to Analysis
- Pointer Analyses: An Engineer's Landscape
- Liveness Based Points-to Analysis
- Generalizations to Heap, Arrays, Pointer Arithmetic, and Unions

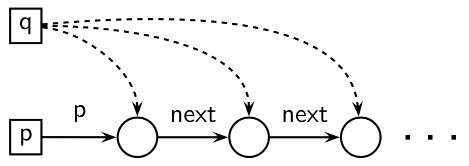
---

Code Optimization In Presence of Pointers

Program

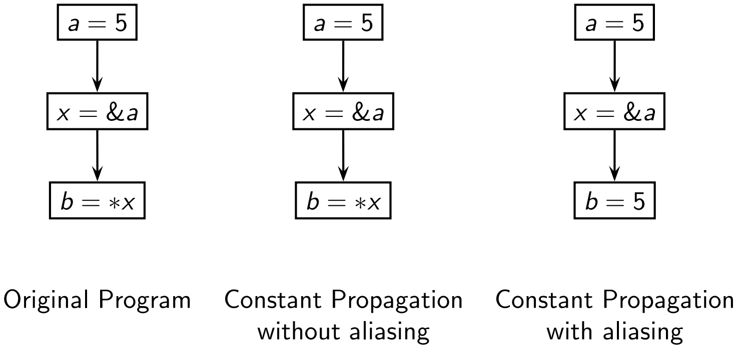
```
1.  q = p;
2.  while (...) {do {
3.      q = q→next;
4.  }while (...)
5.  p→data = r1;
6.  print (q→data);
7.  p→data = r2;
```

Memory graph at statement 5



- Is p→data live at the exit of line 5? Can we delete line 5?
- We cannot delete line 5 if p and q can be possibly aliased (while loop or do-while loop with a circular list)
- We can delete line 5 if p and q are definitely not aliased (do-while loop without a circular list)

Code Optimization In Presence of Pointers

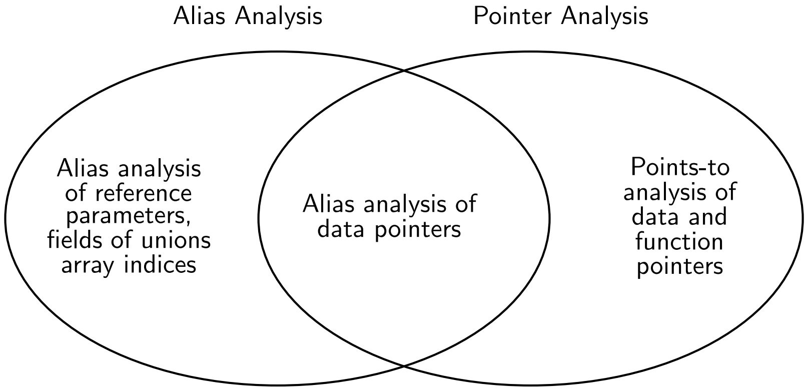


Original Program

Constant Propagation without aliasing

Constant Propagation with aliasing

The World of Pointer Analysis



Pointer Analysis Musings

- Pointer analysis collects information about indirect accesses in programs
    - Enables precise data analysis
    - Enable precise interprocedural control flow analysis
- Needs to scale to large programs
- Pointer Analysis Musings
    - Which Pointer Analysis should I Use?

      Michael Hind and Anthony Pioli. ISTAA 2000
    - Pointer Analysis: Haven't we solved this problem yet ?

      Michael Hind PASTE 2001
    - 2017 . . ☹

The Mathematics of Pointer Analysis

In the most general situation

- Alias analysis is undecidable.

  Landi-Ryder [POPL 1991], Landi [LOPLAS 1992], Ramalingam [TOPLAS 1994]
- Flow insensitive alias analysis is NP-hard

  Horwitz [TOPLAS 1997]
- Points-to analysis is undecidable

  Chakravarty [POPL 2003]

Adjust your expectations suitably to avoid disappointments!

The Engineering of Pointer Analysis

So what should we expect? To quote Hind [PASTE 2001]

- "Fortunately many approximations exist"
- "Unfortunately too many approximations exist!"

Engineering of pointer analysis is much more dominant than its science

---

Pointer Analysis: Engineering or Science?

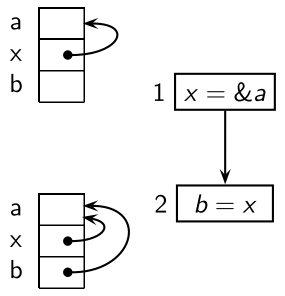
- Engineering view.
  - Build quick approximations
  - The tyranny of (exclusive) OR!
    Precision OR Efficiency?
- Science view.
  - Build clean abstractions
  - Can we harness the Genius of AND?
    Precision AND Efficiency?
- A distinction between approximation and abstraction is subjective
  Our working definition
  - Abstractions focus on precision and conciseness of modelling
  - Approximations focus on efficiency and scalability

An Outline of Pointer Analysis Coverage

- The larger perspective
- Comparing Points-to and Alias information Next Topic
- Flow Insensitive Points-to Analysis
- Flow Sensitive Points-to Analysis
- Pointer Analyses: An Engineer's Landscape
- Liveness Based Points-to Analysis
- Generalizations to Heap, Arrays, Pointer Arithmetic, and Unions

---

Alias Information Vs. Points-to Information



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

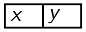
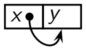
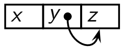
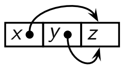
Neither
Symmetric
Nor Reflexive

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

Symmetric
and
Reflexive

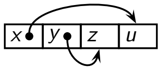
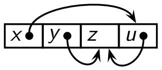
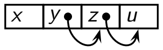
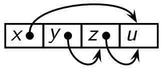
- What about transitivity?
  - Points-to: No.
  - Alias: Depends.

Comparing Points-to and Alias Relations (1)

Statement	Memory		Points-to		Aliases	
$x = \&y$	Before (assume)		Existing		Existing	
	After		New	$x \rightarrowtail y$	New Direct	$x \doteq \&y$
$x = y$	Before (assume)		Existing	$y \rightarrowtail z$	Existing	$y \doteq \&z$
	After		New	$x \rightarrowtail z$	New Direct	$x \doteq y$
					New Indirect	$x \doteq \&z$

- Indirect aliases. Substitute a name by its aliases for transitivity
- Derived aliases. Apply indirection operator to aliases (ignored here)
  $x \doteq y \;\Rightarrow\; *x \doteq *y$

Comparing Points-to and Alias Relations (2)

Statement	Memory		Points-to		Aliases	
$*x = y$	Before (assume)		Existing	$x \rightarrowtail u$ $y \rightarrowtail z$	Existing	$x \doteq \&u$ $y \doteq \&z$
					New Direct	$*x \doteq y$
	After		New	$u \rightarrowtail z$	New Indirect	$u \doteq \&z$ $y \doteq u$ $*x \doteq \&z$
$x = *y$	Before (assume)		Existing	$y \rightarrowtail z$ $z \rightarrowtail u$	Existing	$y \doteq \&z$ $z \doteq \&u$ $*y \doteq \&u$
					New Direct	$x \doteq *y$
	After		New	$x \rightarrowtail u$	New Indirect	$x \doteq \&u$ $x \doteq z$
The resulting memories look similar but are different. In the first case we have $u \rightarrowtail z$ whereas in the second case the arrow direction is opposite (i.e. $z \rightarrowtail u$).						

Comparing Points-to and Alias Relations (3)

- Points-to information records edges in the memory graph
  - aliases of the kind $x \doteq \&y$
    x holds the address of y
  - other aliases can be discovered by composing edges
  - since addresses are explicated, it can represent only those memory locations that can be named at compile time

  More compact but less general
- Alias information records paths in the memory graph
  - paths incident on the same node
    x and y hold the same address (and the address is left implicit)
  - since addresses are implicit, it can represent unnamed memory locations too
  - if we have $x \doteq y$ then $*x \doteq *y$ is redundant and is not recorded

  More general and more complex

An Outline of Pointer Analysis Coverage

- The larger perspective
- Comparing Points-to and Alias information
- Flow Insensitive Points-to Analysis Next Topic
- Flow Sensitive Points-to Analysis
- Pointer Analyses: An Engineer's Landscape
- Liveness Based Points-to Analysis
- Generalizations to Heap, Arrays, Pointer Arithmetic, and Unions

Flow Sensitive Vs. Flow Insensitive Pointer Analysis

- Flow insensitive pointer analysis
  - Inclusion based: Andersen's approach
  - Equality based: Steensgaard's approach
- Flow sensitive pointer analysis
  - May points-to analysis
  - Must points-to analysis

Flow Insensitivity in Data Flow Analysis

- Assumption: Statements can be executed in any order.
- Instead of computing point-specific data flow information, summary data flow information is computed.

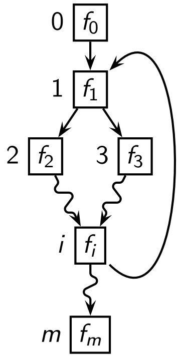
  The summary information is required to be a safe approximation of point-specific information for each point.
- $\text{Kill}_n(X)$ component is ignored.

  If statement n kills data flow information, there is an alternate path that excludes n.

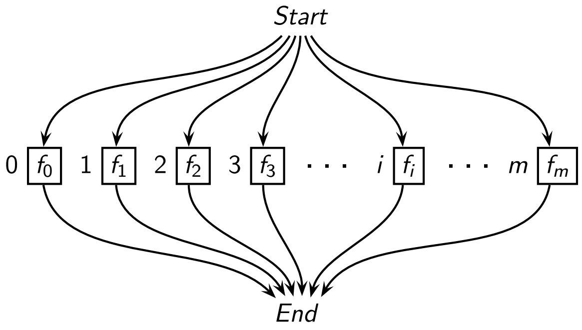
The control flow graph is a complete graph
(except for the Start and End nodes)

Flow Insensitivity in Data Flow Analysis

Assuming that there are no dependent parts in Gen_n and Kill_n is ignored



Control flow graph



Flow insensitive analysis

Function composition is replaced by function confluence

Examples of Flow Insensitive Analyses

- Type checking/inferencing

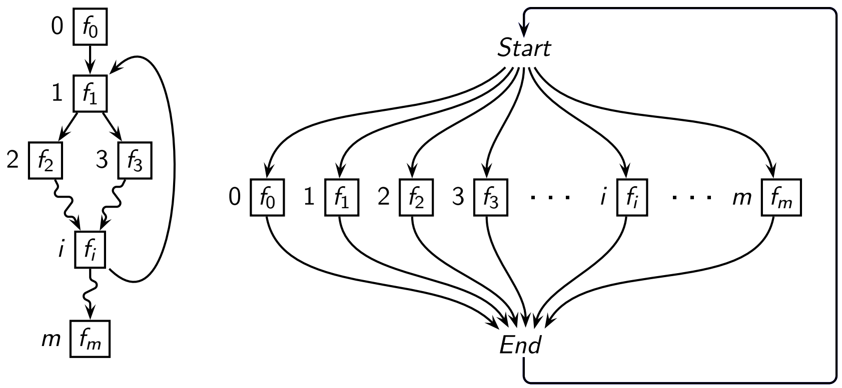
  (What about interpreted languages?)
- Address taken analysis

  Which variables have their addresses taken?
- Side effects analysis

  Does a procedure modify a global variable? Reference Parameter?

Flow Insensitivity in Data Flow Analysis

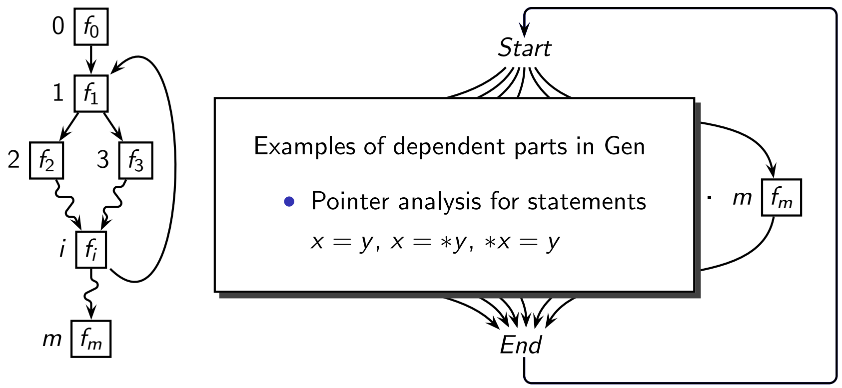
Assuming $\text{Gen}_n(X)$ has dependent parts and $\text{Kill}_n(X)$ is ignored



Allows arbitrary compositions of flow functions in any order
⇒ Flow insensitivity

Flow Insensitivity in Data Flow Analysis

Assuming $\text{Gen}_n(X)$ has dependent parts and $\text{Kill}_n(X)$ is ignored



Examples of dependent parts in Gen

- Pointer analysis for statements $x = y$, $x = *y$, $*x = y$

In practice, dependent constraints are collected in a global repository in one pass and then are solved independently

---

Notation for Andersen's and Steensgaard's Points-to Analysis

- P_x denotes the set of pointees of pointer variable x
- $Unify(x, y)$ unifies locations x and y
    - x and y are treated as equivalent locations
    - the pointees of the unified locations are also unified transitively
- $UnifyPTS(x, y)$ unifies the pointees of x and y
    - x and y themselves are not unified

Andersen's and Steensgaard's Points-to Analysis

Statement	Andersen's Points-to Sets	Steensgaard's Points-to Sets
$x = \&y$	$P_x \supseteq \{y\}$	$P_x \supseteq \{y\}$ $Unify(y, z)$ for some $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$UnifyPTS(x, y)$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ UnifyPTS(x, z)$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ UnifyPTS(y, z)$

Andersen's view

- x points to y
- Include y in the points-to set of x

Steensgaard's view

- Equivalence between: All pointees of x
- Unify y and pointees of x

Andersen's and Steensgaard's Points-to Analysis

Statement	Andersen's Points-to Sets	Steensgaard's Points-to Sets
$x = \&y$	$P_x \supseteq \{y\}$	$P_x \supseteq \{y\}$ $\mathit{Unify}(y, z)$ for some $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$\mathit{UnifyPTS}(x, y)$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ \mathit{UnifyPTS}(x, z)$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ \mathit{UnifyPTS}(y, z)$

Andersen's view

- x points to pointees of y
- Include the pointees of y in the points-to set of x

Steensgaard's view

- Equivalence between: Pointees of x and pointees of y
- Unify points-to sets of x and y

Andersen's and Steensgaard's Points-to Analysis

Statement	Andersen's Points-to Sets	Steensgaard's Points-to Sets
$x = \&y$	$P_x \supseteq \{y\}$	$P_x \supseteq \{y\}$ $\mathit{Unify}(y, z)$ for some $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$\mathit{UnifyPTS}(x, y)$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ \mathit{UnifyPTS}(x, z)$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ \mathit{UnifyPTS}(y, z)$

Andersen's view

- x points to pointees of pointees of y
- Include the pointees of pointees of y in the points-to set of x

Steensgaard's view

- Equivalence between: Pointees of x and pointees of pointees of y
- Unify points-to sets of x and pointees of y

Andersen's and Steensgaard's Points-to Analysis

Statement	Andersen's Points-to Sets	Steensgaard's Points-to Sets
$x = \&y$	$P_x \supseteq \{y\}$	$P_x \supseteq \{y\}$ $\mathit{Unify}(y, z)$ for some $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$\mathit{UnifyPTS}(x, y)$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ \mathit{UnifyPTS}(x, z)$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ \mathit{UnifyPTS}(y, z)$

Andersen's view

- Pointees of x points to pointees of y
- Include the pointees of y in the points-to set of the pointees of x

Steensgaard's view

- Equivalence between: Pointees of pointees of x and pointees of y
- Unify points-to sets of pointees of x and y

Andersen's and Steensgaard's Points-to Analysis

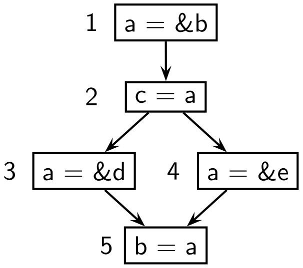
Statement	Andersen's Points-to Sets	Steensgaard's Points-to Sets
$x = \&y$	$P_x \supseteq \{y\}$	$P_x \supseteq \{y\}$ $\mathit{Unify}(y, z)$ for some $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$\mathit{UnifyPTS}(x, y)$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ \mathit{UnifyPTS}(x, z)$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ \mathit{UnifyPTS}(y, z)$

Inclusion

Equality

Inclusion Based (aka Andersen's) Points-to Analysis: Example 1

Program



1 a = &b

2 c = a

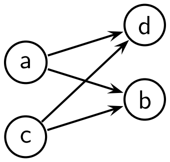
3 a = &d

4 a = &e

5 b = a

Node	Constraint
1	$P_a \supseteq \{b\}$
2	$P_c \supseteq P_a$
3	$P_a \supseteq \{d\}$
4	$P_a \supseteq \{e\}$
5	$P_b \supseteq P_a$

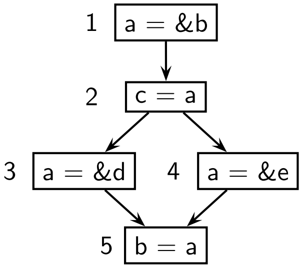
Points-to Graph



- Since P_a has changed, P_c needs to be processed again

Inclusion Based (aka Andersen's) Points-to Analysis: Example 1

Program



1 a = &b

2 c = a

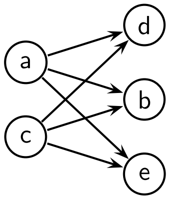
3 a = &d

4 a = &e

5 b = a

Node	Constraint
1	$P_a \supseteq \{b\}$
2	$P_c \supseteq P_a$
3	$P_a \supseteq \{d\}$
4	$P_a \supseteq \{e\}$
5	$P_b \supseteq P_a$

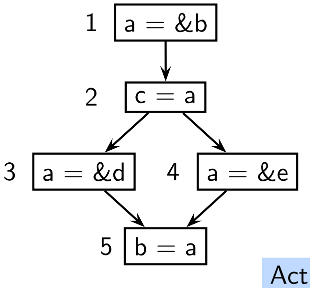
Points-to Graph



- Observe that P_c is processed for the third time
- Order of processing the sets influences efficiency significantly
- A plethora of heuristics have been proposed

Inclusion Based (aka Andersen's) Points-to Analysis: Example 1

Program



1 a = &b

2 c = a

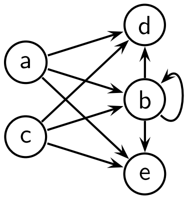
3 a = &d

4 a = &e

5 b = a

Node	Constraint
1	$P_a \supseteq \{b\}$
2	$P_c \supseteq P_a$
3	$P_a \supseteq \{d\}$
4	$P_a \supseteq \{e\}$
5	$P_b \supseteq P_a$

Points-to Graph

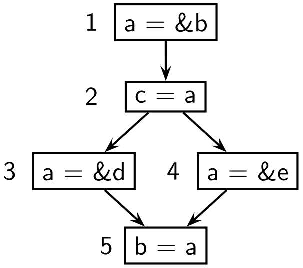


Actually:

- c does not point to any location in block 1
- a does not point b in block 5
(the method ignores the kill due to 3 and 4)
- b does not point to itself at any time

Equality Based (aka Steensgaard's) Points-to Analysis: Example 1

Program



1 a = &b

2 c = a

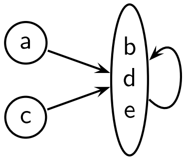
3 a = &d

4 a = &e

5 b = a

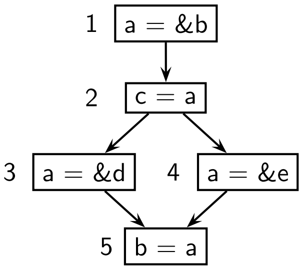
Node	Constraint
1	$P_a \supseteq \{b\}$ $Unify(x, d), x \in P_a$
2	$UnifyPTS(c, a)$
3	$P_a \supseteq \{d\}$ $Unify(x, d), x \in P_a$
4	$P_a \supseteq \{e\}$ $Unify(x, e), x \in P_a$
5	$UnifyPTS(b, a)$

Points-to Graph



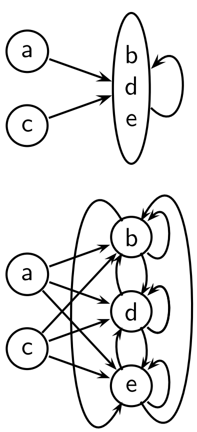
Equality Based (aka Steensgaard's) Points-to Analysis: Example 1

Program



Node	Constraint
1	$P_a \supseteq \{b\}$ $Unify(x, d), x \in P_a$
2	$UnifyPTS(c, a)$
3	$P_a \supseteq \{d\}$ $Unify(x, d), x \in P_a$
4	$P_a \supseteq \{e\}$ $Unify(x, e), x \in P_a$
5	$UnifyPTS(b, a)$

Points-to Graph

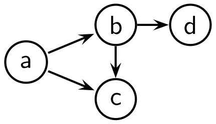
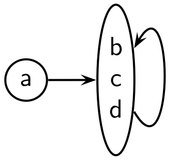


- The full blown up points-to graph has far more edges than in the graph created by Andersen's method
- Far more efficient but far less precise

Comparing Equality and Inclusion Based Analyses (2)

- Andersen's algorithm is cubic in number of pointers
- Steensgaard's algorithm is nearly linear in number of pointers
  - How can it be more efficient by an orders of magnitude?

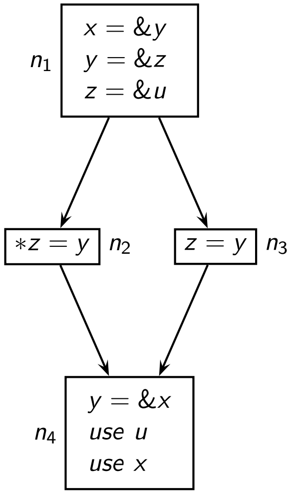
Efficiency of Equality Based Approach

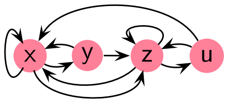
Program	Andersen's approach	Steensgaard's approach
a = &b a = &c b = &d b = &c		

- Andersen's inclusion based wisdom:
  - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
  - Merge multiple successors and maintain a single successor of any node
  - Since a larger number of pointers treated are alike and fewer distinctions are maintained, we get much smaller points-to graphs
  - Efficient *Union-Find* algorithms to merge intersecting subsets

---

Inclusion Based (aka Andersen's) Points-to Analysis: Example 2



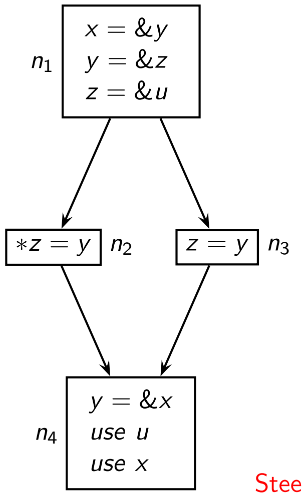


Points-to Graph

Constraints on Points-to Sets

$$
\begin{aligned}
P_x &\supseteq \{y\} \\
P_y &\supseteq \{z\} \\
P_z &\supseteq \{u\} \\
\forall w \in P_z,\ P_w &\supseteq P_y \\
P_z &\supseteq P_y \\
P_y &\supseteq \{x\}
\end{aligned}
$$

Equality Based (aka Steensgaard's) Points-to Analysis: Example 2



n_1

$x = \&y$

$y = \&z$

$z = \&u$

$*z = y$ n_2

$z = y$ n_3

n_4

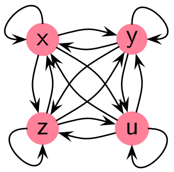
$y = \&x$

use u

use x

- Treat all pointees of a pointer as "equivalent" locations
- Transitive closure

  Pointees of all equivalent locations become equivalent



Steensgaard's Points-to Graph

Effective additional constraints

$Unify(x, y)$
/* pointees of x */

$Unify(x, z)$
/* pointees of y */

$Unify(x, u)$
/* pointees of z */

⇒ x, y, z, u are equivalent

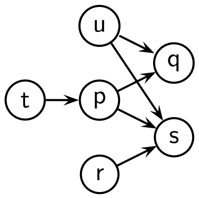
⇒ Complete graph

Tutorial Problem for Flow Insensitive Pointer Analysis (1)

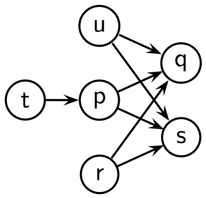
Program

p = &q
r = &s
t = &p
u = p
∗t = r

Inclusion based



Equality based



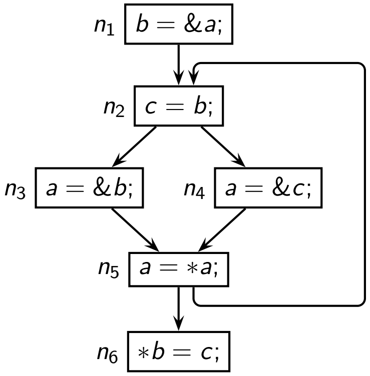
Tutorial Problems for Flow Insensitive Pointer Analysis (2)

Compute flow insensitive points-to information using inclusion based method as well as equality based method

```
if (...)
        p = &x;
else
        p = &y;
x = &a;
y = &b;
∗p = &c;
∗y = &a;
```

Tutorial Problem for Flow Insensitive Pointer Analysis (3)

Compute flow insensitive points-to information using inclusion based method as well as equality based method



n_1 $b = \&a;$

n_2 $c = b;$

n_3 $a = \&b;$

n_4 $a = \&c;$

n_5 $a = *a;$

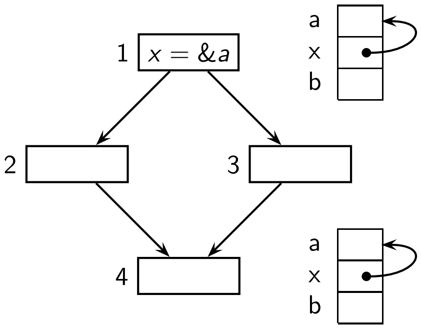
n_6 $*b = c;$

An Outline of Pointer Analysis Coverage

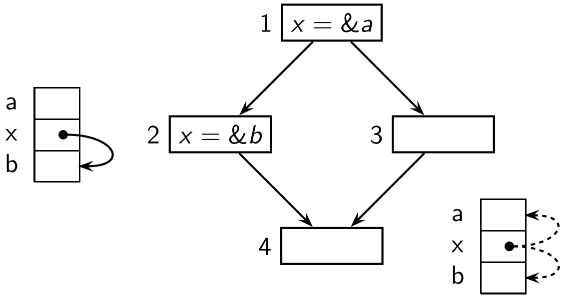
- The larger perspective
- Comparing Points-to and Alias information
- Flow Insensitive Points-to Analysis
- Flow Sensitive Points-to Analysis Next Topic
- Pointer Analyses: An Engineer's Landscape
- Liveness Based Points-to Analysis
- Generalizations to Heap, Arrays, Pointer Arithmetic, and Unions

---

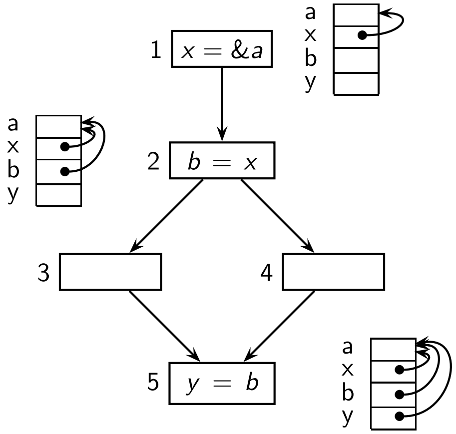
Must Points-to Information



May Points-to Information

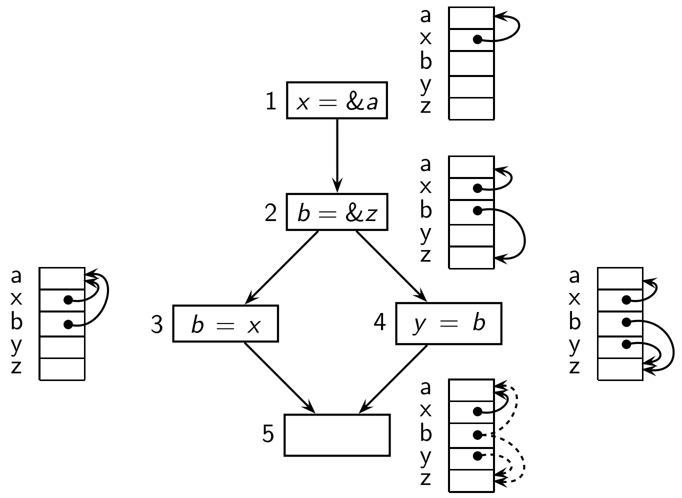


Must Alias Information



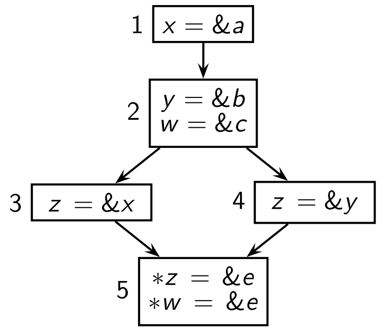
$x \overset{\circ}{=} b$ and $b \overset{\circ}{=} y \Rightarrow x \overset{\circ}{=} y$

May Alias Information



$x \doteq b$ and $b \doteq y \nRightarrow x \doteq y$

Strong and Weak Updates



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

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

How is this concept related to May/Must nature of information?

What About Heap Data?

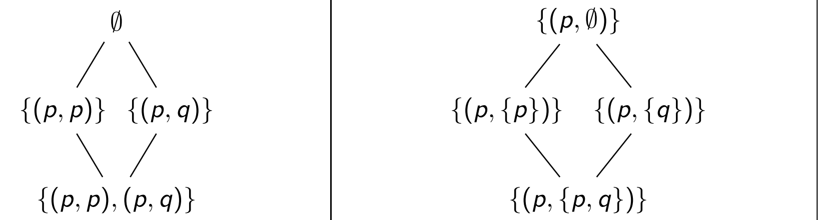
- Compile time entities, abstract entities, or summarized entities
- Three options:
    - Represent all heap locations by a single abstract heap location
    - Represent all heap locations of a particular type by a single abstract heap location
    - Represent all heap locations allocated at a given memory allocation site by a single abstract heap location
- Summarization: Usually based on the length of pointer expression
- *Initially, we will restrict ourselves to stack and static data*
  We will later introduce heap using the allocation site based abstraction

---

Lattice for May Points-to Analysis

Let $\mathbf{P} \subseteq \mathbb{V}\text{ar}$ be the set of pointers. Assume $\mathbb{V}\text{ar} = \{p, q\}$ and $\mathbf{P} = \{p\}$

Product View | Mapping view



Data flow values $\subseteq \boxed{\mathbf{P} \times \mathbb{V}\text{ar}}$

Lattice $= (2^{\mathbf{P} \times \mathbb{V}\text{ar}}, \supseteq)$

Points-to graph as a list of directed edges

Data flow values $\in \boxed{\mathbf{P} \to 2^{\mathbb{V}\text{ar}}}$

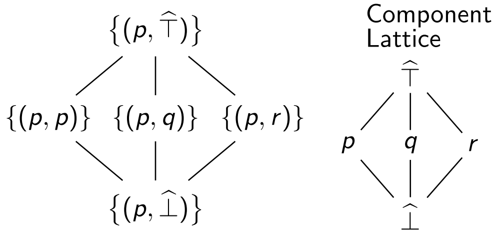
Lattice $= (\mathbf{P} \to 2^{\mathbb{V}\text{ar}}, \sqsubseteq_{map})$

Points-to graph as a list of adjacency lists

Lattice for Must Points-to Analysis

Let $\mathbf{P} \subseteq \mathbb{V}\mathrm{ar}$ be the set of pointers. Assume $\mathbb{V}\mathrm{ar} = \{p, q, r\}$ and $\mathbf{P} = \{p\}$

Mapping View



$\{(p, \widehat{\top})\}$

$\{(p, p)\}$ $\{(p, q)\}$ $\{(p, r)\}$

$\{(p, \widehat{\bot})\}$

Component Lattice

$\widehat{\top}$

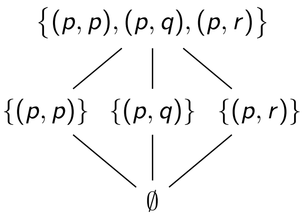
p q r

$\widehat{\bot}$

Data flow values = $\mathbf{P} \to \mathbb{V}\mathrm{ar} \cup \{\widehat{\top}, \widehat{\bot}\}$

Lattice = $\left(2^{\mathbf{P} \to \mathbb{V}\mathrm{ar} \cup \{\widehat{\top}, \widehat{\bot}\}}, \sqsubseteq_{map}\right)$

Set View



$\{(p, p), (p, q), (p, r)\}$

$\{(p, p)\}$ $\{(p, q)\}$ $\{(p, r)\}$

$\emptyset$

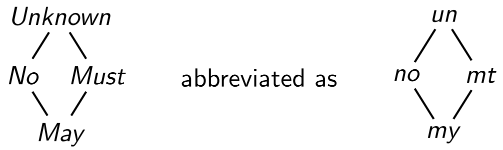
Restricted subset of $\mathbf{P} \times \mathbb{V}\mathrm{ar}$

$\cap$ can be used for $\sqcap$

A pointer can point to at most one location

Lattice for Combined May-Must Points-to Analysis (1)

- Consider the following abbreviation of the May-Must lattice $\widehat{L}$



Unknown

No *Must*

May

abbreviated as

un

no *mt*

my

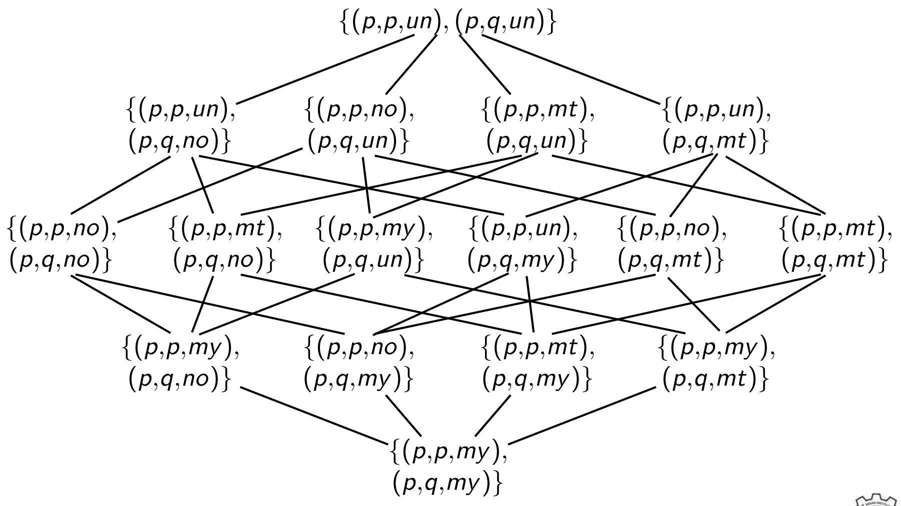
- For $\mathbb{V}\mathrm{ar} = \{p, q\}$, $\mathbf{P} = \{p\}$, the May-Must points-to lattice is the product

$$\mathbf{P} \times \mathbb{V}\mathrm{ar} \times \widehat{L}$$

  - Some elements are prohibited because of the semantics of *Must*
  - If we have (p,p,mt) in a data flow value $X \in \mathbf{P} \times \mathbb{V}\mathrm{ar} \times \widehat{L}$, then
    - we cannot have (p,q,un), (p,q,mt), or (p,q,my) in X
    - we can only have (p,q,no) in X

Lattice for Combined May-Must Points-to Analysis (2)

For $\mathbb{V}\mathrm{ar} = \{p, q\}$, $\mathbf{P} = \{p\}$, the May-Must points-to lattice is



$\{(p,p,un),(p,q,un)\}$

$\{(p,p,un),(p,q,no)\}$ $\{(p,p,no),(p,q,un)\}$ $\{(p,p,mt),(p,q,un)\}$ $\{(p,p,un),(p,q,mt)\}$

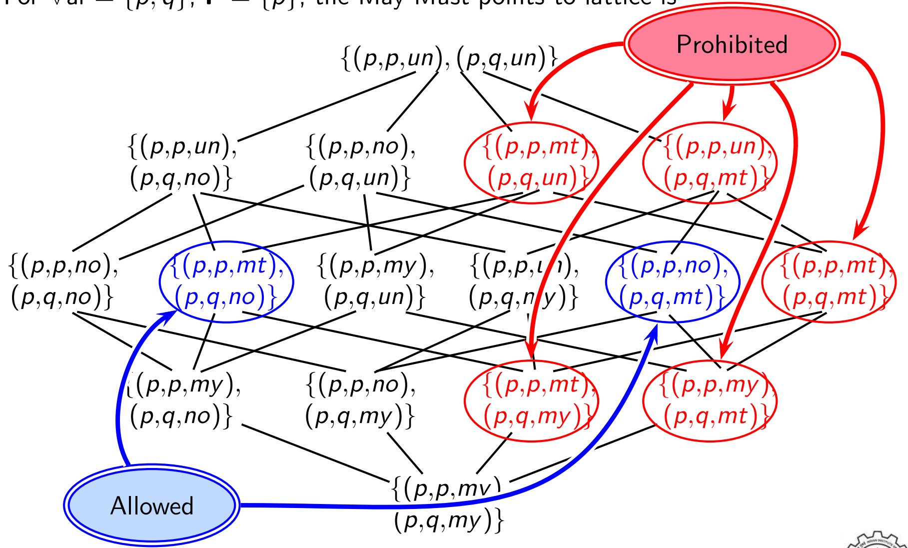
$\{(p,p,no),(p,q,no)\}$ $\{(p,p,mt),(p,q,no)\}$ $\{(p,p,my),(p,q,un)\}$ $\{(p,p,un),(p,q,my)\}$ $\{(p,p,no),(p,q,mt)\}$ $\{(p,p,mt),(p,q,mt)\}$

$\{(p,p,my),(p,q,no)\}$ $\{(p,p,no),(p,q,my)\}$ $\{(p,p,mt),(p,q,my)\}$ $\{(p,p,my),(p,q,mt)\}$

$\{(p,p,my),(p,q,my)\}$

Lattice for Combined May-Must Points-to Analysis (2)

For $\mathbb{V}\mathrm{ar} = \{p, q\}$, $\mathbf{P} = \{p\}$, the May-Must points-to lattice is



Prohibited

$\{(p,p,un),(p,q,un)\}$

$\{(p,p,un),(p,q,no)\}$ $\{(p,p,no),(p,q,un)\}$ $\{(p,p,mt),(p,q,un)\}$ $\{(p,p,un),(p,q,mt)\}$

$\{(p,p,no),(p,q,no)\}$ $\{(p,p,mt),(p,q,no)\}$ $\{(p,p,my),(p,q,un)\}$ $\{(p,p,un),(p,q,my)\}$ $\{(p,p,no),(p,q,mt)\}$ $\{(p,p,mt),(p,q,mt)\}$

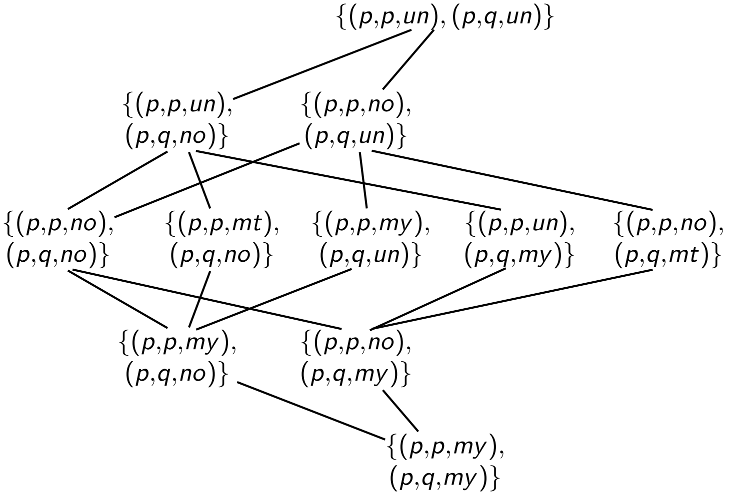
$\{(p,p,my),(p,q,no)\}$ $\{(p,p,no),(p,q,my)\}$ $\{(p,p,mt),(p,q,my)\}$ $\{(p,p,my),(p,q,mt)\}$

$\{(p,p,my),(p,q,my)\}$

Allowed

Lattice for Combined May-Must Points-to Analysis (2)

For $\mathbb{V}\text{ar} = \{p, q\}$, $\mathbf{P} = \{p\}$, the May-Must points-to lattice is



$\{(p,p,un),(p,q,un)\}$

$\{(p,p,un),(p,q,no)\}$ $\{(p,p,no),(p,q,un)\}$

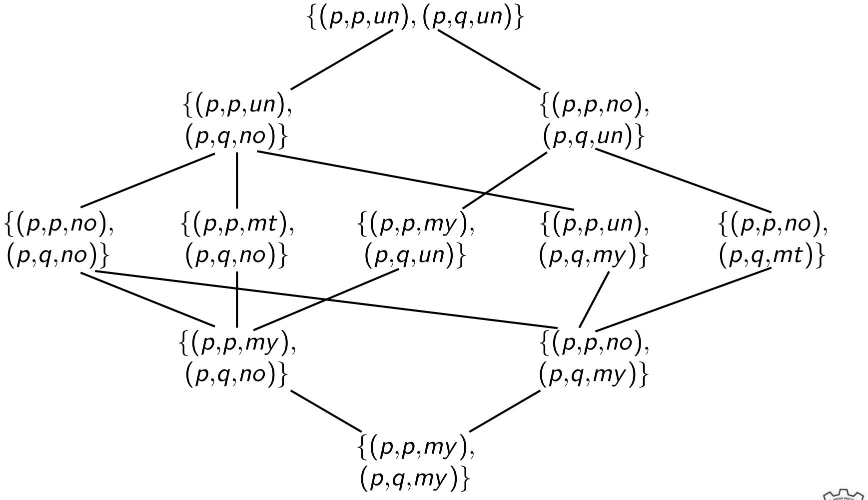
$\{(p,p,no),(p,q,no)\}$ $\{(p,p,mt),(p,q,no)\}$ $\{(p,p,my),(p,q,un)\}$ $\{(p,p,un),(p,q,my)\}$ $\{(p,p,no),(p,q,mt)\}$

$\{(p,p,my),(p,q,no)\}$ $\{(p,p,no),(p,q,my)\}$

$\{(p,p,my),(p,q,my)\}$

Lattice for Combined May-Must Points-to Analysis (2)

For $\mathbb{V}\text{ar} = \{p, q\}$, $\mathbf{P} = \{p\}$, the May-Must points-to lattice is



$\{(p,p,un),(p,q,un)\}$

$\{(p,p,un),(p,q,no)\}$ $\{(p,p,no),(p,q,un)\}$

$\{(p,p,no),(p,q,no)\}$ $\{(p,p,mt),(p,q,no)\}$ $\{(p,p,my),(p,q,un)\}$ $\{(p,p,un),(p,q,my)\}$ $\{(p,p,no),(p,q,mt)\}$

$\{(p,p,my),(p,q,no)\}$ $\{(p,p,no),(p,q,my)\}$

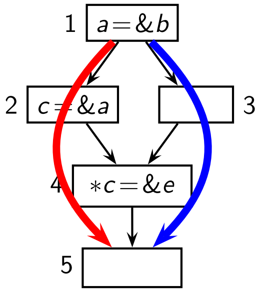
$\{(p,p,my),(p,q,my)\}$

May and Must Analysis for Killing Points-to Information (1)

May Points-to Analysis

- (a, b) should be in $MayIn_5$

  Holds along path 1-3-4
- Block 4 should not kill (a, b)
- Possible if pointee set of c is $\emptyset$ (Use $MustIn_4$)
- However, $MayIn_4$ contains (c, a)



Must Points-to Analysis

- (a, b) should not be in $MustIn_5$

  Does not hold along path 1-2-4
- Block 4 should kill (a, b)
- Possible if pointee set of c is $\{a\}$ (Use $MayIn_4$)
- However, $MustIn_4$ contains (a, b)

For killing points-to information through indirection,

- Must points-to analysis should identify pointees of c using $MayIn_4$
- May points-to analysis should identify pointees of c using $MustIn_4$

May and Must Analysis for Killing Points-to Information (2)

- May Points-to analysis should remove a May points-to pair
  - only if it must be removed along all paths

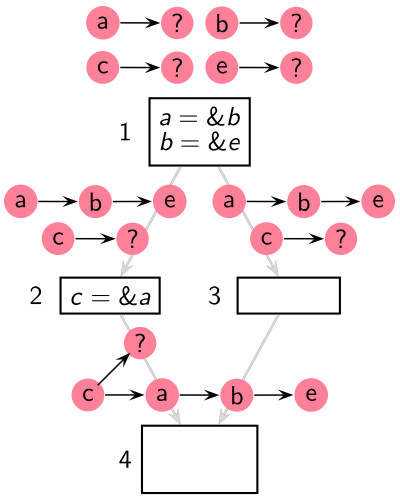
  Kill should remove only strong updates

  ⇒ should use Must Points-to information
- Must Points-to analysis should remove a Must points-to pair
  - if it can be removed along any path

  Kill should remove all weak updates

  ⇒ should use May Points-to information

Discovering Must Points-to Information from May Points-to Information



- *BI.* every pointer points to “?”
- Perform usual may points-to analysis
- Since c has multiple pointees, it is a MAY relation
- Since a has a single pointee, it is a MUST relation

Relevant Algebraic Operations on Relations (1)

- Let $\mathbf{P} \subseteq \mathbb{V}\mathrm{ar}$ be the set of pointer variables
- May-points-to information: $\mathcal{A} = \langle 2^{\mathbf{P} \times \mathbb{V}\mathrm{ar}}, \supseteq \rangle$
- Standard algebraic operations on points-to relations
  Given relation $R \subseteq \mathbf{P} \times \mathbb{V}\mathrm{ar}$ and $X \subseteq \mathbf{P}$,
  - Relation *application* $R\ X = \{v \mid u \in X \wedge (u, v) \in R\}$
    (Find out the pointees of the pointers contained in X)
  - Relation *restriction* $(R|_X)$ $R|_X = \{(u, v) \in R \mid u \in X\}$
    (Restrict the relation only to the pointers contained in X by removing points-to information of other pointers)

Relevant Algebraic Operations on Relations (2)

Let

$$\begin{aligned}
\mathbb{V}\mathrm{ar} &= \{a, b, c, d, e, f, g, ?\} \\
\mathbf{P} &= \{a, b, c, d, e\} \\
R &= \{(a, b), (a, c), (b, d), (c, e), (c, g), (d, a), (e, ?)\} \\
X &= \{a, c\}
\end{aligned}$$

Then,

$$\begin{aligned}
R\ X &= \{v \mid u \in X \wedge (u, v) \in R\} \\
&= \{b, c, e, g\} \\
R|_X &= \{(u, v) \in R \mid u \in X\} \\
&= \{(a, b), (a, c), (c, e), (c, g)\}
\end{aligned}$$

Points-to Analysis Data Flow Equations

$$Ain_n = \begin{cases} \mathbb{V}\mathrm{ar} \times \{?\} & n \text{ is } Start_p \\ \displaystyle\bigcup_{p \in pred(n)} Aout_p & \text{otherwise} \end{cases}$$

$$Aout_n = \Big(Ain_n - \Big(Kill_n \times \mathbb{V}\mathrm{ar}\Big)\Big) \cup \Big(Def_n \times Pointee_n\Big)$$

- $Ain/Aout$: sets of mAy points-to pairs
- $Kill_n$, Def_n, and $Pointee_n$ are defined in terms of Ain_n

Points-to Analysis Data Flow Equations

Pointees (i.e. locations whose addresses are stored)

$$Ain_n = \begin{cases} \mathbb{V}\text{ar} \times \{?\} & n \text{ is } Start_p \\ \bigcup\limits_{p\in pred(n)} Aout_p & \text{otherwise} \end{cases}$$

$$Aout_n = \left(Ain_n - \left(\boxed{Kill_n} \times \mathbb{V}\text{ar}\right)\right) \cup \left(\boxed{Def_n} \times \boxed{Pointee_n}\right)$$

- $Ain/Aout$: sets of mAy points-to pairs
- $Kill_n$, Def_n, and $Pointee_n$ are defined in terms of Ain_n

Pointers whose points-to relations should be removed

Pointers that are defined (i.e. pointers in which addresses are stored)

Extractor Functions for Points-to Analysis

Values defined in terms of Ain_n (denoted A)

	Def_n	$Kill_n$	$Pointee_n$
use x	$\emptyset$	$\emptyset$	$\emptyset$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$
$x = y$	$\{x\}$	$\{x\}$	$A\{y\}$
$x = *y$	$\{x\}$	$\{x\}$	$A(A\{y\} \cap \mathbf{P})$
$*x = y$	$A\{x\} \cap \mathbf{P}$	$Must(A)\{x\} \cap \mathbf{P}$	$A\{y\}$
other	$\emptyset$	$\emptyset$	$\emptyset$

Extractor Functions for Points-to Analysis

Values defined in terms of Ain_n (denoted A)

	Def_n	$Kill_n$	$Pointee_n$
use x	$\emptyset$	$\emptyset$	$\emptyset$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$
$x = y$	$\{x\}$	$\{x\}$	$\boxed{A\{y\}}$
$x = *y$	$\{x\}$	$\{x\}$	$A(A\{y\} \cap \mathbf{P})$
$*x = y$	$A\{x\} \cap \mathbf{P}$	$Must(A)\{x\} \cap \mathbf{P}$	$A\{y\}$
other	$\emptyset$	$\emptyset$	$\emptyset$

Pointees of y in Ain_n are the targets of defined pointers

Extractor Functions for Points-to Analysis

Values defined in terms of Ain_n (denoted A)

	Def_n	$Kill_n$	$Pointee_n$
use x	$\emptyset$	$\emptyset$	$\emptyset$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$
$x = y$	$\{x\}$	$\{x\}$	$A\{y\}$
$x = *y$	$\{x\}$	$\{x\}$	$\boxed{A(A\{y\} \cap \mathbf{P})}$
$*x = y$	$A\{x\} \cap \mathbf{P}$	$Must(A)\{x\} \cap \mathbf{P}$	$A\{y\}$
other	$\emptyset$	$\emptyset$	$\emptyset$

Pointees of those pointees of y in Ain_n which are pointers

Extractor Functions for Points-to Analysis

Values defined in terms of Ain_n (denoted A)

	Def_n	$Kill_n$	$Pointee_n$
$use\ x$	$\emptyset$	$\emptyset$	$\emptyset$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$
$x = y$	$\{x\}$	$\{x\}$	$A\{y\}$
$x = *y$	$\{x\}$	$\{x\}$	$A(A\{y\} \cap \mathbf{P})$
$*x = y$	$\boxed{A\{x\} \cap \mathbf{P}}$	$Must(A)\{x\} \cap \mathbf{P}$	$A\{y\}$
other	$\emptyset$	$\emptyset$	$\emptyset$

Pointees of x in Ain_n receive new addresses

Extractor Functions for Points-to Analysis

Values defined in terms of Ai

Strong update using must-points-to information computed from Ain_n

	Def_n	$Kill_n$	
$use\ x$	$\emptyset$	$\emptyset$	$\emptyset$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$
$x = y$	$\{x\}$	$\{x\}$	$A\{y\}$
$x = *y$	$\{x\}$	$\{x\}$	$A(A\{y\} \cap \mathbf{P})$
$*x = y$	$A\{x\} \cap \mathbf{P}$	$\boxed{Must(A)\{x\} \cap \mathbf{P}}$	$A\{y\}$
other	$\emptyset$	$\emptyset$	$\emptyset$

$$Must(R) = \boxed{\bigcup_{z \in \mathbf{P}} \{z\}} \times \begin{cases} \{w\} & R\{z\} = \{w\} \wedge w \neq ? \\ \emptyset & \text{otherwise} \end{cases}$$

Find out must-pointees of all pointers

Extractor Functions for Points-to Analysis

Values defined in terms of Ai

Strong update using must-points-to information computed from Ain_n

	Def_n	$Kill_n$	
$use\ x$	$\emptyset$	$\emptyset$	$\emptyset$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$
$x = y$	$\{x\}$	$\{x\}$	$A\{y\}$
$x = *y$	$\{x\}$	$\{x\}$	$A(A\{y\} \cap \mathbf{P})$
$*x = y$	$A\{x\} \cap \mathbf{P}$	$\boxed{Must(A)\{x\} \cap \mathbf{P}}$	$A\{y\}$
other	$\emptyset$	$\emptyset$	$\emptyset$

$$Must(R) = \bigcup_{z \in \mathbf{P}} \{z\} \times \begin{cases} \boxed{\{w\}} & \boxed{R\{z\} = \{w\} \wedge w \neq ?} \\ \emptyset & \text{otherwise} \end{cases}$$

z has a single pointee w in must-points-to relation

Extractor Functions for Points-to Analysis

Values defined in terms of Ai

Strong update using must-points-to information computed from Ain_n

	Def_n	$Kill_n$	
$use\ x$	$\emptyset$	$\emptyset$	$\emptyset$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$
$x = y$	$\{x\}$	$\{x\}$	$A\{y\}$
$x = *y$	$\{x\}$	$\{x\}$	$A(A\{y\} \cap \mathbf{P})$
$*x = y$	$A\{x\} \cap \mathbf{P}$	$\boxed{Must(A)\{x\} \cap \mathbf{P}}$	$A\{y\}$
other	$\emptyset$	$\emptyset$	$\emptyset$

$$Must(R) = \bigcup_{z \in \mathbf{P}} \{z\} \times \begin{cases} \{w\} & R\{z\} = \{w\} \wedge w \neq ? \\ \boxed{\emptyset} & \boxed{\text{otherwise}} \end{cases}$$

z has no pointee in must-points-to relation

Extractor Functions for Points-to Analysis

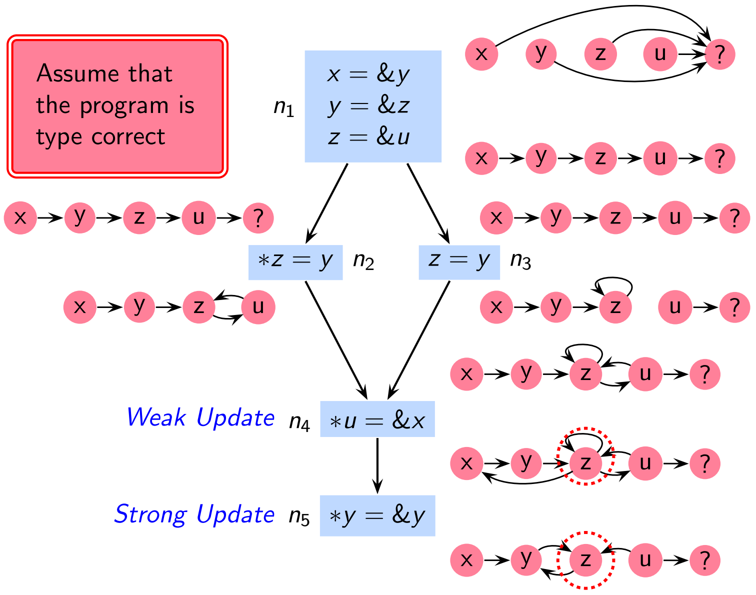
Values defined in terms of Ain_n (denoted A)

	Def_n	$Kill_n$	$Pointee_n$
use x	$\emptyset$	$\emptyset$	$\emptyset$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$
$x = y$	$\{x\}$	$\{x\}$	$A\{y\}$
$x = *y$	$\{x\}$	$\{x\}$	$A(A\{y\} \cap \mathbf{P})$
$*x = y$	$A\{x\} \cap \mathbf{P}$	$\boxed{Must(A)\{x\} \cap \mathbf{P}}$	$\boxed{A\{y\}}$
other	$\emptyset$	$\emptyset$	$\emptyset$

$$Must(R) = \bigcup_{z \in \mathbf{P}} \{z\} \times \begin{cases} \{w\} & R\{z\} = \{w\} \land w \neq ? \\ \emptyset & \text{otherwise} \end{cases}$$

Pointees of y in Ain_n are the targets of defined pointers

An Example of Flow Sensitive May Points-to Analysis



Assume that the program is type correct

n_1: $x = \&y$, $y = \&z$, $z = \&u$

n_2: $*z = y$

n_3: $z = y$

Weak Update n_4: $*u = \&x$

Strong Update n_5: $*y = \&y$

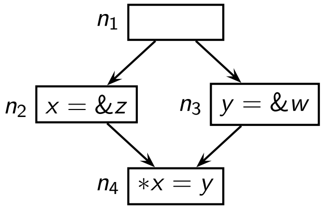
Tutorial Problems for Flow Sensitive Pointer Analysis (2)

Compute May and Must points-to information

```
if (...)
        p = &x;
else
        p = &y;
x = &a;
y = &b;
*p = &c;
*y = &a;
```

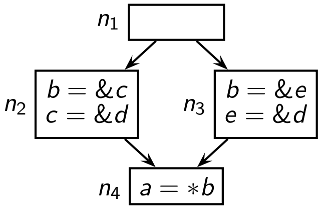
Non-Distributivity of Points-to Analysis

May Points-to



$z \rightarrow w$ is spurious

Must Points-to



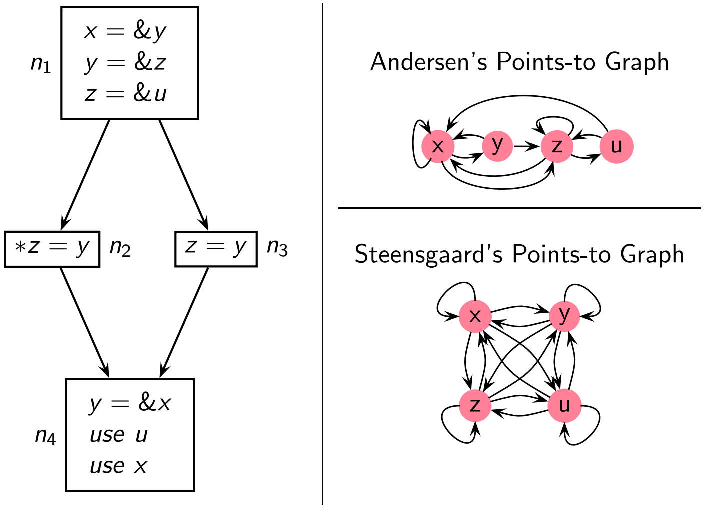
$a \rightarrow d$ is missing

An Outline of Pointer Analysis Coverage

- The larger perspective
- Comparing Points-to and Alias information
- Flow Insensitive Points-to Analysis
- Flow Sensitive Points-to Analysis
- Pointer Analyses: An Engineer's Landscape Next Topic
- Liveness Based Points-to Analysis
- Generalizations to Heap, Arrays, Pointer Arithmetic, and Unions

---

An Example of Flow Insensitive May Points-to Analysis



n_1: $x = \&y$, $y = \&z$, $z = \&u$

n_2: $*z = y$

n_3: $z = y$

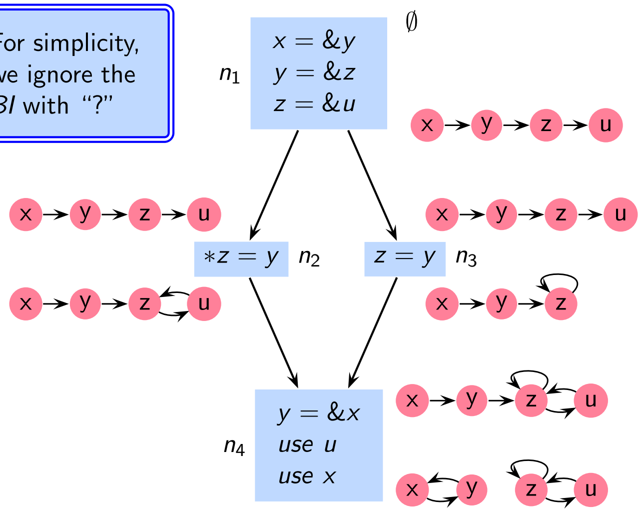
n_4: $y = \&x$, use u, use x

Andersen's Points-to Graph

Steensgaard's Points-to Graph

An Example of Flow Sensitive May Points-to Analysis

For simplicity, we ignore the *BI* with "?"



n_1: $x = \&y$, $y = \&z$, $z = \&u$

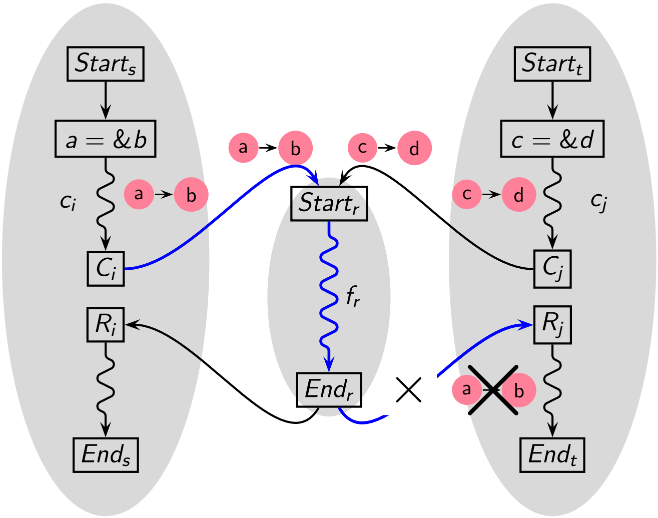
n_2: $*z = y$

n_3: $z = y$

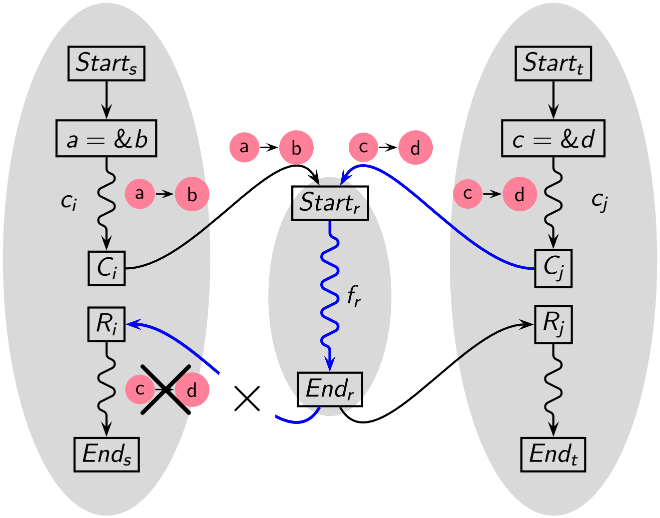
n_4: $y = \&x$, use u, use x

$\emptyset$

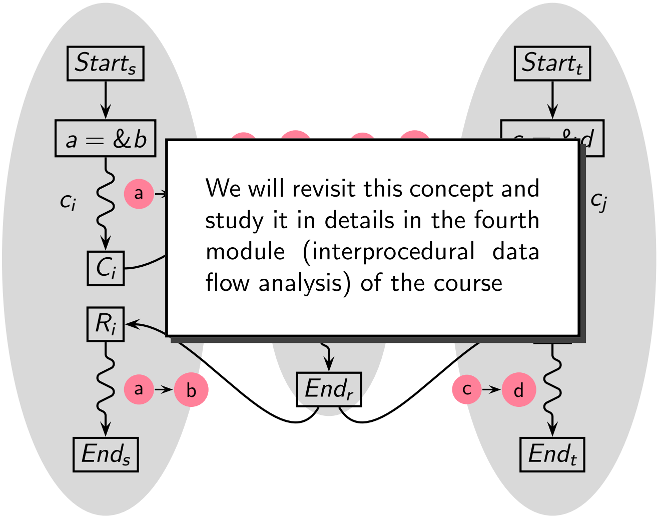
Context Sensitivity in Interprocedural Analysis



Context Sensitivity in Interprocedural Analysis



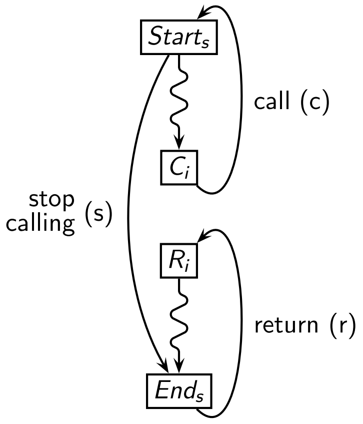
Context Sensitivity in Interprocedural Analysis



We will revisit this concept and study it in details in the fourth module (interprocedural data flow analysis) of the course

---

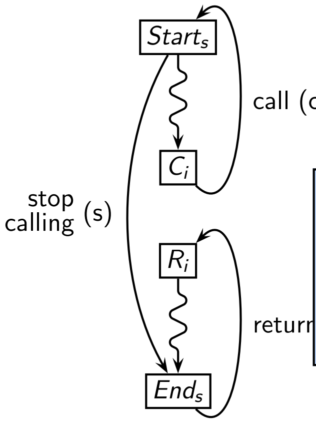
Context Sensitivity in the Presence of Recursion



- Paths from $Start_s$ to End_s should constitute a context free language c^nsr^n
- Many interprocedural analyses treat cycle of recursion as an SCC and approximate paths by a regular language c^*sr^*
- We do not know any practical points-to analysis that is fully context sensitive

  Most context sensitive approaches
  - either do not consider recursion, or
  - do not consider recursive pointer manipulation (e.g. “$p = p \rightarrow n$”), or
  - are context insensitive in recursion

Context Sensitivity in the Presence of Recursion



- Paths from $Start_s$ to End_s should constitute a context free language c^nsr^n
- Many interprocedural analyses treat cycle of recursion as an SCC and approximate paths by a regular language

We will revisit this concept and study it in details in the fourth module (interprocedural data flow analysis) of the course

al points-to

sensitive

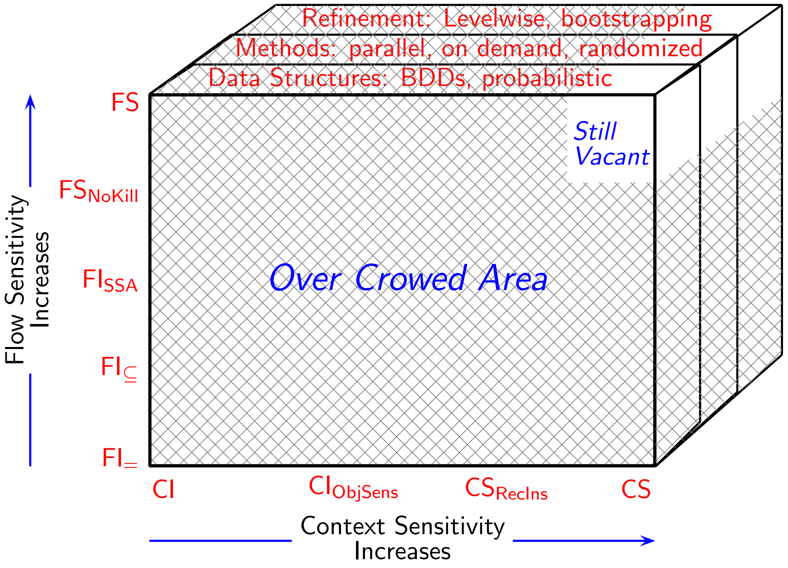
aches

ecursion, or

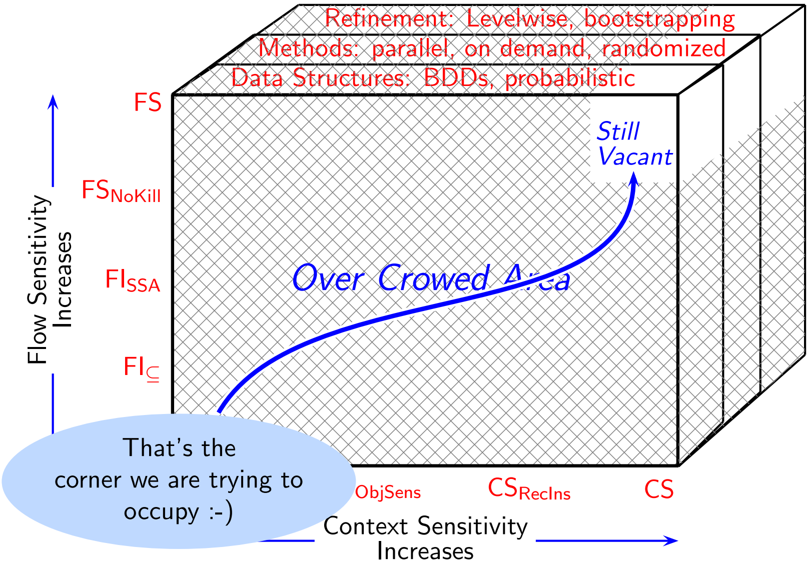
manipulation (e.g. “$p = p \rightarrow n$”), or

- are context insensitive in recursion

Pointer Analysis: An Engineer's Landscape



Pointer Analysis: An Engineer's Landscape



That's the corner we are trying to occupy :-)

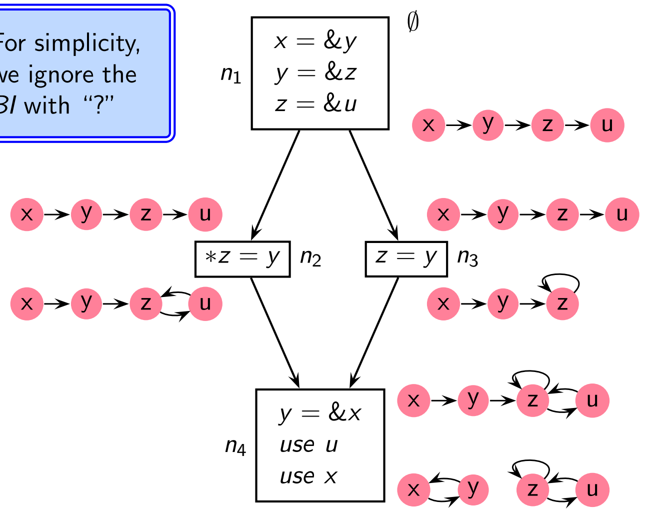
An Outline of Pointer Analysis Coverage

- The larger perspective
- Comparing Points-to and Alias information
- Flow Insensitive Points-to Analysis
- Flow Sensitive Points-to Analysis
- Pointer Analyses: An Engineer's Landscape
- Liveness Based Points-to Analysis Next Topic
- Generalizations to Heap, Arrays, Pointer Arithmetic, and Unions

---

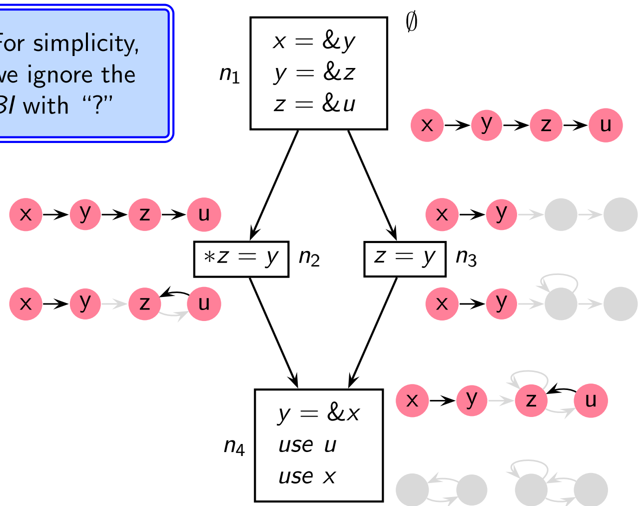
Our Motivating Example for FCPA

For simplicity, we ignore the *BI* with "?"



Is All This Information Useful?

For simplicity, we ignore the *BI* with "?"



The L and P of LFCPA

Mutual dependence of liveness and points-to information

- Define points-to information only for live pointers
- For pointer indirections, define liveness information using points-to information

---

The F and C of LFCPA

- Use call strings method for full flow and context sensitivity
- Use value contexts for efficient interprocedural analysis [Khedker-Karkare-CC-2008, Padhye-Khedker-SOAP-2013]

Use of Strong Liveness

- 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

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis *Generation of strong liveness*

	Def_n	$Kill_n$	$Pointee_n$	Ref_n	
				$Def_n \cap Lout_n \neq \emptyset$	*otherwise*
use x	$\emptyset$	$\emptyset$	$\emptyset$	$\{x\}$	$\{x\}$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$	$\emptyset$	$\emptyset$
$x = y$	$\{x\}$	$\{x\}$	$A\{y\}$	$\{y\}$	$\emptyset$
$x = *y$	$\{x\}$	$\{x\}$	$A(A\{y\} \cap \mathbf{P})$	$\{y\} \cup A\{y\} \cap \mathbf{P}$	$\emptyset$
$*x = y$	$A\{x\} \cap \mathbf{P}$	$Must(A)\{x\} \cap \mathbf{P}$	$A\{y\}$	$\{x, y\}$	$\{x\}$
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

- *Lin*/*Lout*: set of Live pointers, *Ain*/*Aout*: sets of mAy points-to pairs
- Ref_n, $Kill_n$, Def_n, and $Pointee_n$ are defined in terms of Ain_n

---

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis *Generation of strong liveness*

	Def_n	$Kill_n$	$Pointee_n$	$\boxed{Ref_n}$	
				$\boxed{Def_n \cap Lout_n \neq \emptyset}$	$\boxed{otherwise}$
use x	$\emptyset$	$\emptyset$	$\emptyset$	$\{x\}$	$\{x\}$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$	$\emptyset$	$\emptyset$
$x = y$	$\{x\}$	$\{x\}$	$A\{y\}$	$\{y\}$	$\emptyset$
$x = *y$	$\{x\}$	$\{x\}$	$A(A\{y\} \cap \mathbf{P})$	$\{y\} \cup A\{y\} \cap \mathbf{P}$	$\emptyset$
$*x = y$	$A\{x\} \cap \mathbf{P}$	$Must(A)\{x\} \cap \mathbf{P}$	$A\{y\}$	$\{x, y\}$	$\{x\}$
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Pointers that become live

Defined pointers must be live at the exit for the read pointers to become live

Some pointers are unconditionally live

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis *Generation of strong liveness*

	Def_n	$Kill_n$	$Pointee_n$	Ref_n	
				$Def_n \cap Lout_n \neq \emptyset$	*otherwise*
use x	$\emptyset$	$\emptyset$	$\emptyset$	$\boxed{\{x\}}$	$\boxed{\{x\}}$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$	$\emptyset$	$\emptyset$
$x = y$	$\{x\}$	$\{x\}$	$A\{y\}$	$\{y\}$	$\emptyset$
$x = *y$	$\{x\}$	$\{x\}$	$A(A\{y\} \cap \mathbf{P})$	$\{y\} \cup A\{y\} \cap \mathbf{P}$	$\emptyset$
$*x = y$	$A\{x\} \cap \mathbf{P}$	$Must(A)\{x\} \cap \mathbf{P}$	$A\{y\}$	$\{x, y\}$	$\{x\}$
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

x is unconditionally live

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis *Generation of strong liveness*

	Def_n	$Kill_n$	$Pointee_n$	Ref_n	
				$Def_n \cap Lout_n \neq \emptyset$	*otherwise*
use x	$\emptyset$	$\emptyset$	$\emptyset$	$\{x\}$	$\{x\}$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$	$\emptyset$	$\emptyset$
$x = y$	$\{x\}$	$\{x\}$	$A\{y\}$	$\boxed{\{y\}}$	$\emptyset$
$x = *y$	$\{x\}$	$\{x\}$	$A(A\{y\} \cap \mathbf{P})$	$\{y\} \cup A\{y\} \cap \mathbf{P}$	$\emptyset$
$*x = y$	$A\{x\} \cap \mathbf{P}$	$Must(A)\{x\} \cap \mathbf{P}$	$A\{y\}$	$\{x, y\}$	$\{x\}$
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

y is live if defined pointers are live

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis *Generation of strong liveness*

	Def_n	$Kill_n$	$Pointee_n$	Ref_n	
				$Def_n \cap Lout_n \neq \emptyset$	*otherwise*
use x	$\emptyset$	$\emptyset$	$\emptyset$	$\{x\}$	$\{x\}$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$	$\emptyset$	$\emptyset$
$x = y$	$\{x\}$	$\{x\}$	$A\{y\}$	$\{y\}$	$\emptyset$
$x = *y$	$\{x\}$	$\{x\}$	$A(A\{y\} \cap \mathbf{P})$	$\boxed{\{y\} \cup A\{y\} \cap \mathbf{P}}$	$\emptyset$
$*x = y$	$A\{x\} \cap \mathbf{P}$	$Must(A)\{x\} \cap \mathbf{P}$	$A\{y\}$	$\{x, y\}$	$\{x\}$
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

y and its pointees in Ain_n are live if defined pointers are live

---

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis *Generation of strong liveness*

	Def_n	$Kill_n$	$Pointee_n$	Ref_n	
				$Def_n \cap Lout_n \neq \emptyset$	*otherwise*
use x	$\emptyset$	$\emptyset$	$\emptyset$	$\{x\}$	$\{x\}$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$	$\emptyset$	$\emptyset$
$x = y$	$\{x\}$	$\{x\}$	$A\{y\}$	$\{y\}$	$\emptyset$
$x = *y$	$\{x\}$	$\{x\}$	$A(A\{y\} \cap \mathbf{P})$	$\{y\} \cup A\{y\} \cap \mathbf{P}$	$\emptyset$
$*x = y$	$A\{x\} \cap \mathbf{P}$	$Must(A)\{x\} \cap \mathbf{P}$	$A\{y\}$	$\boxed{\{x, y\}}$	$\boxed{\{x\}}$
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

y is live if defined pointers are live

x is unconditionally live

Deriving *Must* Points-to for LFCPA

For $*x = y$, unless the pointees of x are known

- points-to propagation should be blocked
- liveness propagation should be blocked

to ensure monotonicity

$$Must(R) = \bigcup_{x \in \mathbf{P}} \{x\} \times \begin{cases} \mathbb{V}\text{ar} & R\{x\} = \emptyset \vee R\{x\} = \{?\} \\ \{y\} & R\{x\} = \{y\} \wedge y \neq \,? \\ \emptyset & \text{otherwise} \end{cases}$$

LFCPA Data Flow Equations

$$Lout_n = \begin{cases} \emptyset & n \text{ is } End_p \\ \bigcup_{s \in succ(n)} Lin_s & \text{otherwise} \end{cases}$$

$$Lin_n = \left(Lout_n - Kill_n\right) \cup Ref_n$$

$$Ain_n = \begin{cases} Lin_n \times \{?\} & n \text{ is } Start_p \\ \left(\bigcup_{p \in pred(n)} Aout_p\right)\Bigg|_{Lin_n} & \text{otherwise} \end{cases}$$

$$Aout_n = \left(\left(Ain_n - \left(Kill_n \times \mathbb{V}\text{ar}\right)\right) \cup \left(Def_n \times Pointee_n\right)\right)\Bigg|_{Lout_n}$$

- $Lin/Lout$: set of Live pointers
- $Ain/Aout$: definitions remain unchanged except for restriction to liveness

LFCPA Data Flow Equations

$$Lout_n = \begin{cases} \emptyset & n \text{ is } End_p \\ \displaystyle\bigcup_{s \in succ(n)} Lin_s & \text{otherwise} \end{cases}$$

$$Lin_n = \left(Lout_n - \boxed{Kill_n}\right) \cup \boxed{Ref_n}$$

$$Ain_n = \begin{cases} Lin_n \times \{?\} & n \text{ is } Start_p \\ \left(\displaystyle\bigcup_{p \in pred(n)} Aout_p\right)\Bigg|_{Lin_n} & \text{otherwise} \end{cases}$$

$$Aout_n = \left(\left(Ain_n - \left(Kill_n \times \mathbb{V}\text{ar}\right)\right) \cup \left(Def_n \times Pointee_n\right)\right)\Bigg|_{Lout_n}$$

$Kill_n$ defined in terms of Ain_n

Ref_n defined in terms of Ain_n and $Lout_n$

- $Lin/Lout$: set of Live pointers
- $Ain/Aout$: definitions remain unchanged except for restriction to liveness

---

LFCPA Data Flow Equations

$$Lout_n = \begin{cases} \emptyset & n \text{ is } End_p \\ \displaystyle\bigcup_{s \in succ(n)} Lin_s & \text{otherwise} \end{cases}$$

$$Lin_n = \left(Lout_n - Kill_n\right) \cup Ref_n$$

$$Ain_n = \begin{cases} Lin_n \times \{?\} & n \text{ is } Start_p \\ \left(\displaystyle\bigcup_{p \in pred(n)} Aout_p\right)\Bigg|_{\boxed{Lin_n}} & \text{otherwise} \end{cases}$$

$$Aout_n = \left(\left(Ain_n - \left(Kill_n \times \mathbb{V}\text{ar}\right)\right) \cup \left(Def_n \times Pointee_n\right)\right)\Bigg|_{\boxed{Lout_n}}$$

Ain_n and $Aout_n$ are restricted to Lin_n and $Lout_n$

- $Lin/Lout$: set of Live pointers
- $Ain/Aout$: definitions remain unchanged except for restriction to liveness

---

LFCPA Data Flow Equations

$$Lout_n = \begin{cases} \emptyset & n \text{ is } End_p \\ \displaystyle\bigcup_{s \in succ(n)} Lin_s & \text{otherwise} \end{cases}$$

$$Lin_n = \left(Lout_n - Kill_n\right) \cup Ref_n$$

$$Ain_n = \begin{cases} \boxed{Lin_n \times \{?\}} & n \text{ is } Start_p \\ \left(\displaystyle\bigcup_{p \in pred(n)} Aout_p\right)\Bigg|_{Lin_n} & \text{otherwise} \end{cases}$$

$$Aout_n = \left(\left(Ain_n - \left(Kill_n \times \mathbb{V}\text{ar}\right)\right) \cup \left(Def_n \times Pointee_n\right)\right)\Bigg|_{Lout_n}$$

BI restricted to live pointers

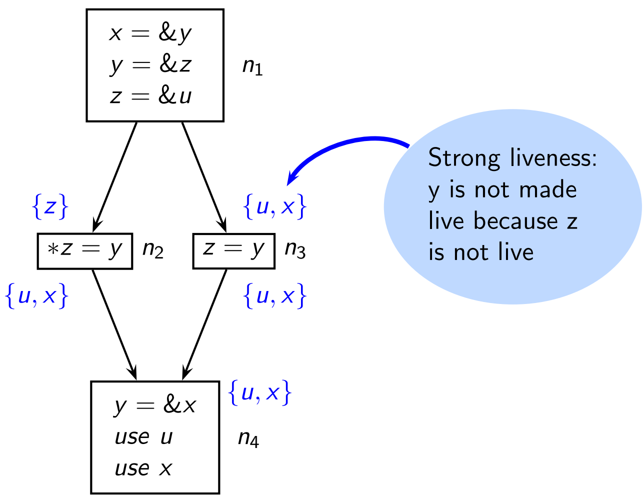
- $Lin/Lout$: set of Live pointers
- $Ain/Aout$: definitions remain unchanged except for restriction to liveness

---

Motivating Example Revisited

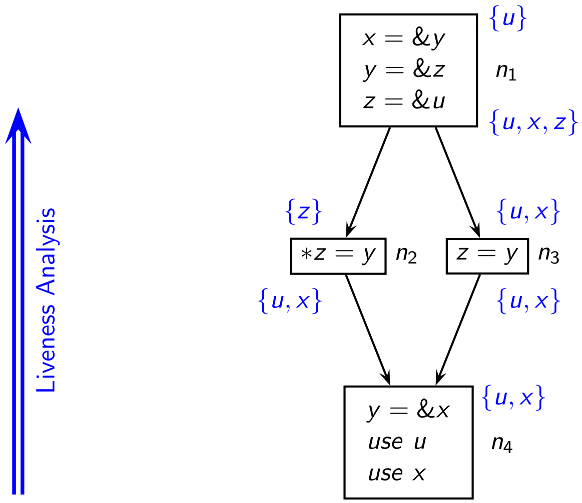
- For convenience, we show complete sweeps of liveness and points-to analysis repeatedly
- This is not required by the computation
- The data flow equations define a single fixed point computation

First Round of Liveness Analysis and Points-to Analysis

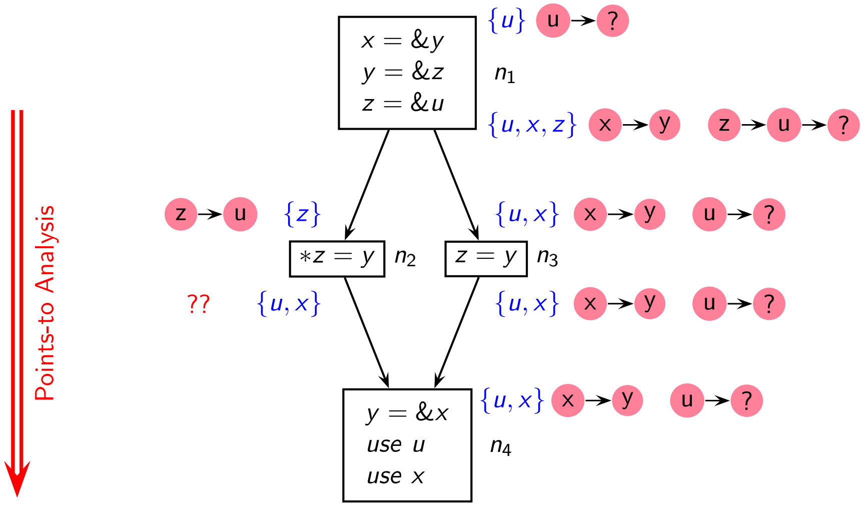


Strong liveness: y is not made live because z is not live

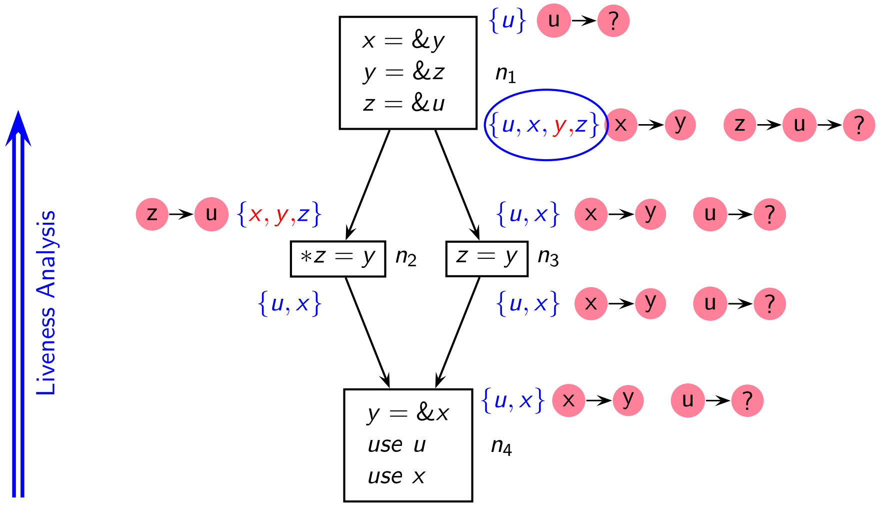
First Round of Liveness Analysis and Points-to Analysis



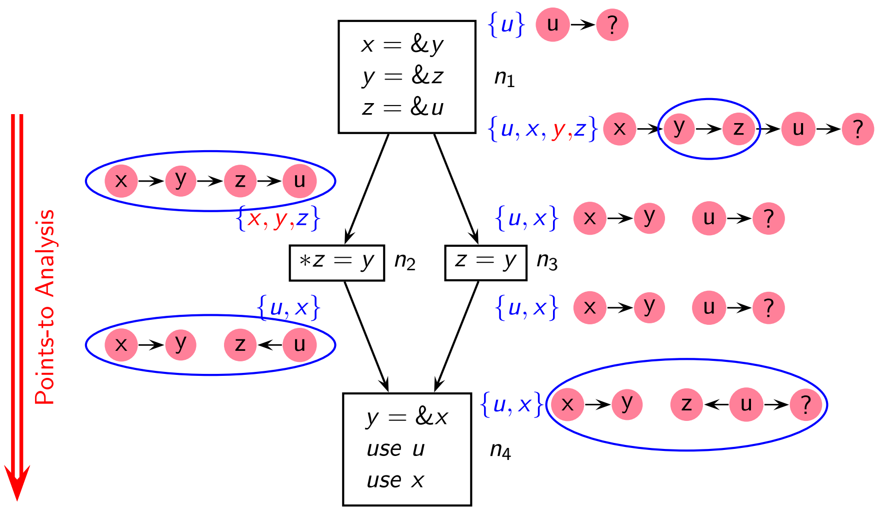
First Round of Liveness Analysis and Points-to Analysis



Second Round of Liveness Analysis and Points-to Analysis



Second Round of Liveness Analysis and Points-to Analysis



LFCPA Implementation

- LTO framework of GCC 4.6.0
- Naive prototype implementation
  (Points-to sets implemented using linked lists)
- Implemented FCPA without liveness for comparison
- Comparison with GCC's flow and context insensitive method
- SPEC 2006 benchmarks

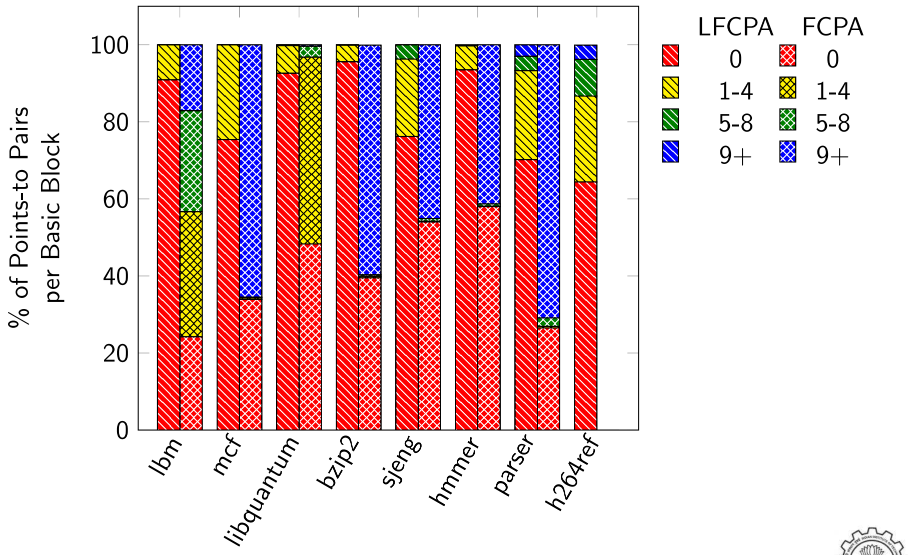
Analysis Time

Program	kLoC	Call Sites	Time in milliseconds			
			L-FCPA		FCPA	GPTA
			Liveness	Points-to		
lbm	0.9	33	0.55	0.52	1.9	5.2
mcf	1.6	29	1.04	0.62	9.5	3.4
libquantum	2.6	258	2.0	1.8	5.6	4.8
bzip2	3.7	233	4.5	4.8	28.1	30.2
parser	7.7	1123	1.2×10^3	145.6	4.3×10^5	422.12
sjeng	10.5	678	858.2	99.0	3.2×10^4	38.1
hmmer	20.6	1292	90.0	62.9	2.9×10^5	246.3
h264ref	36.0	1992	2.2×10^5	2.0×10^5	?	4.3×10^3

Unique Points-to Pairs

Program	kLoC	Call Sites	Unique points-to pairs		
			L-FCPA	FCPA	GPTA
lbm	0.9	33	12	507	1911
mcf	1.6	29	41	367	2159
libquantum	2.6	258	49	119	2701
bzip2	3.7	233	60	210	8.8×10^4
parser	7.7	1123	531	4196	1.9×10^4
sjeng	10.5	678	267	818	1.1×10^4
hmmer	20.6	1292	232	5805	1.9×10^6
h264ref	36.0	1992	1683	?	1.6×10^7

Points-to Information is Small and Sparse



LFCPA Observations

- Usable pointer information is very small and sparse
- Data flow propagation in real programs seems to involve only a small subset of all possible data flow values
- Earlier approaches reported inefficiency and non-scalability because they computed far more information than the actual usable information

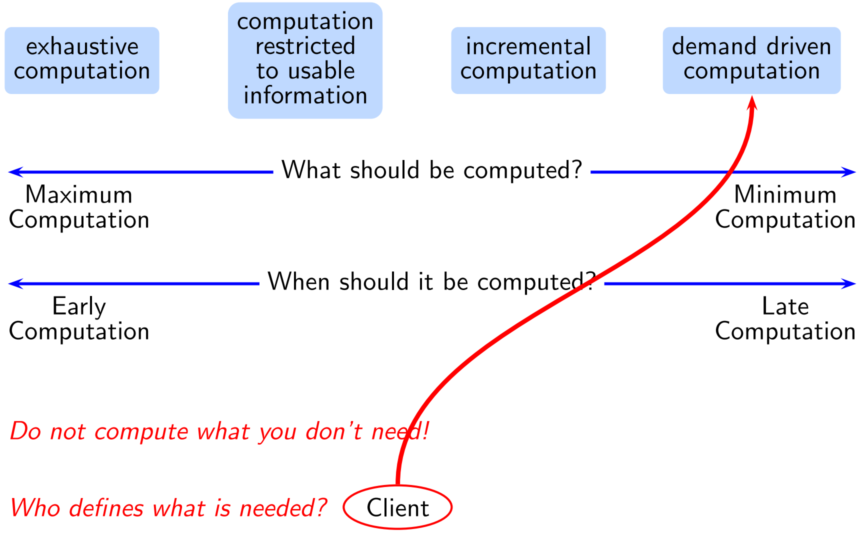
---

LFCPA Conclusions

- Building quick approximations and compromising on precision may not be necessary for efficiency
- Building clean abstractions to separate the necessary information from redundant information is much more significant

  Our experience of points-to analysis shows that
  - Use of liveness reduced the pointer information . . .
  - which reduced the number of contexts required . . .
  - which reduced the liveness and pointer information . . .
- Approximations should come *after* building abstractions rather than *before*

LFCPA Lessons: The Larger Perspective



exhaustive computation

computation restricted to usable information

incremental computation

demand driven computation

What should be computed?

Maximum Computation

Minimum Computation

When should it be computed?

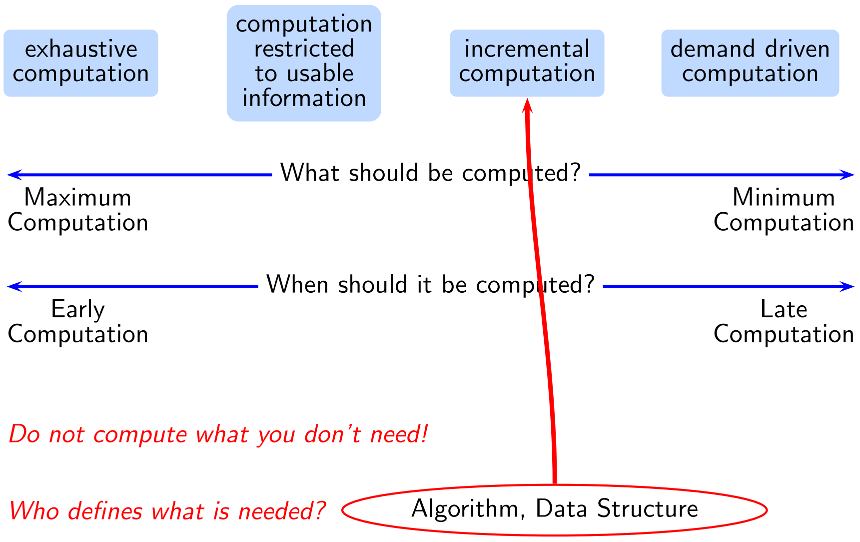
Early Computation

Late Computation

Do not compute what you don't need!

Who defines what is needed? Client

LFCPA Lessons: The Larger Perspective



exhaustive computation

computation restricted to usable information

incremental computation

demand driven computation

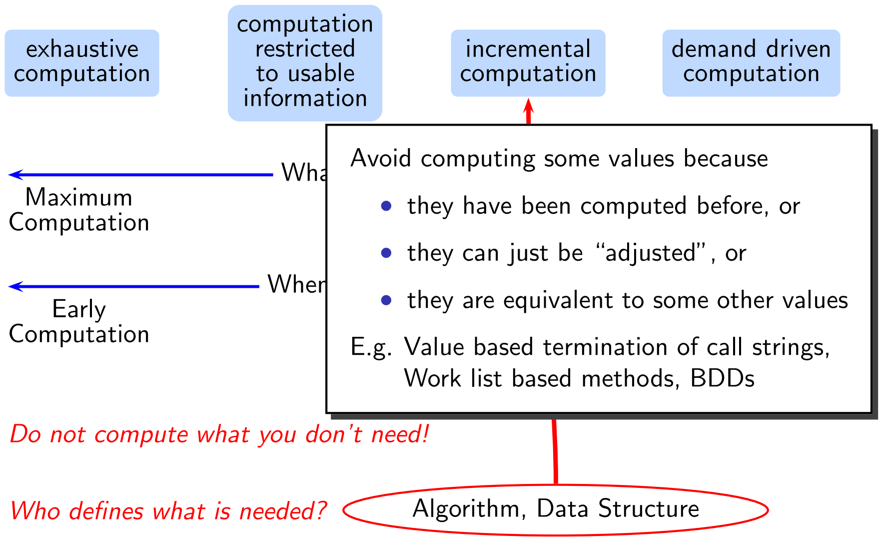
Maximum Computation ← What should be computed? → Minimum Computation

Early Computation ← When should it be computed? → Late Computation

Do not compute what you don't need!

Who defines what is needed? Algorithm, Data Structure

LFCPA Lessons: The Larger Perspective



exhaustive computation

computation restricted to usable information

incremental computation

demand driven computation

Maximum Computation ← Wha

Early Computation ← When

Avoid computing some values because

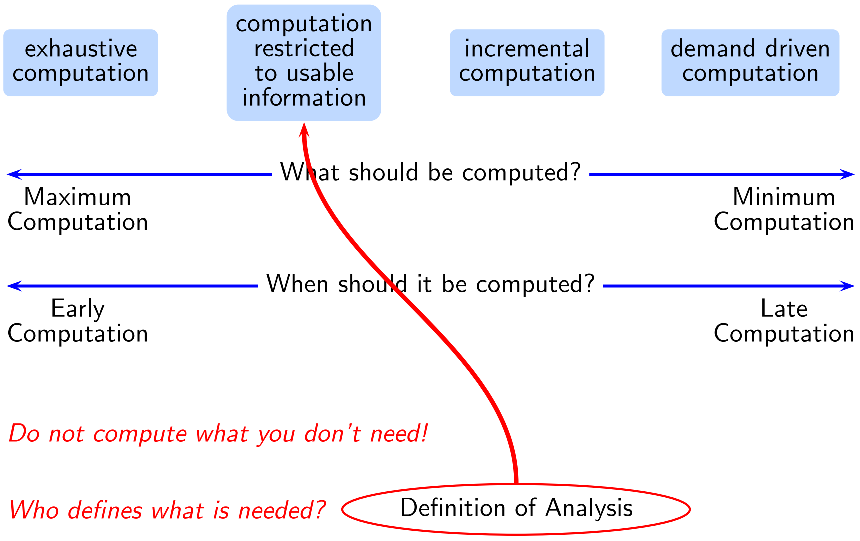
- they have been computed before, or
- they can just be "adjusted", or
- they are equivalent to some other values

E.g. Value based termination of call strings, Work list based methods, BDDs

Do not compute what you don't need!

Who defines what is needed? Algorithm, Data Structure

LFCPA Lessons: The Larger Perspective



exhaustive computation

computation restricted to usable information

incremental computation

demand driven computation

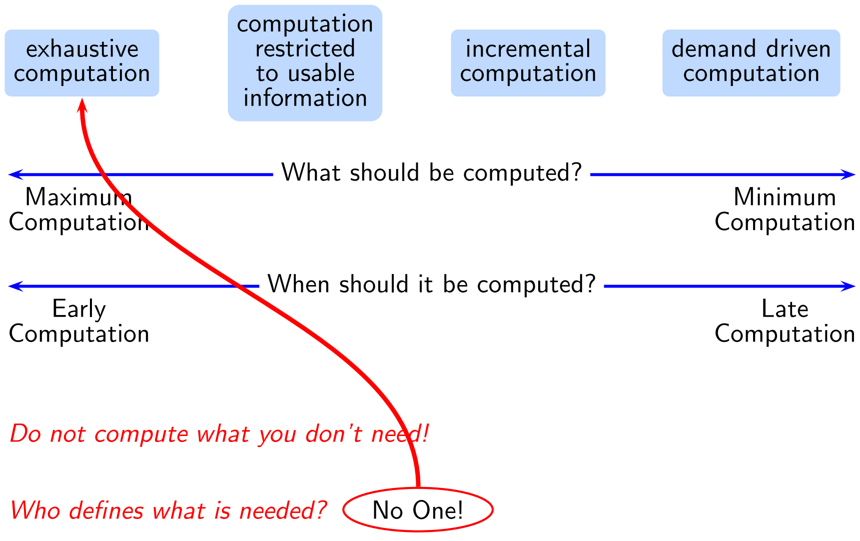
Maximum Computation ← What should be computed? → Minimum Computation

Early Computation ← When should it be computed? → Late Computation

Do not compute what you don't need!

Who defines what is needed? Definition of Analysis

LFCPA Lessons: The Larger Perspective



exhaustive computation

computation restricted to usable information

incremental computation

demand driven computation

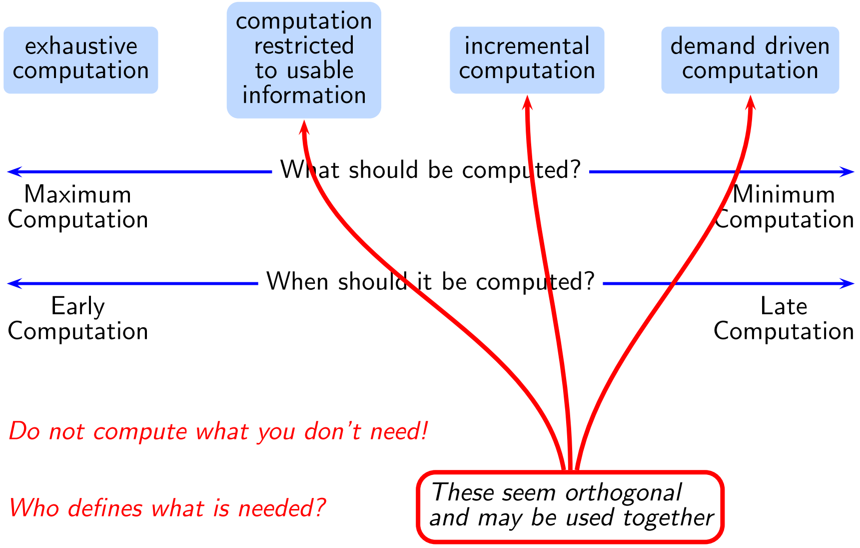
Maximum Computation ← What should be computed? → Minimum Computation

Early Computation ← When should it be computed? → Late Computation

Do not compute what you don't need!

Who defines what is needed? No One!

LFCPA Lessons: The Larger Perspective



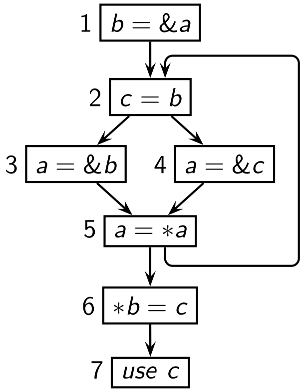
Do not compute what you don't need!

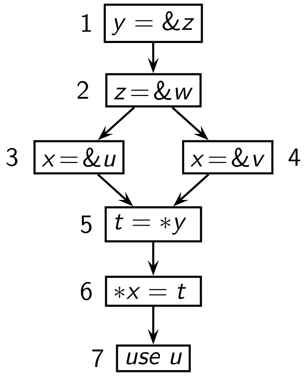
Who defines what is needed?

These seem orthogonal and may be used together

Tutorial Problems for FCPA and LFCPA

- Perform may points-to analysis by deriving must info using "?" in *BI*
- Perform liveness based points-to analysis





An Outline of Pointer Analysis Coverage

- The larger perspective
- Comparing Points-to and Alias information
- Flow Insensitive Points-to Analysis
- Flow Sensitive Points-to Analysis
- Pointer Analyses: An Engineer's Landscape
- Liveness Based Points-to Analysis
- Generalizations to Heap, Arrays, Pointer Arithmetic, and Unions

Next Topic

Original LFCPA Formulation

Data flow equations
Lin/Lout, Ain/Aout

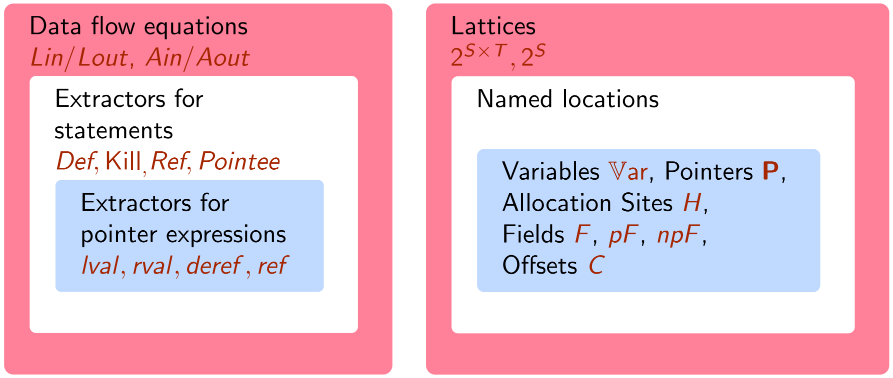
Extractors for statements
Def, Kill, *Ref*, *Pointee*

Lattices
$2^{\mathbf{P} \times \mathbb{V}\mathrm{ar}}, 2^{\mathbf{P}}$

Named locations

Variables $\mathbb{V}$ar, Pointers **P**,

Formulating Generalizations in LFCPA



Data flow equations
Lin/*Lout*, *Ain*/*Aout*

Extractors for statements
Def, Kill, *Ref*, *Pointee*

Extractors for pointer expressions
lval, *rval*, *deref*, *ref*

Lattices
$2^{S\times T}, 2^{S}$

Named locations

Variables $\mathbb{V}$ar, Pointers **P**,
Allocation Sites *H*,
Fields *F*, *pF*, *npF*,
Offsets *C*

Generalization for Heap and Structures

- Grammar.

$$\boxed{\begin{array}{l}\alpha := malloc \mid \&\beta \mid \beta \\ \beta := x \mid \beta.f \mid \beta \rightarrow f \mid *\beta\end{array}}$$

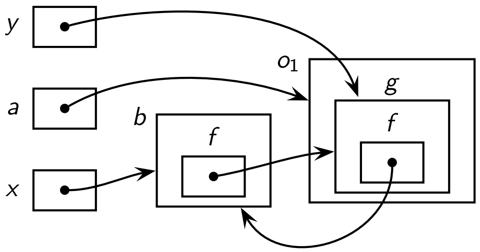
where α is a pointer expression, x is a variable, and f is a field

- Memory model: Named memory locations. No numeric addresses

$$\begin{array}{lll} S & = \mathbf{P} \cup H \cup S_p & \text{(source locations)} \\ T & = \mathbb{V}\text{ar} \cup H \cup S_m \cup \{?\} & \text{(target locations)} \\ S_p & = R \times npF^* \times pF & \text{(pointers in structures)} \\ S_m & = R \times npF^* \times (pF \cup npF) & \text{(other locations in structures)} \end{array}$$

Named Locations for Pointer Expressions

```
typedef struct B
{   ...
    struct B *f;
} sB;
typedef struct A
{   ...
    struct B g;
} sA;
      sA *a;
      sB *x, *y, b;
1.  a = (sA*) malloc
         (sizeof(sA));
2.  y = &a->g;
3.  b.f = y;
4.  x = &b;
5.  y.f = &x;
6.  return x->f->f;
```



Pointer Expression	l-value	r-value
x	x	b
$x \rightarrow f$	$b.f$	$o_1.g.f$
$x \rightarrow f \rightarrow f$	$o_1.g.f$	b

L- and R-values of Pointer Expressions

$$lval(\alpha, A) = \begin{cases} \{\sigma\} & (\alpha \equiv \sigma) \wedge (\sigma \in \mathbb{V}\text{ar}) \\ \{\sigma.f \mid \sigma \in lval(\beta, A)\} & \alpha \equiv \beta.f \\ \{\sigma.f \mid \sigma \in rval(\beta, A), \sigma \neq ?\} & \alpha \equiv \beta \rightarrow f \\ \{\sigma \mid \sigma \in rval(\beta, A), \sigma \neq ?\} & \alpha \equiv *\beta \\ \emptyset & \text{otherwise} \end{cases}$$

$$rval(\alpha, A) = \begin{cases} lval(\beta, A) & \alpha \equiv \&\beta \\ \{o_i\} & \alpha \equiv malloc \wedge o_i = get\_heap\_loc() \\ A(lval(\alpha, A) \cap S) & \text{otherwise} \end{cases}$$

Defining Extractor Functions

- Pointer assignment statement $lhs_n = rhs_n$

$$\begin{aligned}
Def_n &= lval(lhs_n, Ain_n)\\
\text{Kill}_n &= lval\,(lhs_n, Must(Ain_n))\\
Ref_n &= \begin{cases} deref\,(lhs_n, Ain_n) & Def_n \cap Lout_n = \emptyset\\ deref\,(lhs_n, Ain_n) \cup ref\,(rhs_n, Ain_n) & \text{otherwise}\end{cases}\\
Pointee_n &= rval(rhs_n, Ain_n)
\end{aligned}$$

- *Use* α statement

$$\begin{aligned}
Def_n = \text{Kill}_n = Pointee_n &= \emptyset\\
Ref_n &= ref\,(\alpha, Ain_n)
\end{aligned}$$

- Any other statement

$$Def_n = \text{Kill}_n = Ref_n = Pointee_n = \emptyset$$

Extensions for Handling Arrays and Pointer Arithmetic

- Grammar.

$$\boxed{\begin{aligned}\alpha &:= malloc \mid \&\beta \mid \beta \mid \&\beta + e\\ \beta &:= x \mid \beta.f \mid \beta \rightarrow f \mid *\beta \mid \beta[e] \mid \beta + e\end{aligned}}$$

- Memory model: Named memory locations. No numeric addresses
  - No address calculation
  - R-values of index expressions retained for each dimension
    If $rval(x) = 10$, then $lval(a.f[5][2 + x].g) = a.f.5.12.g$
  - Sizes of the array elements ignored

$$\begin{aligned}
S &= \mathbf{P} \;\cup\; H \;\cup\; G_p && \text{(source locations)}\\
T &= \mathbb{V}\text{ar} \;\cup\; H \;\cup\; G_m \;\cup\; \{?\} && \text{(target locations)}\\
G_p &= R \times (C \cup npF)^* \times (C \cup pF) && \text{(pointers in aggregates)}\\
G_m &= R \times (C \cup npF)^* \times (C \cup pF \cup npF) && \text{(locations in aggregates)}
\end{aligned}$$

Extending L-Value Computation to Arrays and Pointer Arithmetic

- Pointer arithmetic does not have an l-value
- For handling arrays
  - evaluate index expressions using *evale* and accumulate offsets
  - if e cannot be evaluated at compile time, $evale = \perp_{eval}$
    (i.e. array accesses in that dimension are treated as index-insensitive)

$$lval(\alpha, A) = \begin{cases}
\{\sigma\} & (\alpha \equiv \sigma) \wedge (\sigma \in \mathbb{V}\text{ar})\\
\{\sigma.f \mid \sigma \in lval(\beta, A)\} & \alpha \equiv \beta.f\\
\{\sigma.f \mid \sigma \in rval(\beta, A), \sigma \neq ?\} & \alpha \equiv \beta \rightarrow f\\
\{\sigma \mid \sigma \in rval(\beta, A), \sigma \neq ?\} & \alpha \equiv *\beta\\
\{\sigma.evale \mid \sigma \in lval(\beta, A)\} & \alpha \equiv \beta[e]\\
\emptyset & \text{otherwise}
\end{cases}$$

Extending R-Value Computation to Arrays and Pointer Arithmetic

For handling pointer arithmetic

- If the r-value of the pointer is an array location, add *evale* to the offset
- Otherwise, over-approximate the pointees to all possible locations

$$rval(\alpha, A) = \begin{cases}
lval(\beta, A) & \alpha \equiv \&\beta\\
\{o_i\} & \alpha \equiv malloc \wedge o_i = get\_heap\_loc()\\
T & (\alpha \equiv \beta + e) \wedge\\
 & (\exists \sigma \in rval(\beta, A), \sigma \not\equiv \sigma'.c, \sigma' \in T, c \in C)\\
\bigcup \{\sigma.(c + evale)\} & (\alpha \equiv \beta + e) \wedge\\
 & (\sigma.c \in rval(\beta, A)) \wedge (c \in C)\\
A(lval(\alpha, A) \cap S) & \text{otherwise}
\end{cases}$$

Part 6

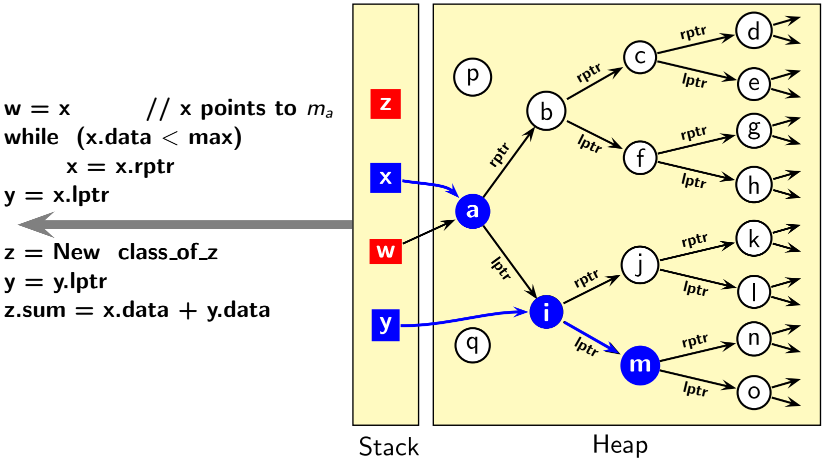
Heap Reference Analysis

Motivating Example for Heap Liveness Analysis

If the while loop is not executed even once.

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

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

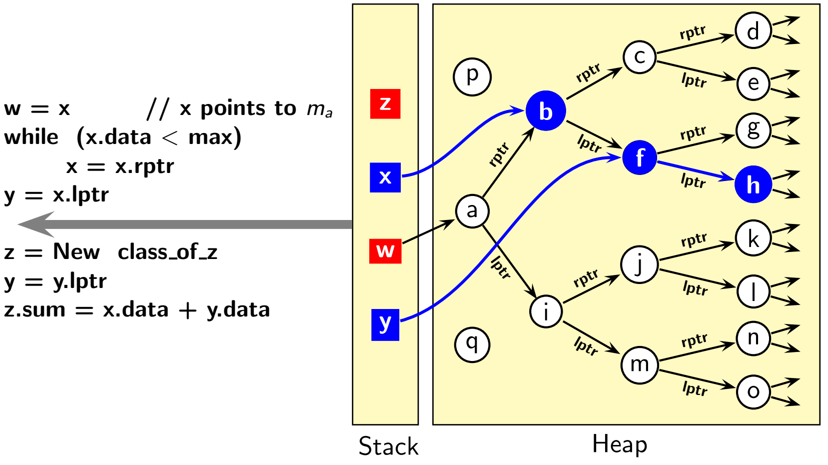


Motivating Example for Heap Liveness Analysis

If the while loop is executed once.

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

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

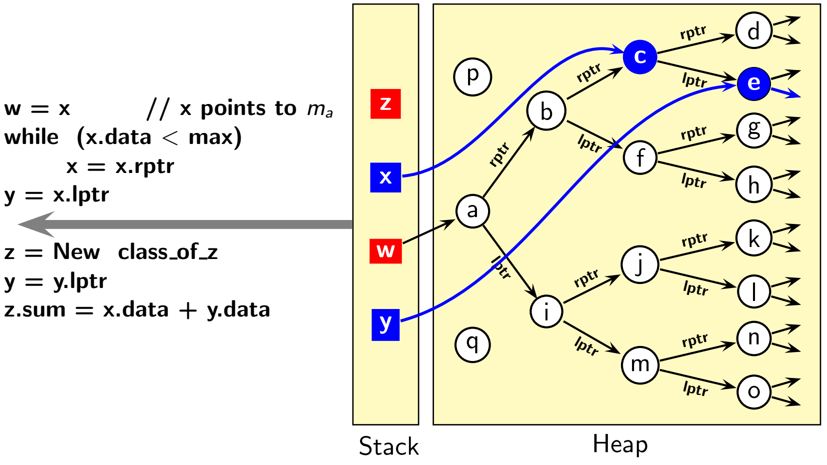


Motivating Example for Heap Liveness Analysis

If the while loop is executed twice.

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

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



The Moral of the Story

- Mappings between access expressions and l-values keep changing
- This is a *rule* for heap data

  For stack and static data, it is an *exception*!
- Static analysis of programs has made significant progress for stack and static data.

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

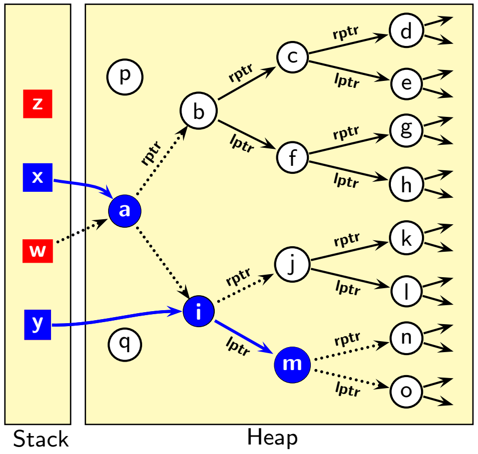
Our Solution

```
                        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
```

Our Solution

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

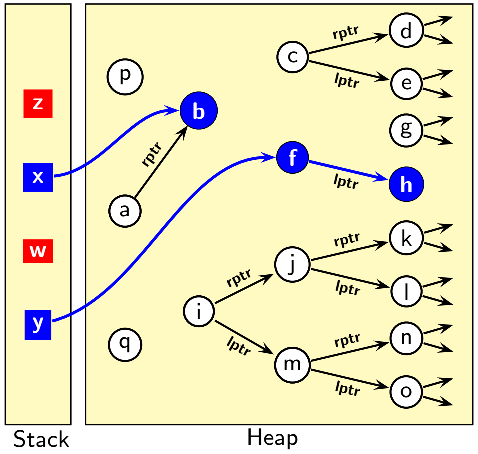
While loop is not executed even once



Our Solution

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

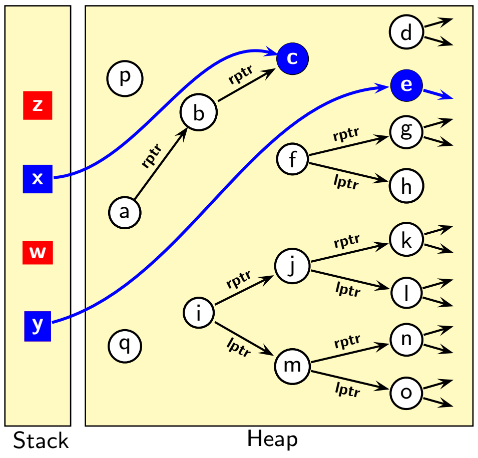
While loop is executed once



Our Solution

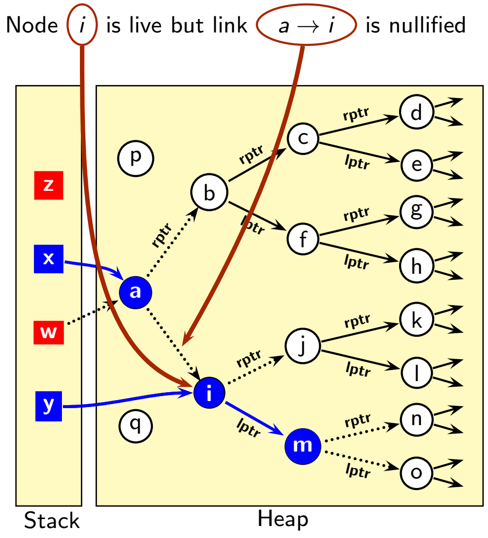
```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {     x.lptr = null
3         x = x.rptr      }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New class_of_z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

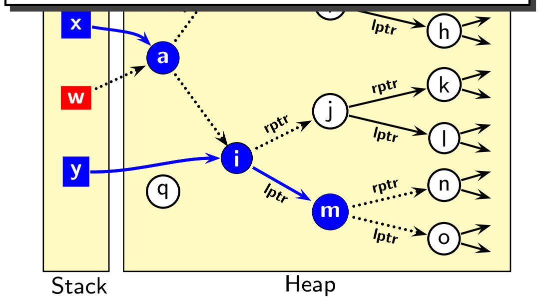
```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {     x.lptr = null
3         x = x.rptr      }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New class_of_z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

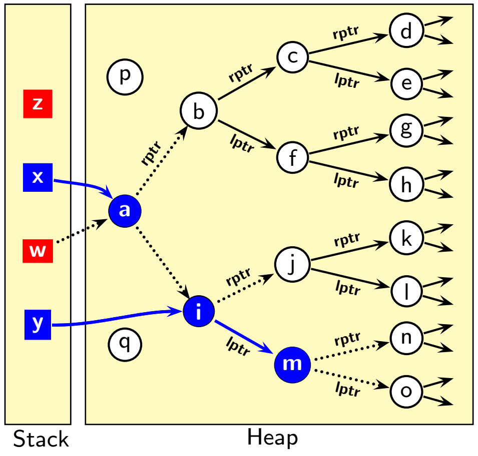
```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {     x.lptr = null
3         x = x.rptr      }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New class_of_z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



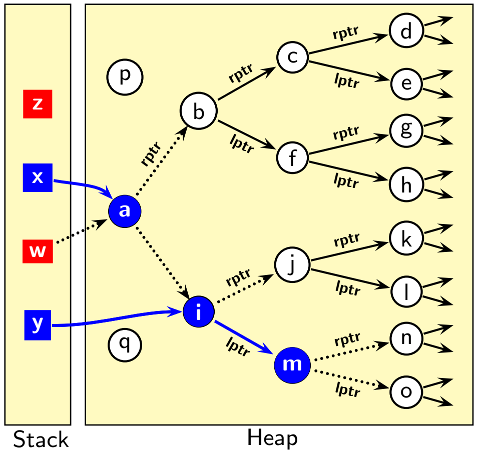
Some Observations

```
    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
```



Some Observations

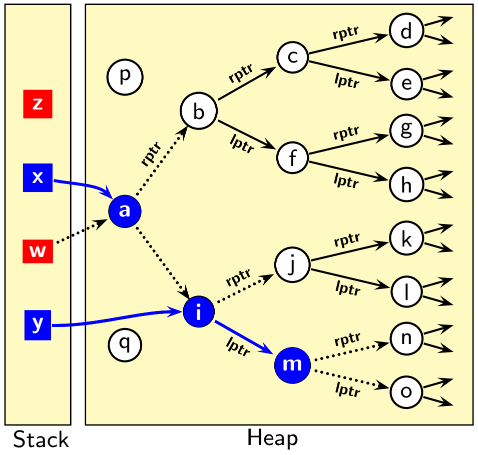
```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {      x.lptr = null
3          x = x.rptr        }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New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



Some Observations

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

New access expressions are created.
Can they cause exceptions?



An Overview of Heap Reference Analysis

- A reference (called a *link*) can be represented by an *access path*.

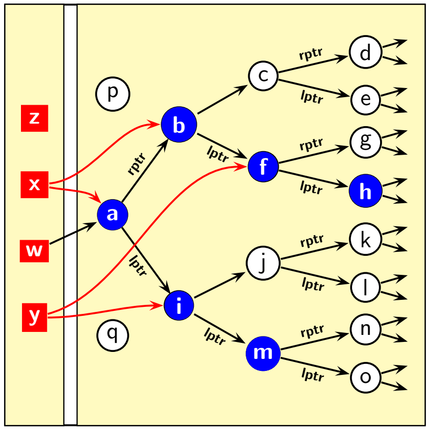
  Eg. "$x \rightarrow$ lptr $\rightarrow$ rptr"
- A link may be accessed in multiple ways
- Setting links to null
  - *Alias Analysis*. Identify all possible ways of accessing a link
  - *Liveness Analysis*. For each program point, identify "dead" links (i.e. links which are not accessed after that program point)
  - *Availability and Anticipability Analyses*. Dead links should be reachable for making null assignment.
  - *Code Transformation*. Set "dead" links to null

Assumptions

For simplicity of exposition

- Java model of heap access
  - Root variables are on stack and represent references to memory in heap.
  - Root variables cannot be pointed to by any reference.
- Simple extensions for C++
  - Root variables can be pointed to by other pointers.
  - Pointer arithmetic is not handled.

Key Idea #1 : Access Paths Denote Links



- Root variables : x, y, z
- Field names : rptr, lptr
- Access path : $x \rightarrow$rptr$\rightarrow$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

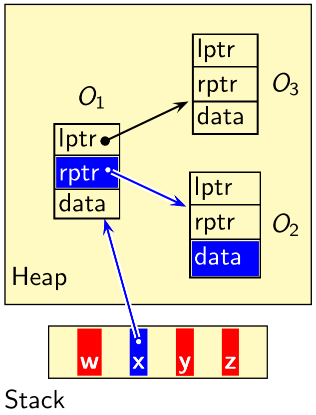
---

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	$x, x \rightarrow$rptr
if (x.rptr.data < sum)	x, O_1, O_2	$x, x \rightarrow$rptr

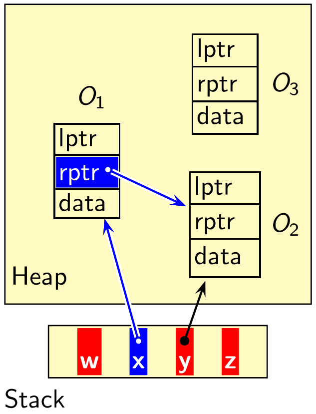


What Makes a Link Live?

Assuming that a statement 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

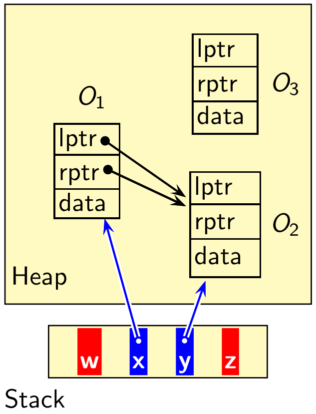


What Makes a Link Live?

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
$y = x$.rptr	x, O_1	x, x.rptr
x.lptr $= y$	x, O_1, y	x, y

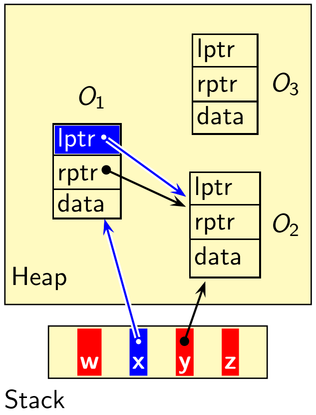


What Makes a Link Live?

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

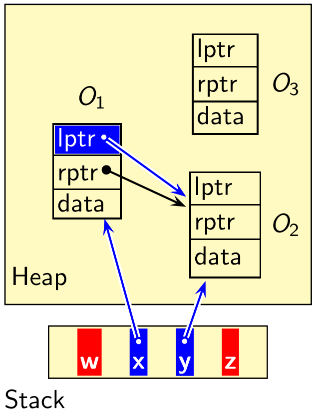


What Makes a Link Live?

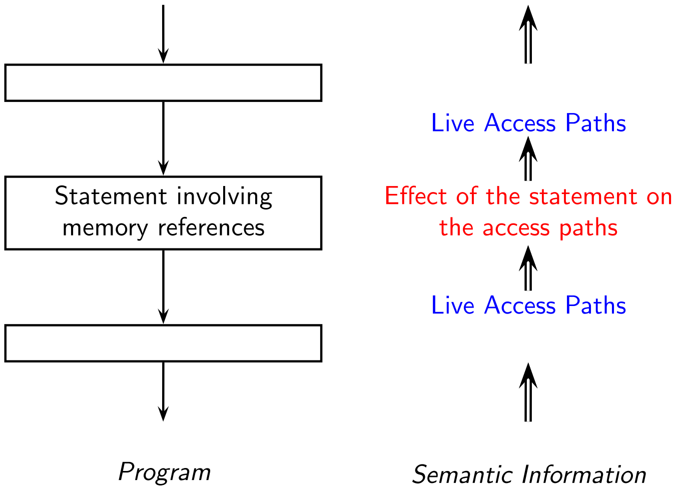
Assuming that a statement must be executed, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

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



Statement involving memory references

Live Access Paths

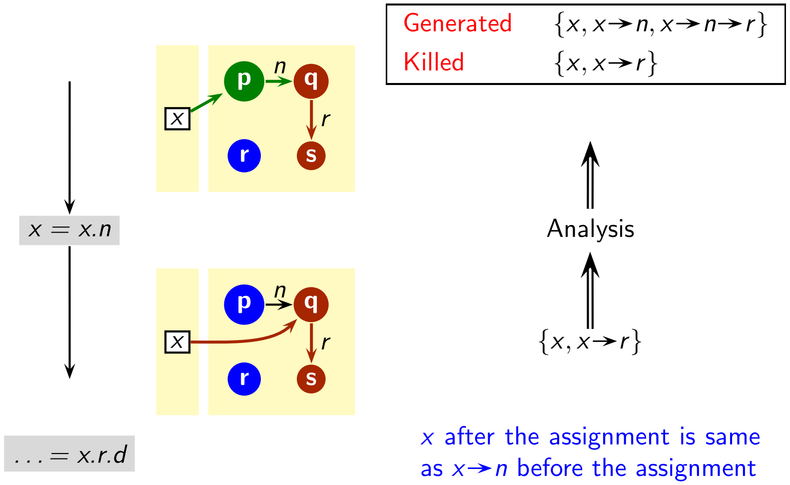
Effect of the statement on the access paths

Live Access Paths

Program

Semantic Information

Key Idea #2 : Transfer of Access Paths



$x = x.n$

$\ldots = x.r.d$

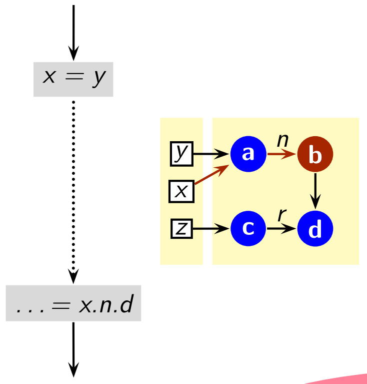
Generated	$\{x, x \rightarrow n, x \rightarrow n \rightarrow r\}$
Killed	$\{x, x \rightarrow r\}$

Analysis

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

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

Key Idea #3 : Liveness Closure Under Link Aliasing



$x = y$

$\ldots = x.n.d$

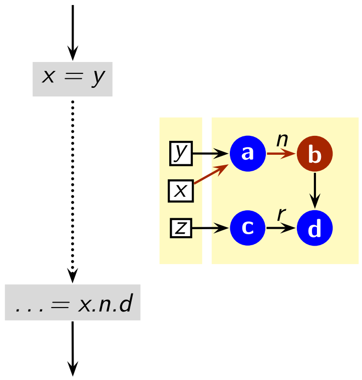
x and y are node aliases

$x.n$ 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



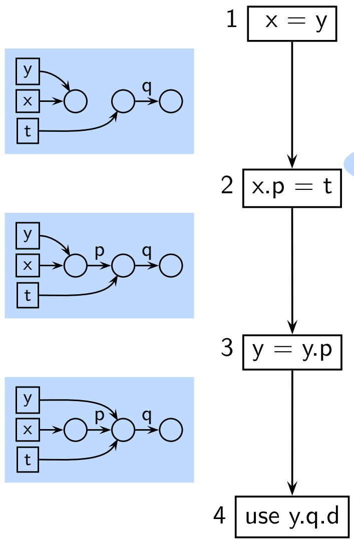
$x = y$

$\ldots = x.n.d$

$x \rightarrow n$ 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



1 x = y

2 x.p = t

3 y = y.p

4 use y.q.d

Explicit Liveness | *Required Liveness*

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

Spurious

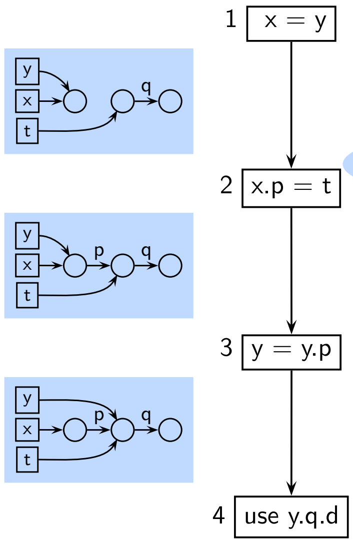
Missing

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

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

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

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*

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

Spurious

Missing

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

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

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

$$In_n = \left(Out_n - \text{Kill}_n(Out_n)\right) \cup \text{Gen}_n(Out_n)$$

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

Flow Functions for Explicit Liveness Analysis

Let A denote May Aliases at the exit of node n

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

Flow Functions for Explicit Liveness Analysis

Let A denote May Aliases at the exit of node n

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

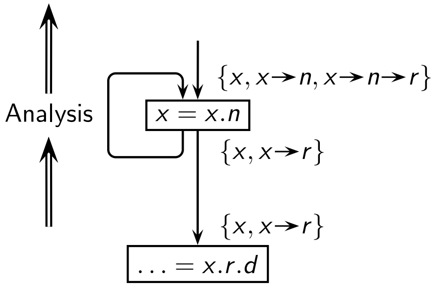
- Why is $y \notin \text{Gen}_n(X)$ for $x.f = y$ when $x \notin X$?

  If $\nexists x \in Out_n$, 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 $\nexists x\rightarrow\sigma \in Out_n$, we can do dead code elimination
- Why is $x \notin \text{Gen}_n(X)$ for $x.f = y$?
  - If $\nexists x\rightarrow f\rightarrow\sigma \in Out_n$, 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


Analysis

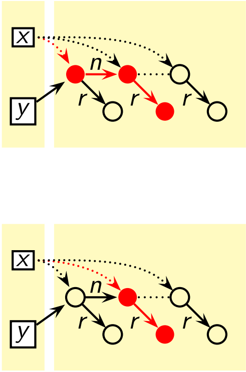
$\{x, x\rightarrow n, x\rightarrow n\rightarrow r\}$

$x = x.n$

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

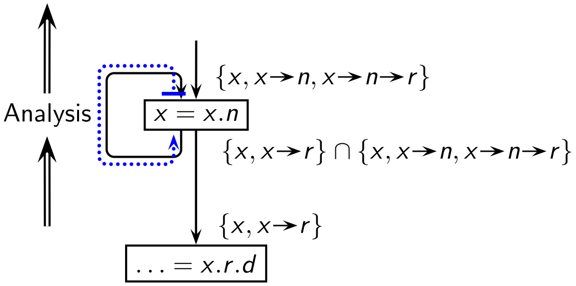
$\{x, x\rightarrow r\}$

$\ldots = x.r.d$



Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An *All Paths* problem


Analysis

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

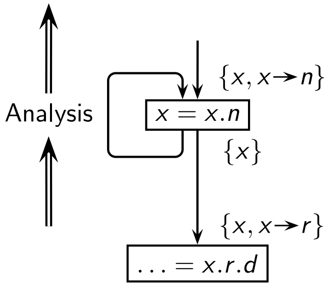
$x = x.n$

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

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

$\ldots = x.r.d$

Computing Explicit Liveness Using Sets of Access Paths


Analysis

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

$x = x.n$

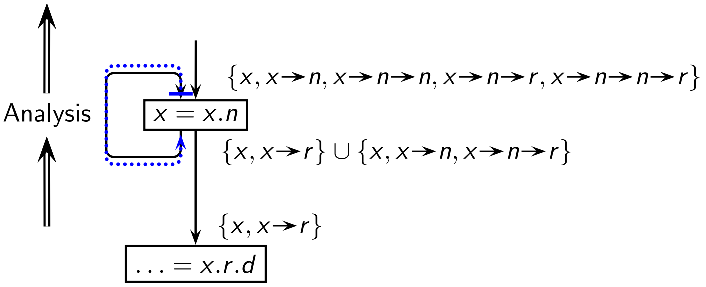
$\{x\}$

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

$\ldots = x.r.d$

Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



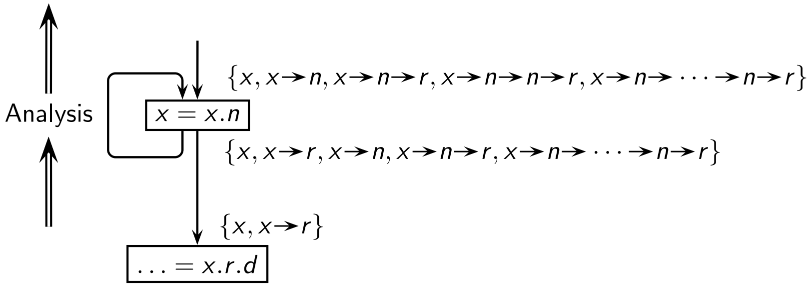
$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow r, x\rightarrow n\rightarrow n\rightarrow r\}$

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

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

Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



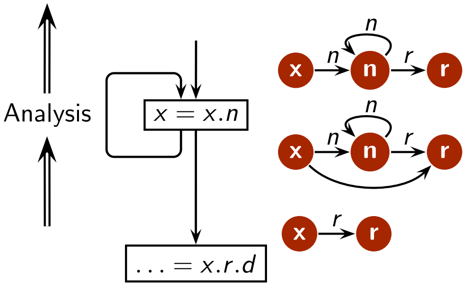
$\{x, x\rightarrow n, x\rightarrow n\rightarrow r, x\rightarrow n\rightarrow n\rightarrow r, x\rightarrow n\rightarrow \cdots \rightarrow n\rightarrow r\}$

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

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

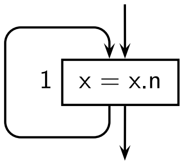
Infinite Number of Unbounded Access Paths

Key Idea #5: Using Graphs as Data Flow Values



Finite Number of Bounded Structures

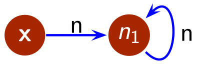
Key Idea #6 : Include Program Point in Graphs

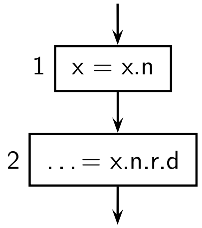


$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow n, \ldots\}$

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

Access Graph :

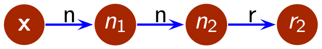




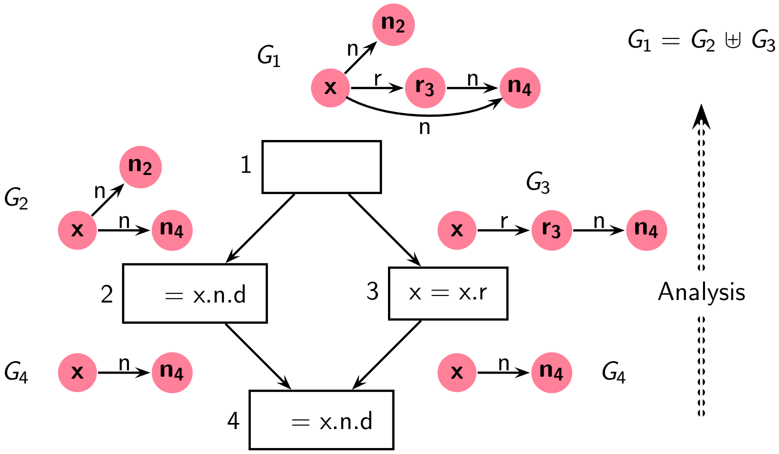
$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow r\}$

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

Access Graph :

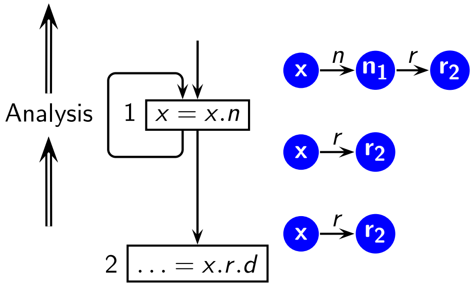


Inclusion of Program Point Facilitates Summarization



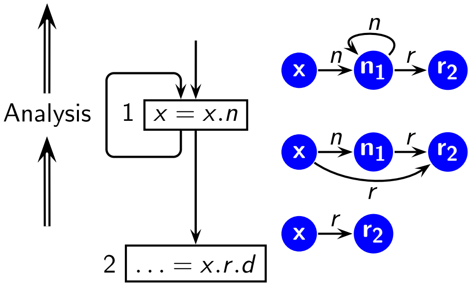
Inclusion of Program Point Facilitates Summarization

Iteration #1



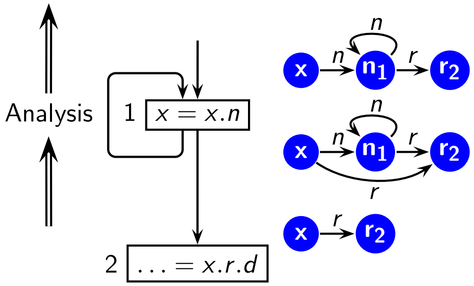
Inclusion of Program Point Facilitates Summarization

Iteration #2

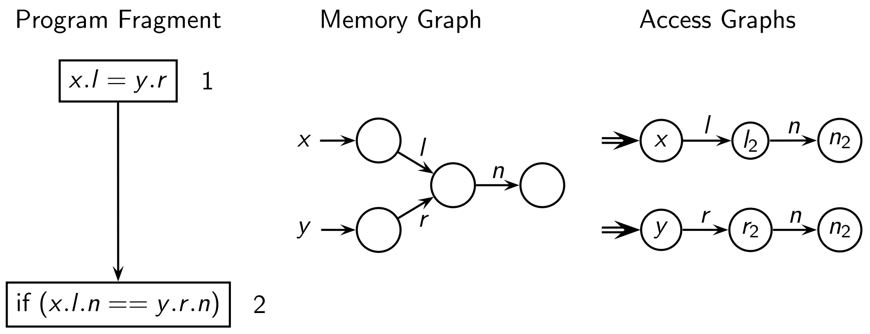


Inclusion of Program Point Facilitates Summarization

Iteration #3



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.

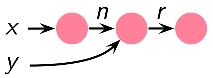
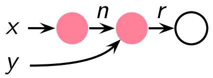
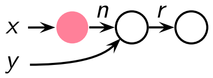
Access Graph Operations

- *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) 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	$G_2 = (G_1, M) \# \emptyset$	$P(G_2) = \emptyset$
	$G_2 = (G_1, M) \# S$	$P(G_2) \supseteq P(G_1) \cup \{\rho \rightarrow \sigma \mid \rho \in P(G_1, M),\ \sigma \in P(S)\}$

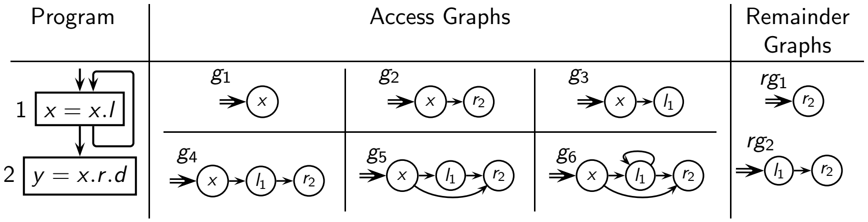
Semantics of Access Graph Operations

- $P(G)$ 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	$G_2 = (G_1, M) \# \emptyset$	$P(G_2) = \emptyset$
	$G_2 = (G_1, M) \# S$	$P(G_2) \supseteq P(G_1) \cup \{\rho \rightarrow \sigma \mid \rho \in P(G_1, M),\ \sigma \in P(S)\}$

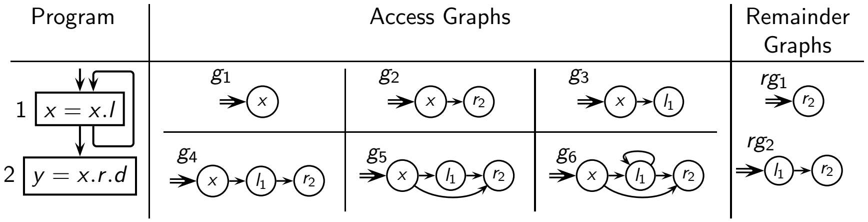
σ represents remainder

Access Graph Operations: Examples



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

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

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

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

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

Flow Functions for Explicit Liveness Analysis: 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

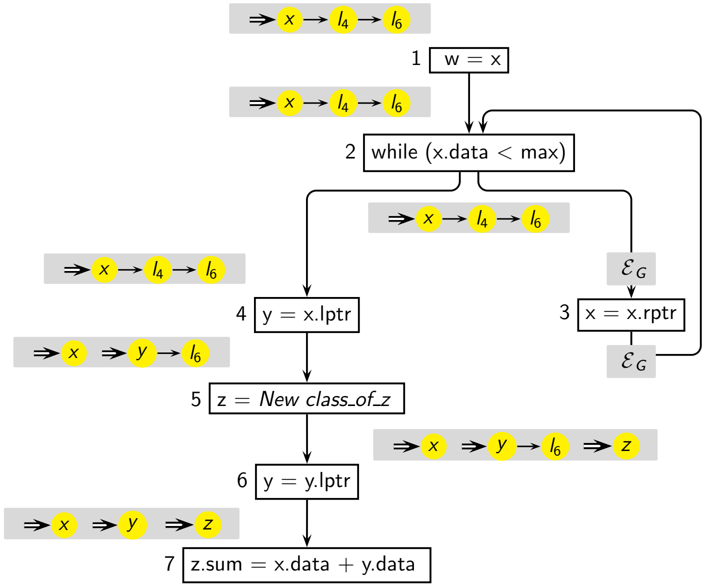
Must link aliasing for precision

Flow Functions for Explicit Liveness Analysis: Access Graphs Version

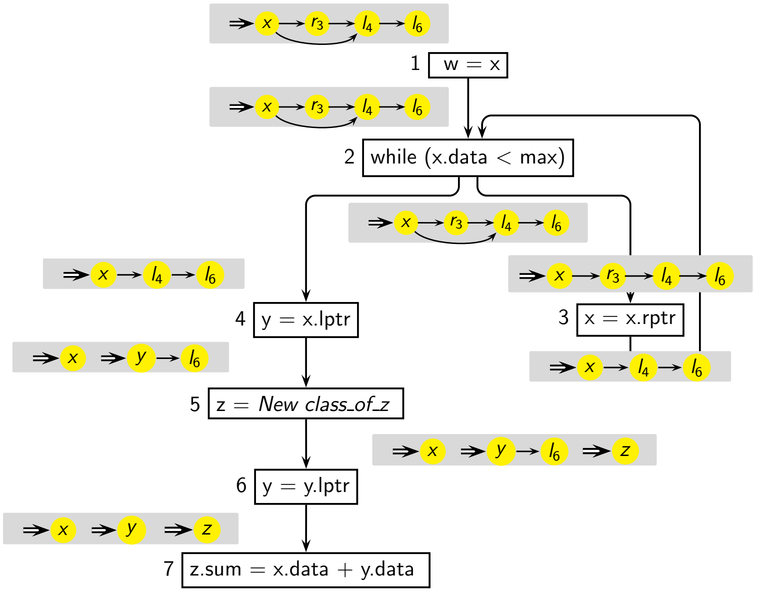
- A denotes May Aliases at the exit of node n
- $mkGraph(\rho)$ creates an access graph for access path ρ

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	$\emptyset$	$\emptyset$

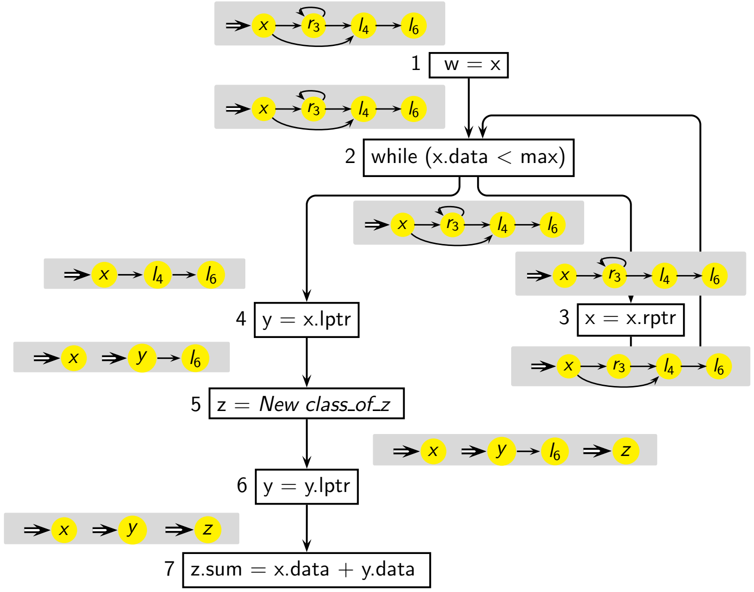
Liveness Analysis of Example Program: Ist Iteration



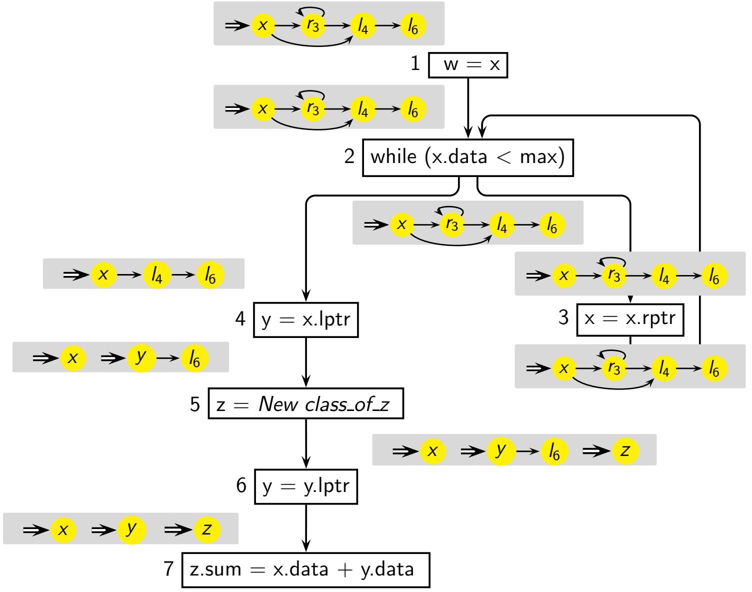
Liveness Analysis of Example Program: 2nd Iteration



Liveness Analysis of Example Program: 3rd Iteration

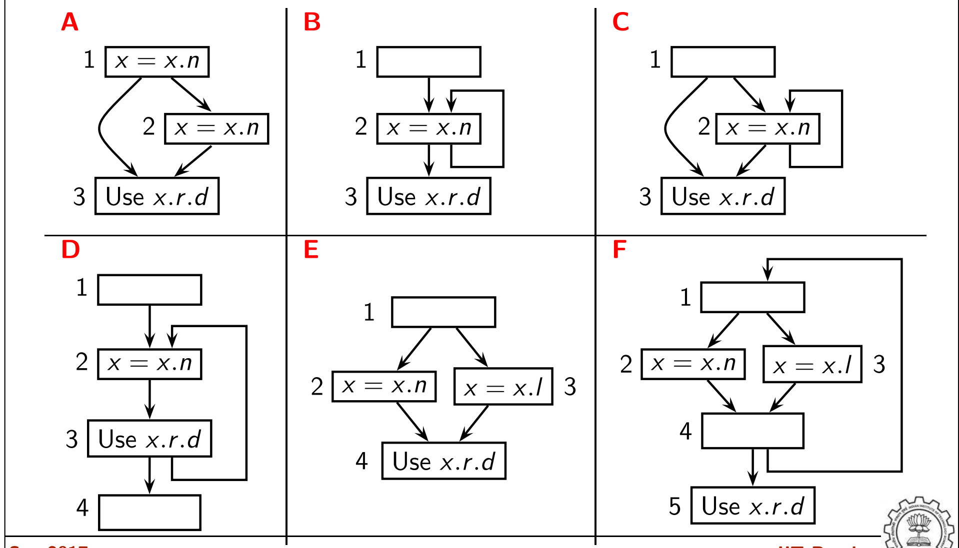


Liveness Analysis of Example Program: 4th Iteration



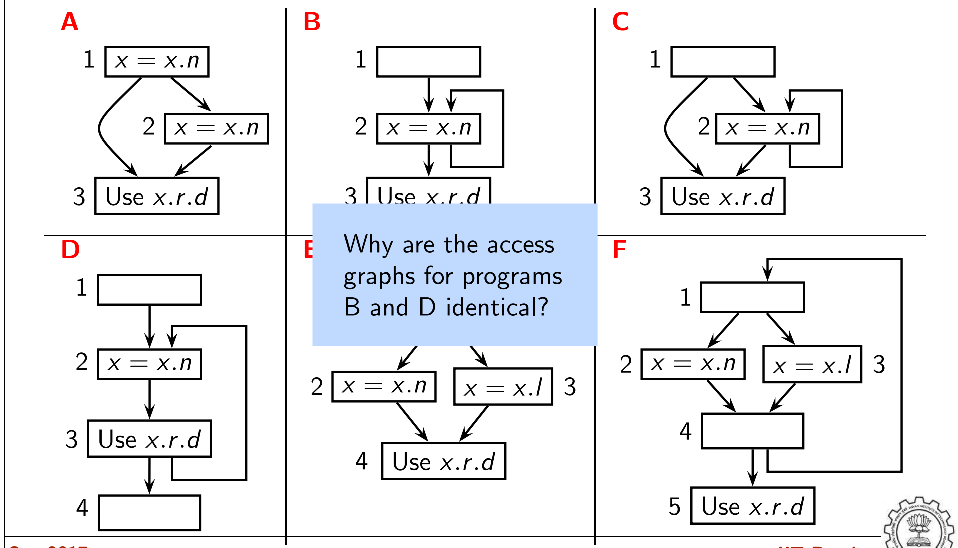
Tutorial Problem for Explicit Liveness (1)

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



Tutorial Problem for Explicit Liveness (1)

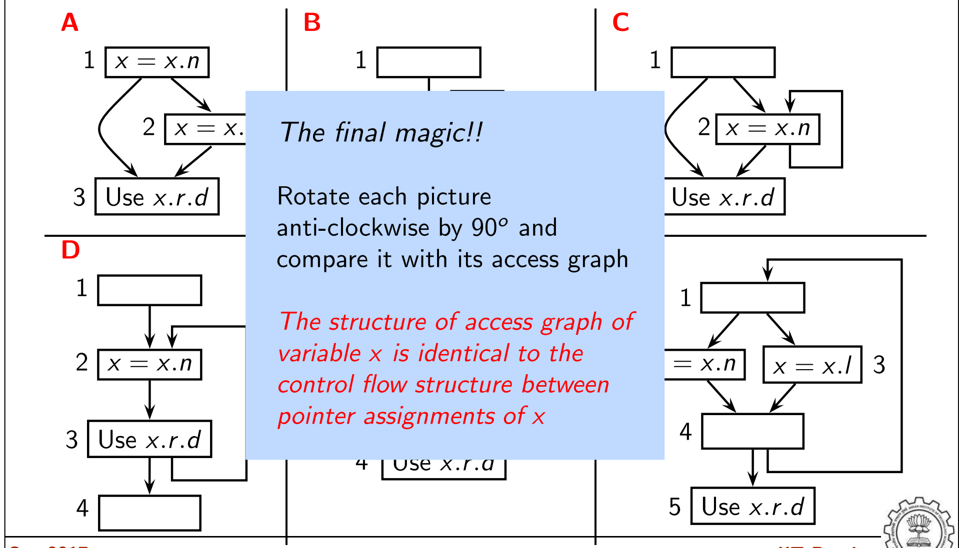
Construct access graphs at the entry of block 1 for the following programs



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



The final magic!!

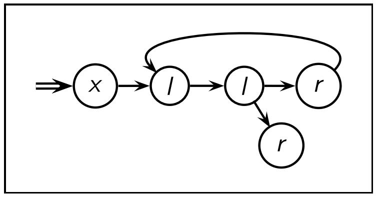
Rotate each picture anti-clockwise by 90° and compare it with its access graph

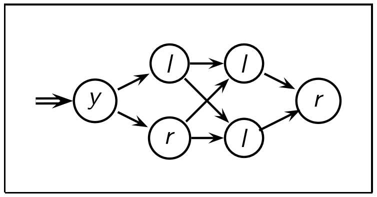
The structure of access graph of variable x is identical to the control flow structure between pointer assignments of x

---

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

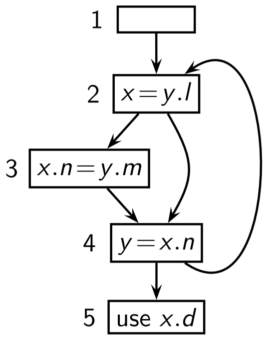




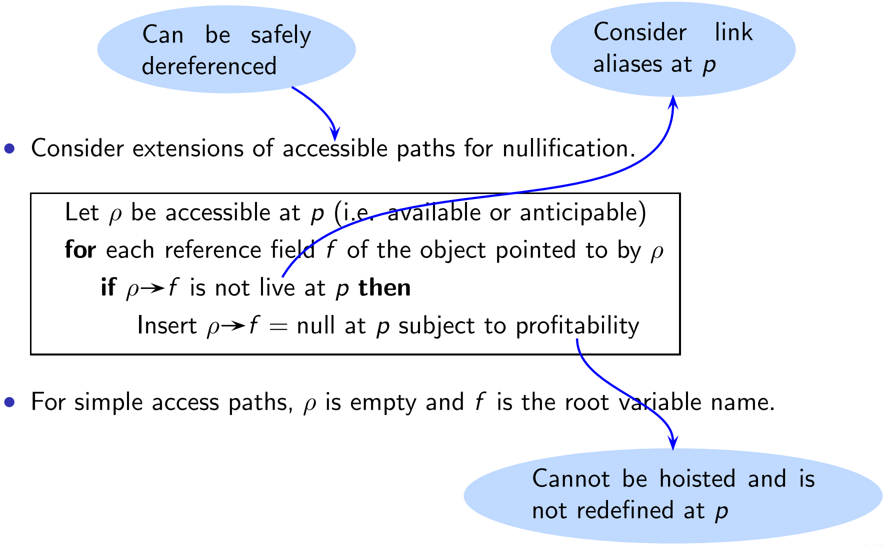
Tutorial Problem for Explicit Liveness (3)

- 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?



Can be safely dereferenced

Consider link aliases at p

- Consider extensions of accessible paths for nullification.

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

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

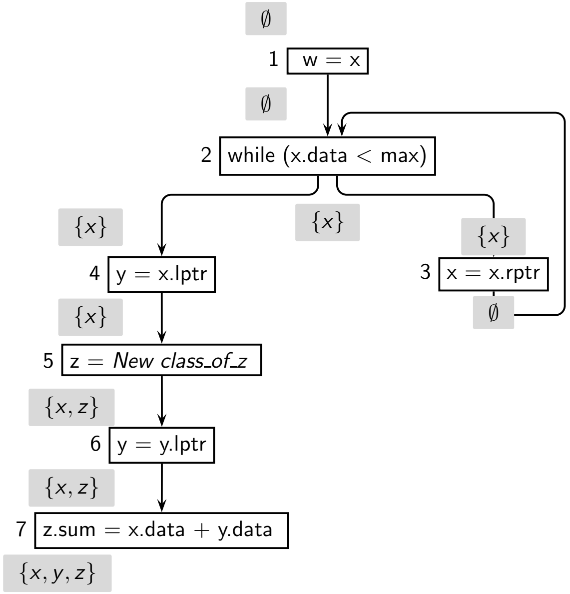
Cannot be hoisted and is not redefined at p

Availability and Anticipability Analyses

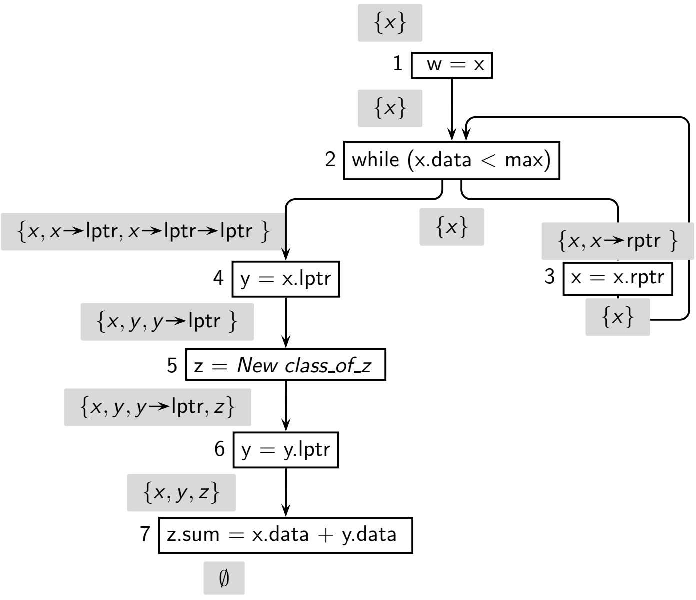
- ρ is available at program point p if the target of each prefix of ρ is guaranteed to be created along every control flow path reaching p.
- ρ is anticipable at program point p if the target of each prefix of ρ is guaranteed to be dereferenced along every control flow path starting at p.
- Finiteness.
  - An anticipable (available) access path must be anticipable (available) along every paths. Thus unbounded paths arising out of loops cannot be anticipable (available).
  - Due to "every control flow path nature", computation of anticipable and available access paths uses $\cap$ as the confluence. Thus the sets are bounded.

$\Rightarrow$ No need of access graphs.

Availability Analysis of Example Program



Anticipability Analysis of Example Program



$\{x\}$

1 w = x

$\{x\}$

2 while (x.data < max)

$\{x, x{\to}\text{lptr}, x{\to}\text{lptr}{\to}\text{lptr}\}$

$\{x\}$

$\{x, x{\to}\text{rptr}\}$

4 y = x.lptr

3 x = x.rptr

$\{x\}$

$\{x, y, y{\to}\text{lptr}\}$

5 z = *New class_of_z*

$\{x, y, y{\to}\text{lptr}, z\}$

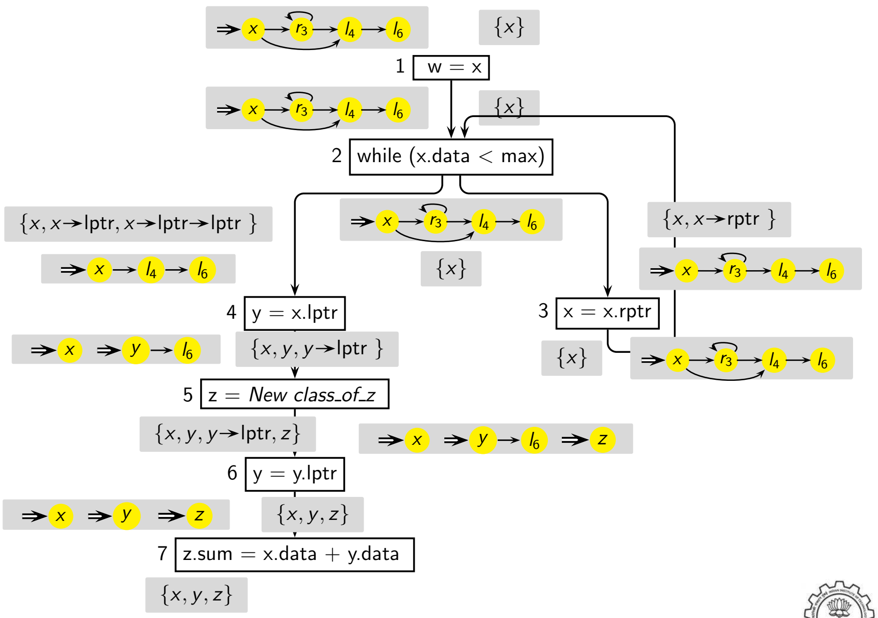
6 y = y.lptr

$\{x, y, z\}$

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

$\emptyset$

Live and Accessible Paths



$\{x\}$

1 w = x

$\{x\}$

2 while (x.data < max)

$\{x, x{\to}\text{lptr}, x{\to}\text{lptr}{\to}\text{lptr}\}$

$\{x\}$

$\{x, x{\to}\text{rptr}\}$

4 y = x.lptr

3 x = x.rptr

$\{x, y, y{\to}\text{lptr}\}$

$\{x\}$

5 z = *New class_of_z*

$\{x, y, y{\to}\text{lptr}, z\}$

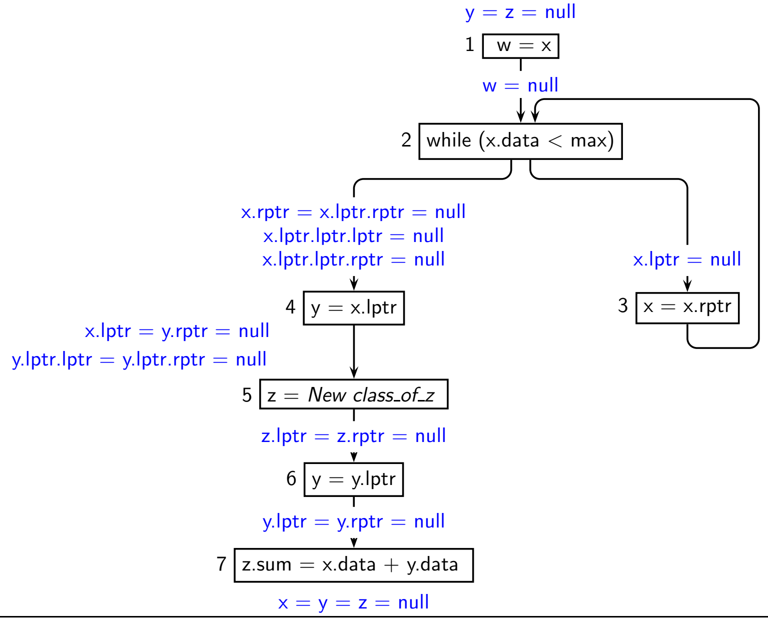
6 y = y.lptr

$\{x, y, z\}$

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

$\{x, y, z\}$

Creating null Assignments from Live and Accessible Paths



y = z = null

1 w = x

w = null

2 while (x.data < max)

x.rptr = x.lptr.rptr = null
x.lptr.lptr.lptr = null
x.lptr.lptr.rptr = null

x.lptr = null

4 y = x.lptr

3 x = x.rptr

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

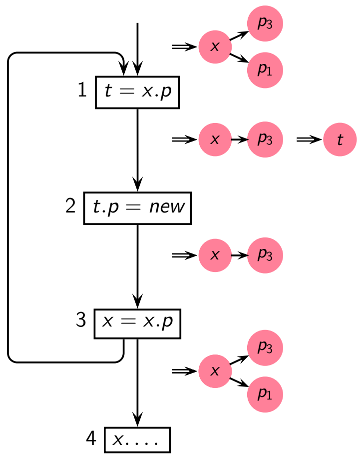
7 z.sum = x.data + y.data

x = y = z = null

The Resulting Program

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

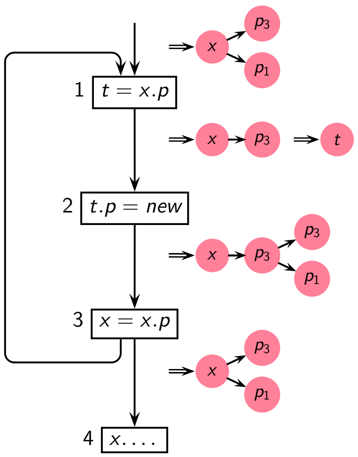
Overapproximation Caused by Our Summarization



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

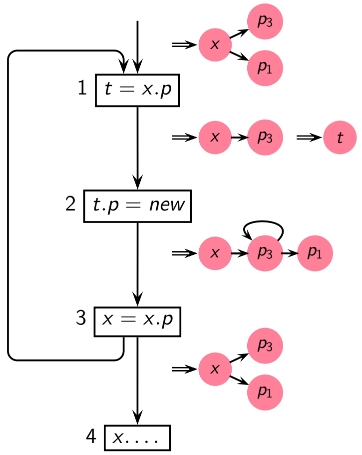
---

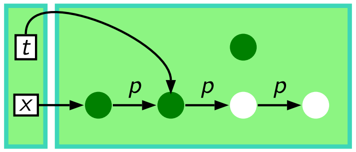
Overapproximation Caused by Our Summarization



- 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





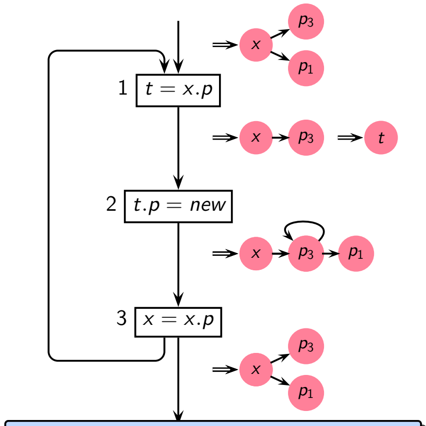
Out_1

- 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$_2$
- $x \rightarrow p \rightarrow p$ is live at Out_2

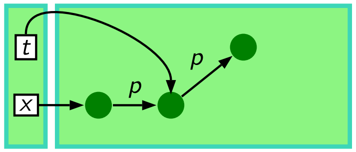
  $x \rightarrow p \rightarrow p \rightarrow p$ is dead at Out_2
- First p used in statement 3

  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!



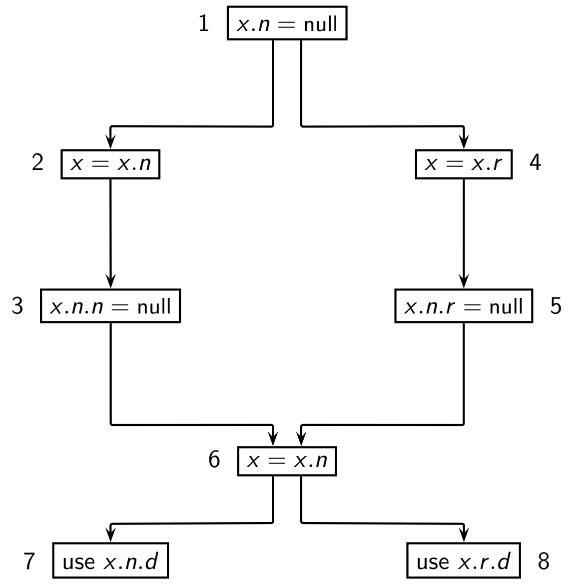
Out_2

- 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

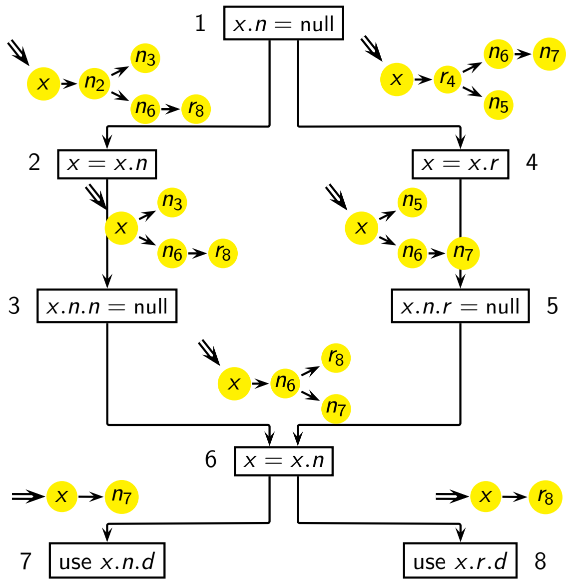
  $x \rightarrow p \rightarrow p \rightarrow p$ is dead at Out_2
- First p used in statement 3

  Second p used in statement 4
- Third p is reallocated

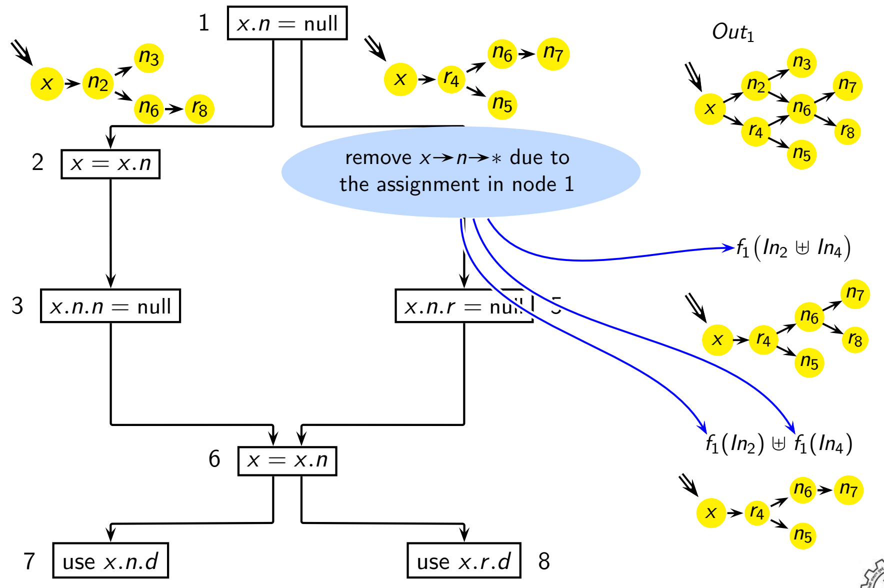
Non-Distributivity of Explicit Liveness Analysis



Non-Distributivity of Explicit Liveness Analysis

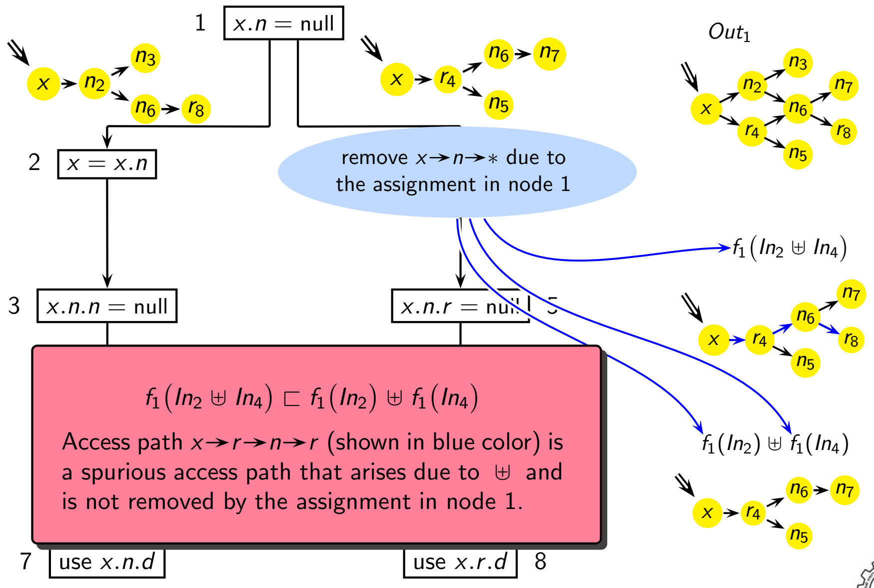


Non-Distributivity of Explicit Liveness Analysis



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

Non-Distributivity of Explicit Liveness Analysis



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

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

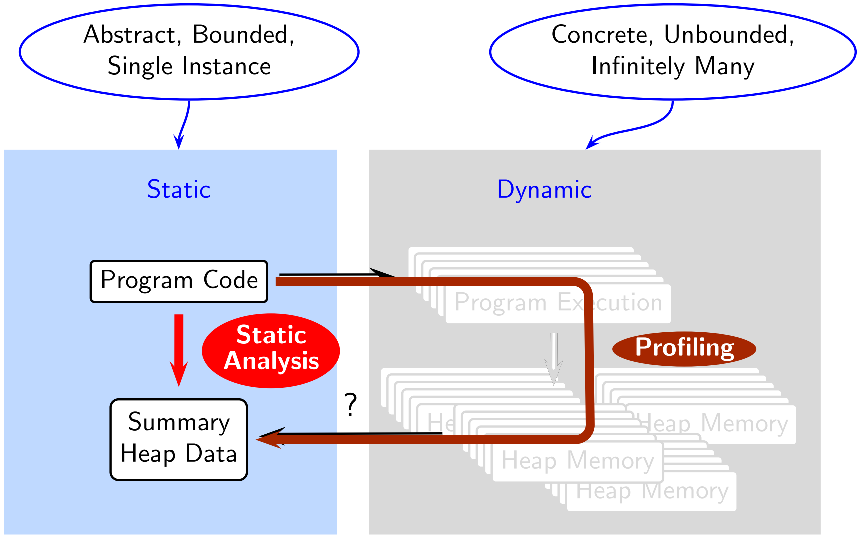
Access path $x \rightarrow r \rightarrow n \rightarrow r$ (shown in blue color) is a spurious access path that arises due to $\uplus$ and is not removed by the assignment in node 1.

Issues Not Covered

- 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?



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