Pointer Analysis

Uday Khedker

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

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

September 2018

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.

An Outline of Pointer Analysis Coverage

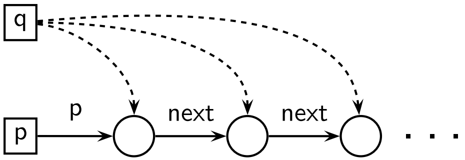
- The larger perspective
- Comparing Points-to and Alias information
- Defining Points-to Analysis
- Flow-Insensitive Points-to Analysis
- Flow-Sensitive Points-to Analysis
- Pointer Analyses: An Engineer's Landscape
- Liveness Based Points-to Analysis
- Generalizations to Heap, Arrays, Pointer Arithmetic,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

Code Optimization In Presence of Pointers (1)

Program

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

Memory graph at statement 5



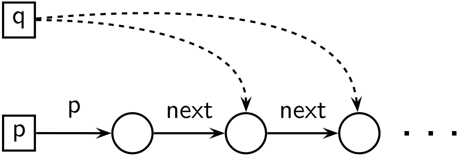
- Is p→data live at the exit of line 5? Can we delete line 5?

Code Optimization In Presence of Pointers (1)

Program

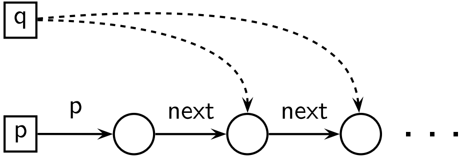
```
1.  q = p;
2.  do {
3.        q = q→next;
4.  }  while (. . . )
5.  p→data = r1;
6.  print (q→data);
7.  p→data = r2;
```

Memory graph at statement 5



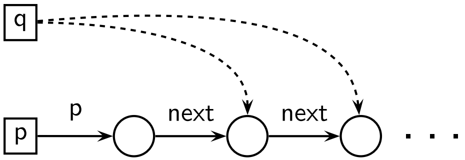
- Is p→data live at the exit of line 5? Can we delete line 5?

Code Optimization In Presence of Pointers (1)

Program	*Memory graph at statement 5*
1. q = p; 2. do { 3. q = q→next; 4. } while (. . .) 5. p→data = r1; 6. print (q→data); 7. p→data = r2;	

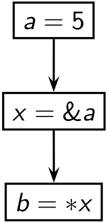
- Is p→data live at the exit of line 5? Can we delete line 5?
- We cannot delete line 5 if p and q can be possibly aliased (while loop or do-while loop with a circular list)

Code Optimization In Presence of Pointers (1)

Program	*Memory graph at statement 5*
1. q = p; 2. do { 3. q = q→next; 4. } while (. . .) 5. p→data = r1; 6. print (q→data); 7. p→data = r2;	

- 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 (2)



Original Program

Code Optimization In Presence of Pointers (2)

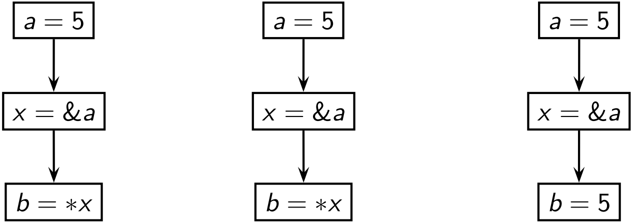
$$\boxed{a = 5} \downarrow \boxed{x = \&a} \downarrow \boxed{b = *x}$$

Original Program

$$\boxed{a = 5} \downarrow \boxed{x = \&a} \downarrow \boxed{b = *x}$$

Constant Propagation without aliasing

Code Optimization In Presence of Pointers (2)

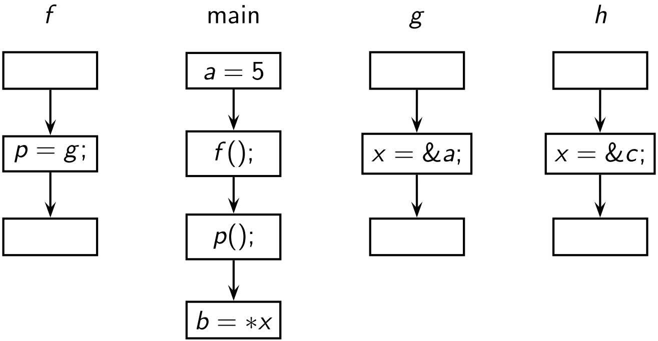


Original Program

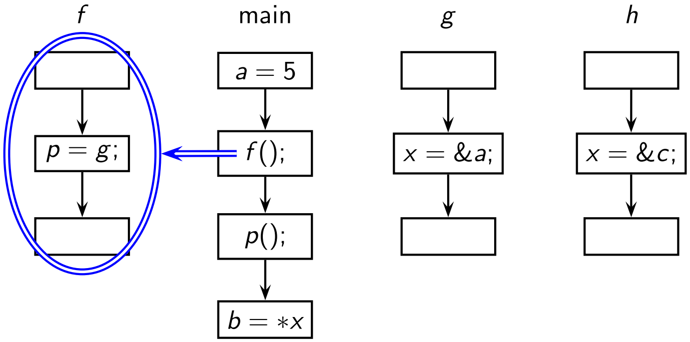
Constant Propagation without aliasing

Constant Propagation with aliasing

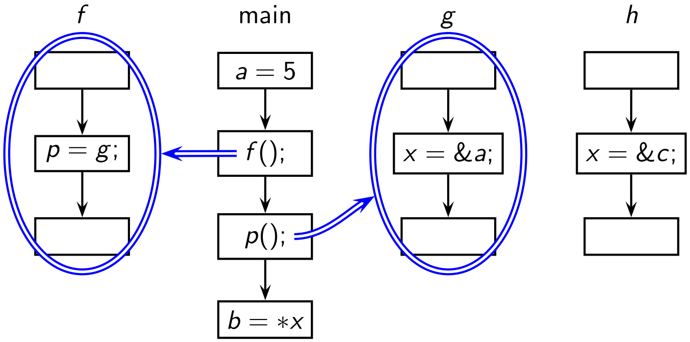
Code Optimization In Presence of Pointers (3)



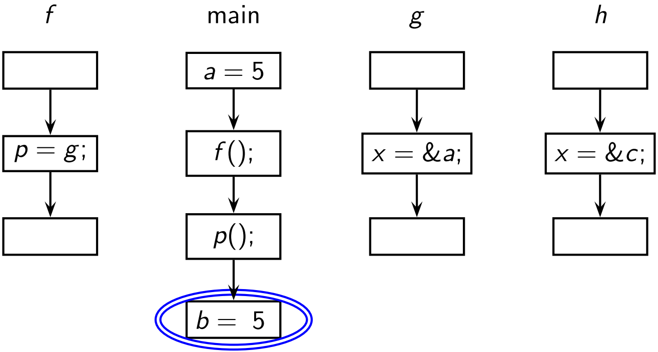
Code Optimization In Presence of Pointers (3)



Code Optimization In Presence of Pointers (3)



Code Optimization In Presence of Pointers (3)

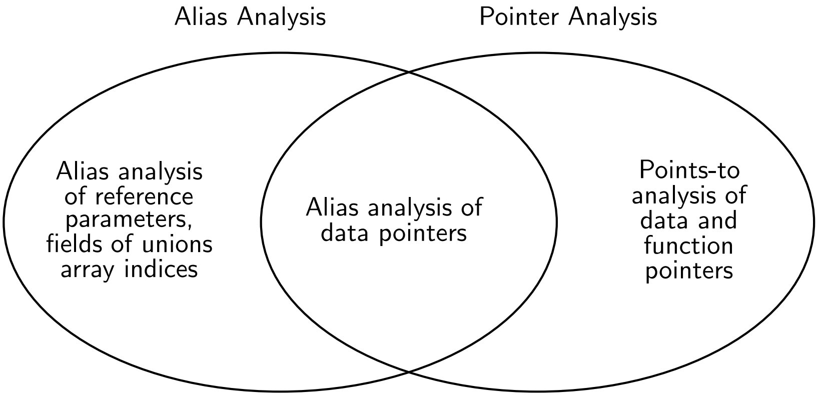


Pointer Analysis

- Answers the following questions for indirect accesses:
  - Which data is read? $x = *y$
  - Which data is written? $*x = y$
  - Which procedure is called? $p()$ or $x \rightarrow f()$
- Enables precise data flow and interprocedural control flow analysis
- Computationally intensive analyses are ineffective when supplied with imprecise points-to information,

  (e.g., model checking, interprocedural analyses)
- Needs to scale to large programs

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

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

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
  - 2018 . . ☹

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]

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?

The Engineering of Pointer Analysis

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

The Engineering of Pointer Analysis

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

- "Fortunately many approximations exist"

The Engineering of Pointer Analysis

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

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

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?

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?
- Most common trend as evidenced by publications
  - Build acceptable approximations guided by empirical observations
  - The notion of acceptability is often constrained by beliefs rather than possibilities

Abstraction Vs. Approximation in Static Analysis

- Static analysis needs to create abstract values that represent many concrete values
- Mapping concrete values to abstract values
  - *Abstraction*.
    Deciding which properties of the concrete values are essential *What*
    Ease of understanding, reasoning, modelling etc. *Why*
  - *Approximation*.
    Deciding which properties of the concrete values cannot be represented accurately and should be summarized *What*
    Decidability, tractability, or efficiency and scalability *Why*

Abstraction Vs. Approximation in Static Analysis

- Abstractions
  - focus on precision and conciseness of modelling
  - tell us what we can ignore without being imprecise
- Approximations
  - focus on efficiency and scalability
  - tell us the imprecision that we have to tolerate

Abstraction Vs. Approximation in Static Analysis

- Abstractions
  - focus on precision and conciseness of modelling
  - tell us what we can ignore without being imprecise
- Approximations
  - focus on efficiency and scalability
  - tell us the imprecision that we have to tolerate
- *Our Holy Grail:*

  Build clean abstractions before surrendering to the approximations

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?

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

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
- Defining Points-to Analysis
- 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

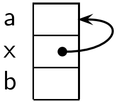
Alias Information Vs. Points-to Information

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

$\downarrow$

2 $\boxed{b = x}$

Alias Information Vs. Points-to Information

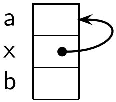


1 $\boxed{x = \&a}$

2 $\boxed{b = x}$

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

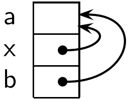
Alias Information Vs. Points-to Information



1 $x = \&a$

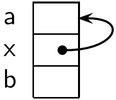
“*x Points-to a*”
denoted $x \rightarrowtail a$

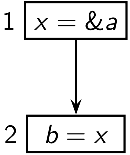
2 $b = x$



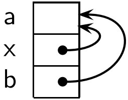
“*x* and *b* are *Aliases*”
denoted $x \stackrel{\circ}{=} b$

Alias Information Vs. Points-to Information





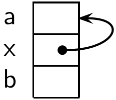
“x *Points-to* a”
denoted $x \rightarrowtail a$



“x and b are *Aliases*”
denoted $x \circeq b$

Symmetric and Reflexive

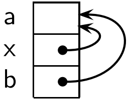
Alias Information Vs. Points-to Information



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

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

Neither Symmetric Nor Reflexive

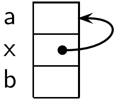


2 $\boxed{b = x}$

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

Symmetric and Reflexive

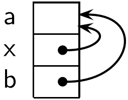
Alias Information Vs. Points-to Information



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

“*x Points-to a*”
denoted $x \rightarrowtail a$

Neither
Symmetric
Nor Reflexive



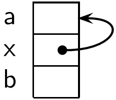
2 $\boxed{b = x}$

“*x* and *b* are *Aliases*”
denoted $x \circeq b$

Symmetric
and
Reflexive

- What about transitivity?

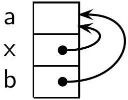
Alias Information Vs. Points-to Information



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

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

Neither
Symmetric
Nor Reflexive



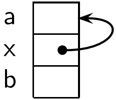
2 $\boxed{b = x}$

"x and b are *Aliases*"
denoted $x \mathrel{\mathring{=}} b$

Symmetric
and
Reflexive

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

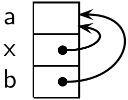
Alias Information Vs. Points-to Information



1 $x = \&a$

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

Neither Symmetric Nor Reflexive



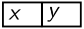
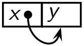
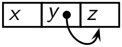
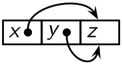
2 $b = x$

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

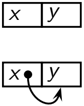
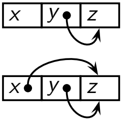
Symmetric and Reflexive

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

Comparing Points-to and Alias Relations (1)

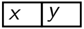
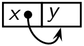
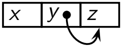
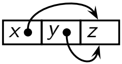
Statement	Memory		Points-to		Aliases	
$x = \&y$	Before (assume)		Existing		Existing	
	After		New	$x \rightarrowtail y$	New Direct	$x \circeq \&y$
$x = y$	Before (assume)		Existing	$y \rightarrowtail z$	Existing	$y \circeq \&z$
	After		New	$x \rightarrowtail z$	New Direct	$x \circeq y$
					New Indirect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

Comparing Points-to and Alias Relations (1)

Statement	Memory	Points-to	Aliases
$x = \&y$	Before (assume) After 	Existing: New: $x \rightarrowtail y$	Existing: New Direct: $x \stackrel{\circ}{=} \&y$
$x = y$	Before (assume) After 	Existing: $y \rightarrowtail z$ New: $x \rightarrowtail z$	Existing: $y \stackrel{\circ}{=} \&z$ New Direct: $x \stackrel{\circ}{=} y$ New Indirect: $x \stackrel{\circ}{=} \&z$

- Indirect aliases. Substitute a name by its aliases for transitivity

Comparing Points-to and Alias Relations (1)

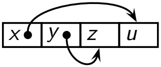
Statement	Memory	Points-to	Aliases
$x = \&y$	Before (assume)  After 	Existing: New: $x \rightarrowtail y$	Existing: New Direct: $x \circeq \&y$
$x = y$	Before (assume)  After 	Existing: $y \rightarrowtail z$ New: $x \rightarrowtail z$	Existing: $y \circeq \&z$ New Direct: $x \circeq y$ New Indirect: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

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

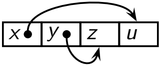
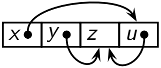
Comparing Points-to and Alias Relations (2)

Statement	Memory	Points-to	Aliases
$*x = y$			
$x = *y$			

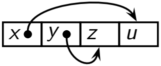
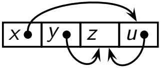
Comparing Points-to and Alias Relations (2)

Statement	Memory	Points-to	Aliases
$*x = y$	Before (assume) 	Existing: $x \rightarrowtail u$, $y \rightarrowtail z$	Existing: $x \circeq \&u$, $y \circeq \&z$
$x = *y$			

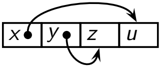
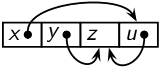
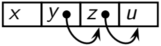
Comparing Points-to and Alias Relations (2)

Statement	Memory	Points-to	Aliases
$*x = y$	Before (assume)  After 	Existing: $x \rightarrowtail u$, $y \rightarrowtail z$ New: $u \rightarrowtail z$	Existing: $x \overset{\circ}{=} \&u$, $y \overset{\circ}{=} \&z$ New Direct: $*x \overset{\circ}{=} y$
$x = *y$			

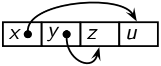
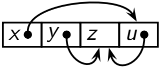
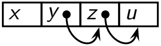
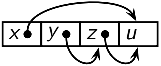
Comparing Points-to and Alias Relations (2)

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

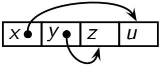
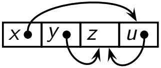
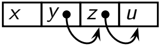
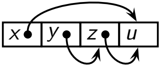
Comparing Points-to and Alias Relations (2)

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

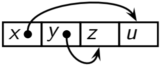
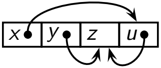
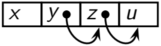
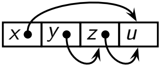
Comparing Points-to and Alias Relations (2)

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

Comparing Points-to and Alias Relations (2)

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

Comparing Points-to and Alias Relations (2)

Statement	Memory	Points-to		Aliases	
$*x = y$	Before (assume) 	Existing	$x \rightarrowtail u$ $y \rightarrowtail z$	Existing	$x \stackrel{\circ}{=} \&u$ $y \stackrel{\circ}{=} \&z$
				New Direct	$*x \stackrel{\circ}{=} y$
	After 	New	$u \rightarrowtail z$	New Indirect	$u \stackrel{\circ}{=} \&z$ $y \stackrel{\circ}{=} u$ $*x \stackrel{\circ}{=} \&z$
$x = *y$	Before (assume) 	Existing	$y \rightarrowtail z$ $z \rightarrowtail u$	Existing	$y \stackrel{\circ}{=} \&z$ $z \stackrel{\circ}{=} \&u$ $*y \stackrel{\circ}{=} \&u$
				New Direct	$x \stackrel{\circ}{=} *y$
	After 	New	$x \rightarrowtail u$	New Indirect	$x \stackrel{\circ}{=} \&u$ $x \stackrel{\circ}{=} z$

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

- Alias information records paths in the memory graph

Comparing Points-to and Alias Relations (3)

- Points-to information records edges in the memory graph
  - aliases of the kind $x \stackrel{\circ}{=} \&y$
    x holds the address of y

- 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)

Comparing Points-to and Alias Relations (3)

- Points-to information records edges in the memory graph
  - aliases of the kind $x \stackrel{\circ}{=} \&y$
    x holds the address of y
  - other aliases can be discovered by composing edges

- 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)

Comparing Points-to and Alias Relations (3)

- Points-to information records edges in the memory graph
  - aliases of the kind $x \stackrel{\circ}{=} \&y$
    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
- 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

Comparing Points-to and Alias Relations (3)

- Points-to information records edges in the memory graph
  - aliases of the kind $x \circeq \&y$
    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

- 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 $x \circeq y$ then $*x \circeq *y$ is redundant and is not recorded

Comparing Points-to and Alias Relations (3)

- Points-to information records edges in the memory graph
  - aliases of the kind $x \circeq \&y$
    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 $x \circeq y$ then $*x \circeq *y$ 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
- Defining Points-to Analysis 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

Pointer Statements

Pointer assignments		Use pointers in expressions
Addr	$x = \&y$	*Use* x
Copy	$x = y$	
Load	$x = *y$	
	$x = y \rightarrow n$	
Store	$*x = y$	
	$x \rightarrow n = y$	

- Field accesses such as $x.n$ are treated as new compile time names
- Containment of $x.n$ within x is recorded in terms of offsets
- Heap will be introduced later

What Does a Use Statement Represent? (1)

Consider the declaration: `int a, *x, **y;`

Source	3 Address representation	Our modelling
$*x = a$	$*x = a$	Use x
$a = *x$	$a = *x$	Use x
if $(x == NULL)$	if $(x == NULL)$	Use x
if $(*x == 5)$	if $(*x == 5)$	Use x
if $(*y == NULL)$	$t = *y$ if $(t == NULL)$	$t = *y$ Use x
$(** y = a)$	$t = *y$ $*t = a$	$t = *y$ Use t

We retain only the pointers

What Does a Use Statement Represent? (2)

Consider the declaration:

```
struct s {
   struct s *n;
   int m;
} a, b, *x;
```

Source	3 Address representation	Our modelling
$a.n = \&b$	$a.n = \&b$	$a.n = \&b$
if $(x \rightarrow n == NULL)$	$t = x \rightarrow n$ if $(t == NULL)$	$t = x \rightarrow n$ Use t
if $(a.n == NULL)$	$t = a.n$ if $(t == NULL)$	$t = a.n$ Use t

We retain only the pointers

Concrete States and Traces for Points-to Analysis

Notation	Illustration
V contains all variables	
$P \subseteq V$ contains all pointer variables	
F contains all pointer fields in structures (and also "$*$")	
Data states $\delta : V \times F \to V$	
Traces τ are sequences of transitions $(n, \delta) \to (n', \delta')$ starting from a given initial n_0, δ_0	

Concrete States and Traces for Points-to Analysis

Notation	Illustration
V contains all variables	$V = \{a, b, x\}$
$P \subseteq V$ contains all pointer variables	$P = \{ \text{x} \}$
F contains all pointer fields in structures (and also "$*$")	$F = \{ \text{*}, \text{f} \}$
Data states $\delta : V \times F \to V$	
Traces τ are sequences of transitions $(n, \delta) \to (n', \delta')$ starting from a given initial n_0, δ_0	

Program statement	Corresponding state after each statement
$0: \quad$ skip	$\delta_0 = \emptyset$
$1: \quad x = \&a$	$\delta_1 = \{((x, *) \mapsto a)\}$
$2: \quad x \to f = \&b$	$\delta_2 = \{((x, *) \mapsto a), ((a, f) \mapsto b)\}$

Trace $\tau_1 = (0, \delta_0) \to (1, \delta_1) \to (2, \delta_2)$

Ideal Points-to Analysis

For a given statement n

- Reachable States are the states reaching the statement along all traces
- Ideal May-Points-to analysis computes Points-to information reaching along all traces
- Ideal Must-Points-to analysis computes Points-to information that is common to all traces

$$RS(n) = \left\{\delta \mid (n, \delta) \text{ occurs in some trace } \tau\right\}$$

$$\text{IdealMayPT}(n) = \bigcup_{\delta \in RS(n)} \delta$$

$$\text{IdealMustPT}(n) = \bigcap_{\delta \in RS(n)} \delta$$

Soundness and Precision of Flow-Sensitive May-Points-to Analysis

A flow-sensitive points-to analysis algorithm A computes $S : N \rightarrow V \times F \times V$

- A flow-sensitive points-to analysis algorithm A is sound if

  $$\forall n : \; S(n) \supseteq \text{IdealMayPT}(n)$$

- A flow-sensitive points-to analysis algorithm A is precise if

  $$\forall n : \; S(n) = \text{IdealMayPT}(n)$$

- A flow-sensitive points-to analysis algorithm A_1 is more precise than A_2 if

  $$\forall n : \; S_2(n) \supset S_1(n) \supseteq \text{IdealMayPT}(n)$$

  (Precision is meaningful only for a sound analysis)

Soundness and Precision of Flow-Insensitive May-Points-to Analysis

A flow-insensitive points-to analysis algorithm A computes $S : V \times F \times V$

- A flow-insensitive points-to analysis algorithm A is sound if

$$S \supseteq \bigcup_{n \in N} \text{IdealMayPT}(n)$$

- A flow-insensitive points-to analysis algorithm A is precise if

$$S = \bigcup_{n \in N} \text{IdealMayPT}(n)$$

- A flow-insensitive points-to analysis algorithm A_1 is more precise than A_2 if

$$S_2 \supset S_1 \supseteq \bigcup_{n \in N} \text{IdealMayPT}(n)$$

(Precision is meaningful only for a sound analysis)

An Outline of Pointer Analysis Coverage

- The larger perspective
- Comparing Points-to and Alias information
- Defining Points-to Analysis
- 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

Defining Points-to Analysis: A Recap

δ: a state — RS: reachable states
FS: flow-sensitive — FI: flow-insensitive
$n \in N(p)$: nodes of procedure p

$$RS(n,p) = \left\{\delta \mid (n,\delta) \text{ occurs in some trace } \tau(p) \text{ of procedure } p\right\}$$

$$\text{FSMayPT}(n,p) \supseteq \text{IdealMayPT}(n,p) = \bigcup_{\delta \in RS(n,p)} \delta$$

$$\text{FSMustPT}(n,p) \subseteq \text{IdealMustPT}(n,p) = \bigcap_{\delta \in RS(n,p)} \delta$$

$$\text{FIPT}(p) \supseteq \bigcup_{n \in N(p)} \text{FSMayPT}(n,p) \supseteq \bigcup_{n \in N(p)} \text{IdealMayPT}(n,p) \supseteq \bigcup_{\substack{\delta \in RS(n,p) \\ n \in N(p)}} \delta$$

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.

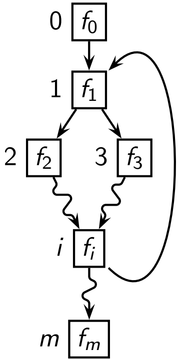
  The summary information is required to be a safe approximation of point-specific information for each point.
- $\text{Kill}_n(X)$ component is ignored.

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

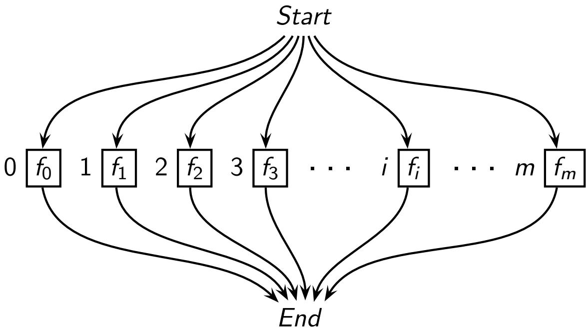
The control flow graph is a complete graph (except for the Start and End nodes)

Flow Insensitivity in Data Flow Analysis

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



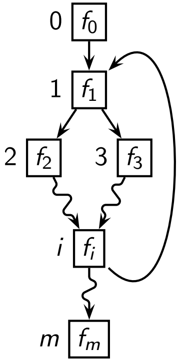
Control flow graph



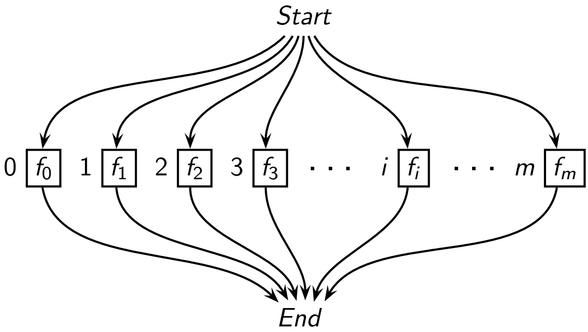
Flow-insensitive analysis

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

Examples of Flow-Insensitive Analyses

- Type checking/inferencing
  (What about interpreted languages?)

Examples of Flow-Insensitive Analyses

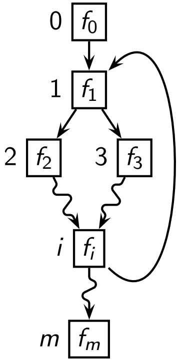
- Type checking/inferencing
  (What about interpreted languages?)
- Address taken analysis
  Which variables have their addresses taken?

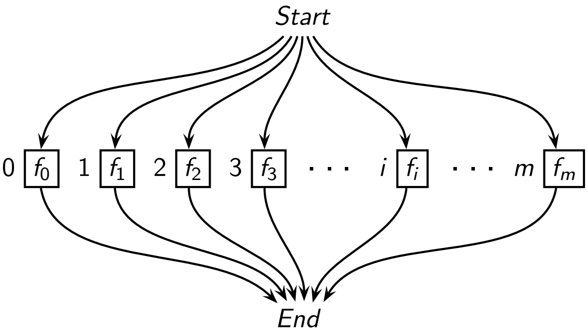
Examples of Flow-Insensitive Analyses

- 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

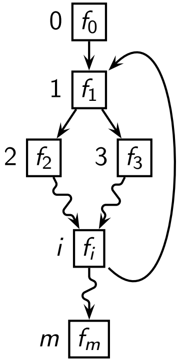
Assuming $\text{Gen}_n(X)$ has dependent parts and $\text{Kill}_n(X)$ is ignored

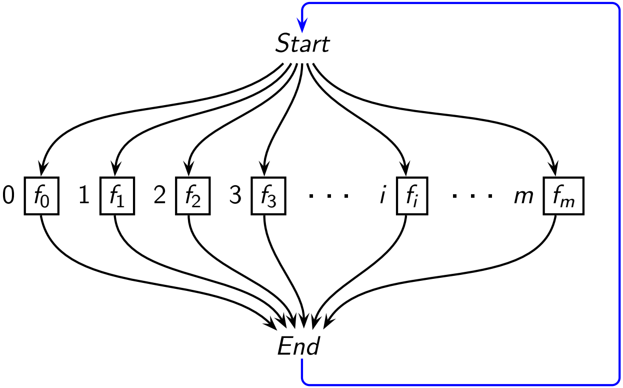




Flow Insensitivity in Data Flow Analysis

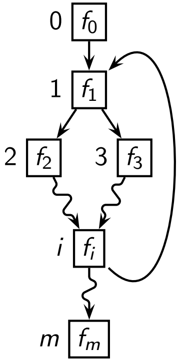
Assuming $\text{Gen}_n(X)$ has dependent parts and $\text{Kill}_n(X)$ is ignored

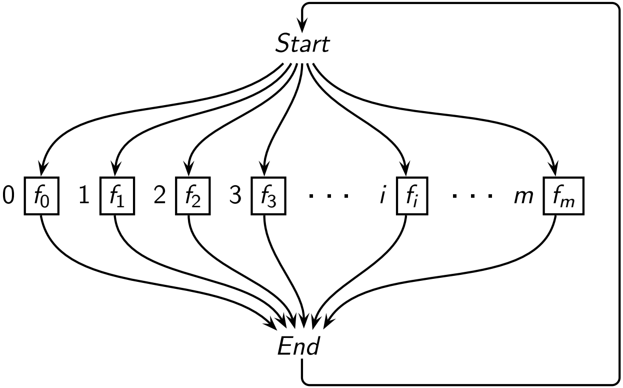




Flow Insensitivity in Data Flow Analysis

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

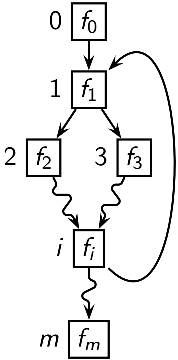


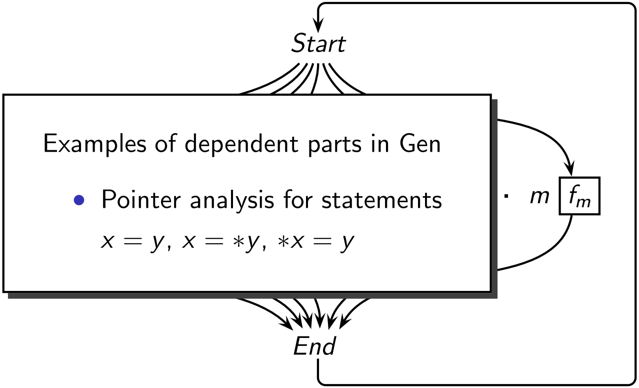


Allows arbitrary compositions of flow functions in any order
$\Rightarrow$ *Flow insensitivity*

Flow Insensitivity in Data Flow Analysis

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





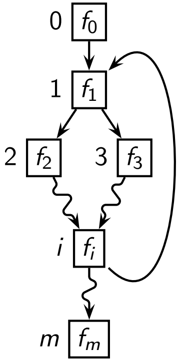
Examples of dependent parts in Gen

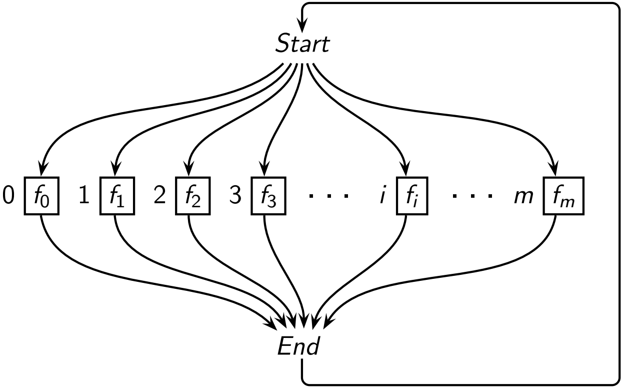
- Pointer analysis for statements

  $x = y$, $x = *y$, $*x = y$

Flow Insensitivity in Data Flow Analysis

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





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

- $P_{x.f}$ denotes the set of pointees of pointer variable x along field f
    - $P_{x.*}$ (concisely written as P_x) denotes the set of pointees of x
    - If x is a structure, P_x is the set of pointees of all fields of 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
- We use $x.f$ if the pointees of field f of x are to be 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 $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$UnifyPTS(x,y)$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ UnifyPTS(x,z)$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ UnifyPTS(y,z)$

For field f of x, we replace x by $x.f$

Andersen's and Steensgaard's Points-to Analysis

Statement	Andersen's Points-to Sets	Steensgaard's Points-to Sets
$\boxed{x = \&y}$	$P_x \supseteq \{y\}$	$P_x \supseteq \{y\}$ $Unify(y, z)$ 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

Steensgaard's view

Andersen's and Steensgaard's Points-to Analysis

Statement	Andersen's Points-to Sets	Steensgaard's Points-to Sets
$\boxed{x = \&y}$	$\boxed{P_x \supseteq \{y\}}$	$P_x \supseteq \{y\}$ $Unify(y, z)$ 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

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
$x = \&y$	$P_x \supseteq \{y\}$	$P_x \supseteq \{y\}$ $Unify(y, z)$ for some $z \in P_x$
$\boxed{x = y}$	$P_x \supseteq P_y$	$UnifyPTS(x, y)$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ UnifyPTS(x, z)$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ UnifyPTS(y, z)$

Andersen's view

Steensgaard's view

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 $z \in P_x$
$\boxed{x = y}$	$\boxed{P_x \supseteq P_y}$	$UnifyPTS(x, y)$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ UnifyPTS(x, z)$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ UnifyPTS(y, z)$

Andersen's view

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

Steensgaard's view

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 $z \in P_x$
$\boxed{x = y}$	$\boxed{P_x \supseteq P_y}$	$\boxed{UnifyPTS(x, y)}$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ UnifyPTS(x, z)$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ UnifyPTS(y, z)$

Andersen's view

- x points to 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
$x = \&y$	$P_x \supseteq \{y\}$	$P_x \supseteq \{y\}$ $Unify(y, z)$ for some $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$UnifyPTS(x, y)$
$\boxed{x = *y}$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ UnifyPTS(x, z)$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ UnifyPTS(y, z)$

Andersen's view

Steensgaard's view

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 $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$UnifyPTS(x,y)$
$\boxed{x = *y}$	$\boxed{P_x \supseteq P_z,\ \forall z \in P_y}$	$\forall z \in P_y,\ UnifyPTS(x,z)$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ UnifyPTS(y,z)$

Andersen's view

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

Steensgaard's view

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 $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$UnifyPTS(x, y)$
$\boxed{x = *y}$	$\boxed{P_x \supseteq P_z,\ \forall z \in P_y}$	$\boxed{\forall z \in P_y,\ UnifyPTS(x, z)}$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ UnifyPTS(y, z)$

Andersen's view

- x points to 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
$x = \&y$	$P_x \supseteq \{y\}$	$P_x \supseteq \{y\}$ $Unify(y,z)$ for some $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$UnifyPTS(x,y)$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ UnifyPTS(x,z)$
$\boxed{*x = y}$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ UnifyPTS(y,z)$

Andersen's view

Steensgaard's view

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 $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$UnifyPTS(x, y)$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ UnifyPTS(x, z)$
$\boxed{*x = y}$	$\boxed{P_z \supseteq P_y,\ \forall z \in P_x}$	$\forall z \in P_x,\ UnifyPTS(y, z)$

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

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 $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$UnifyPTS(x, y)$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ UnifyPTS(x, z)$
$\boxed{*x = y}$	$\boxed{P_z \supseteq P_y,\ \forall z \in P_x}$	$\boxed{\forall z \in P_x,\ UnifyPTS(y, z)}$

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
$x = \&y$	$P_x \supseteq \{y\}$	$P_x \supseteq \{y\}$ $Unify(y, z)$ 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 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 $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$UnifyPTS(x, y)$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ UnifyPTS(x, z)$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ UnifyPTS(y, z)$

Inclusion

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 $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$UnifyPTS(x, y)$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ UnifyPTS(x, z)$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ UnifyPTS(y, z)$

Inclusion

Equality

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 $z \in P_x$
$x = y$	$P_x \supseteq P_y$	$UnifyPTS(x, y)$
$x = *y$	$P_x \supseteq P_z,\ \forall z \in P_y$	$\forall z \in P_y,\ UnifyPTS(x, z)$
$*x = y$	$P_z \supseteq P_y,\ \forall z \in P_x$	$\forall z \in P_x,\ UnifyPTS(y, z)$

- Collect the constraints
- Solve the constraints

  Compute the least fixed point

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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

Type declarations

```
struct s {
    struct s *n;
    int m;
}   *x, *y, a, b, c, d;
```

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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$

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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

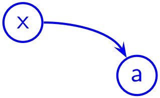
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$

Points-to Graph



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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

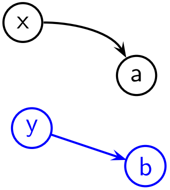
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$

Points-to Graph



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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

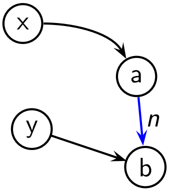
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$

Points-to Graph



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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

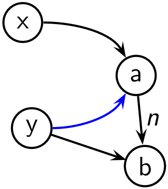
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

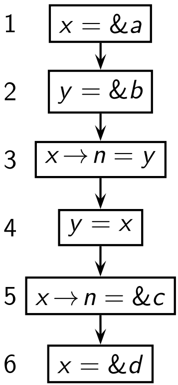
Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$

Points-to Graph



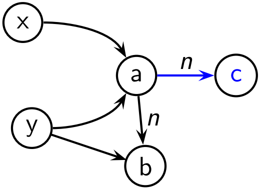
Example of Inclusion Based (aka Andersen's) Points-to Analysis

Program



Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$

Points-to Graph



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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

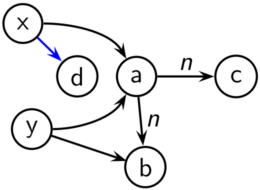
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$

Points-to Graph



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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

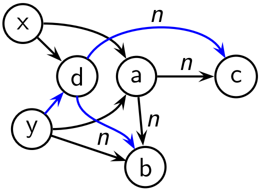
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$

Points-to Graph



- Since P_x has changed, constraints 3, 4, and 5 needs to be processed again
- Order of processing the sets influences the efficiency of this fixed point computation significantly
- A plethora of heuristics have been proposed

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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

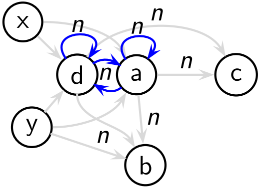
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$

Points-to Graph



- Since P_x has changed, constraints 3, 4, and 5 needs to be processed again
- Order of processing the sets influences the efficiency of this fixed point computation significantly
- A plethora of heuristics have been proposed

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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

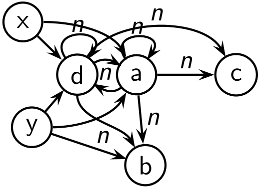
4 $y = x$

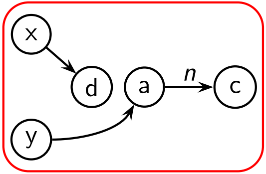
5 $x \rightarrow n = \&c$

6 $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$

Points-to Graph





- Actual graph after statement 6 (red box on the right) is much simpler with many edges killed
- y does not point to d any time in the execution

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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$ $\forall z \in P_x, Unify(a, z)$
2	$P_y \supseteq \{b\}$ $\forall z \in P_y, Unify(b, z)$
3	$\forall z \in P_x, UnifyPTS(y, z.n)$
4	$UnifyPTS(x, y)$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$ $\forall w \in P_{z.n}, Unify(w, c)$
6	$P_x \supseteq \{d\}$ $\forall z \in P_x, Unify(d, z)$

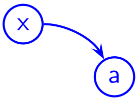
Example of Equality Based (aka Steensgaard's) Points-to Analysis

Program

1. $x = \&a$
2. $y = \&b$
3. $x \rightarrow n = y$
4. $y = x$
5. $x \rightarrow n = \&c$
6. $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$ $\forall z \in P_x, Unify(a, z)$
2	$P_y \supseteq \{b\}$ $\forall z \in P_y, Unify(b, z)$
3	$\forall z \in P_x, UnifyPTS(y, z.n)$
4	$UnifyPTS(x, y)$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$ $\forall w \in P_{z.n}, Unify(w, c)$
6	$P_x \supseteq \{d\}$ $\forall z \in P_x, Unify(d, z)$

Points-to Graph



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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

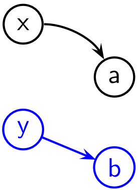
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$ $\forall z \in P_x, Unify(a, z)$
2	$P_y \supseteq \{b\}$ $\forall z \in P_y, Unify(b, z)$
3	$\forall z \in P_x, UnifyPTS(y, z.n)$
4	$UnifyPTS(x, y)$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$ $\forall w \in P_{z.n}, Unify(w, c)$
6	$P_x \supseteq \{d\}$ $\forall z \in P_x, Unify(d, z)$

Points-to Graph



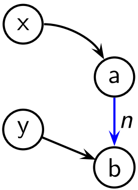
Example of Equality Based (aka Steensgaard's) Points-to Analysis

Program

1. $x = \&a$
2. $y = \&b$
3. $x \rightarrow n = y$
4. $y = x$
5. $x \rightarrow n = \&c$
6. $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$ $\forall z \in P_x, Unify(a, z)$
2	$P_y \supseteq \{b\}$ $\forall z \in P_y, Unify(b, z)$
3	$\forall z \in P_x, UnifyPTS(y, z.n)$
4	$UnifyPTS(x, y)$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$ $\forall w \in P_{z.n}, Unify(w, c)$
6	$P_x \supseteq \{d\}$ $\forall z \in P_x, Unify(d, z)$

Points-to Graph



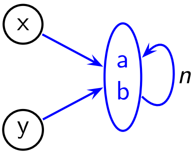
Example of Equality Based (aka Steensgaard's) Points-to Analysis

Program

1. $x = \&a$
2. $y = \&b$
3. $x \rightarrow n = y$
4. $y = x$
5. $x \rightarrow n = \&c$
6. $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$ $\forall z \in P_x, Unify(a, z)$
2	$P_y \supseteq \{b\}$ $\forall z \in P_y, Unify(b, z)$
3	$\forall z \in P_x, UnifyPTS(y, z.n)$
4	$UnifyPTS(x, y)$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$ $\forall w \in P_{z.n}, Unify(w, c)$
6	$P_x \supseteq \{d\}$ $\forall z \in P_x, Unify(d, z)$

Points-to Graph



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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

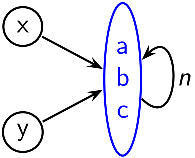
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$ $\forall z \in P_x, Unify(a, z)$
2	$P_y \supseteq \{b\}$ $\forall z \in P_y, Unify(b, z)$
3	$\forall z \in P_x, UnifyPTS(y, z.n)$
4	$UnifyPTS(x, y)$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$ $\forall w \in P_{z.n}, Unify(w, c)$
6	$P_x \supseteq \{d\}$ $\forall z \in P_x, Unify(d, z)$

Points-to Graph



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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

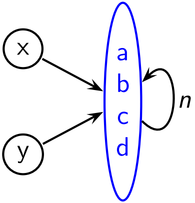
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$ $\forall z \in P_x, Unify(a, z)$
2	$P_y \supseteq \{b\}$ $\forall z \in P_y, Unify(b, z)$
3	$\forall z \in P_x, UnifyPTS(y, z.n)$
4	$UnifyPTS(x, y)$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$ $\forall w \in P_{z.n}, Unify(w, c)$
6	$P_x \supseteq \{d\}$ $\forall z \in P_x, Unify(d, z)$

Points-to Graph



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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

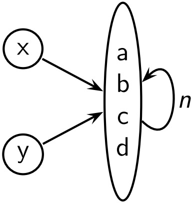
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$ $\forall z \in P_x, Unify(a, z)$
2	$P_y \supseteq \{b\}$ $\forall z \in P_y, Unify(b, z)$
3	$\forall z \in P_x, UnifyPTS(y, z.n)$
4	$UnifyPTS(x, y)$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$ $\forall w \in P_{z.n}, Unify(w, c)$
6	$P_x \supseteq \{d\}$ $\forall z \in P_x, Unify(d, z)$

Points-to Graph



No further change

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

Program

1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

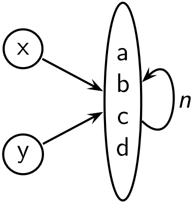
4 $y = x$

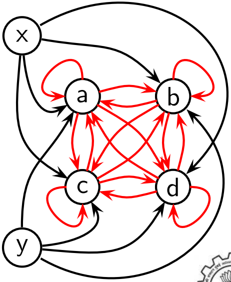
5 $x \rightarrow n = \&c$

6 $x = \&d$

Node	Constraint
1	$P_x \supseteq \{a\}$ $\forall z \in P_x, Unify(a, z)$
2	$P_y \supseteq \{b\}$ $\forall z \in P_y, Unify(b, z)$
3	$\forall z \in P_x, UnifyPTS(y, z.n)$
4	$UnifyPTS(x, y)$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$ $\forall w \in P_{z.n}, Unify(w, c)$
6	$P_x \supseteq \{d\}$ $\forall z \in P_x, Unify(d, z)$

Points-to Graph





Red edges represent field n in the the full blown up graph. It has far more edges than in Andersen's graph

Far more efficient but far less precise

Comparing Equality and Inclusion Based Analyses

- Andersen's algorithm is cubic in number of pointers
- Steensgaard's algorithm is nearly linear in number of pointers

Comparing Equality and Inclusion Based Analyses

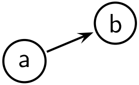
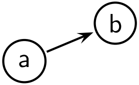
- Andersen's algorithm is cubic in number of pointers
- Steensgaard's algorithm is nearly linear in number of pointers
  - How can it be more efficient by an orders of magnitude?

Efficiency of Equality Based Approach

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

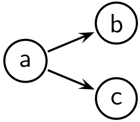
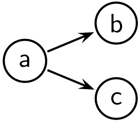
- Andersen's inclusion based wisdom:
  - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
  - Merge multiple successors and maintain a single successor of any node

Efficiency of Equality Based Approach

<table>
<tr><th>Program</th><th>Andersen's approach</th><th>Steensgaard's approach</th></tr>
<tr>
<td>a = &b
a = &c
b.n = &d
b.n = &c</td>
<td>
</td>
<td>
</td>
</tr>
</table>

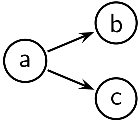
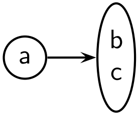
- Andersen's inclusion based wisdom:
  - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
  - Merge multiple successors and maintain a single successor of any node

Efficiency of Equality Based Approach

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

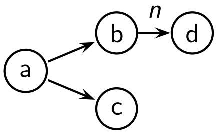
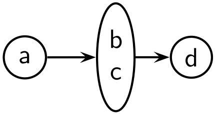
- Andersen's inclusion based wisdom:
  - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
  - Merge multiple successors and maintain a single successor of any node

Efficiency of Equality Based Approach

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

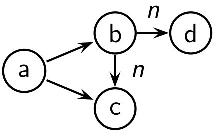
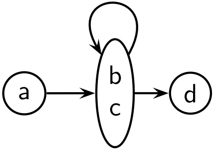
- Andersen's inclusion based wisdom:
  - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
  - Merge multiple successors and maintain a single successor of any node

Efficiency of Equality Based Approach

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

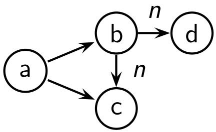
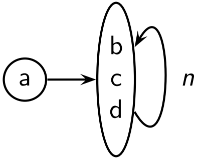
- Andersen's inclusion based wisdom:
  - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
  - Merge multiple successors and maintain a single successor of any node

Efficiency of Equality Based Approach

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

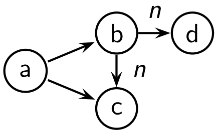
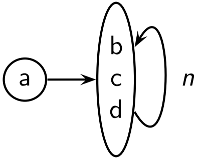
- Andersen's inclusion based wisdom:
  - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
  - Merge multiple successors and maintain a single successor of any node

Efficiency of Equality Based Approach

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

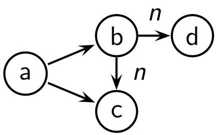
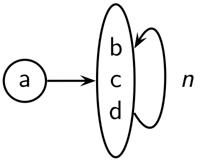
- Andersen's inclusion based wisdom:
  - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
  - Merge multiple successors and maintain a single successor of any node

Efficiency of Equality Based Approach

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

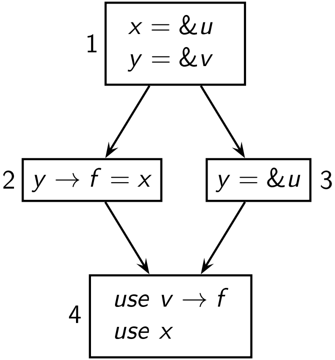
Efficiency of Equality Based Approach

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

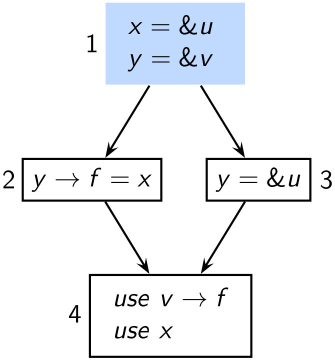
Inclusion Based (aka Andersen's) Points-to Analysis: Example 2

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```

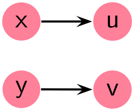


Inclusion Based (aka Andersen's) Points-to Analysis: Example 2

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```



- x "points-to" u
- y "points-to" v



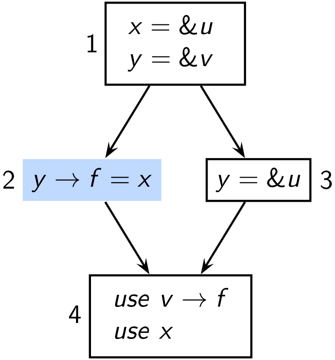
Andersen's Points-to Graph

Constraints on Points-to Sets

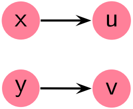
$P_x \supseteq \{u\}$
$P_y \supseteq \{v\}$

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

```
struct s {
    struct s *f;
    int n;
}  *x, *y, u, v;
```



- The f field of pointees of y should point to pointees of x also
- The f field of v should point to u also



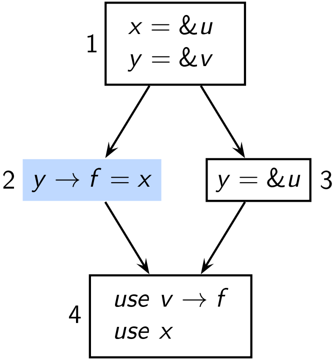
Andersen's Points-to Graph

Constraints on Points-to Sets

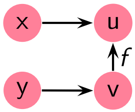
$$P_x \supseteq \{u\}$$
$$P_y \supseteq \{v\}$$
$$\forall w \in P_y,\ P_{w.f} \supseteq P_x$$

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

```
struct s {
    struct s *f;
    int n;
}  *x, *y, u, v;
```



- The f field of pointees of y should point to pointees of x also
- The f field of v should point to u also



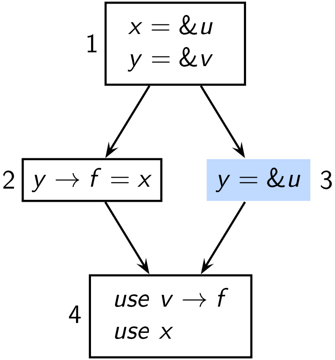
Andersen's Points-to Graph

Constraints on Points-to Sets

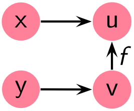
$$P_x \supseteq \{u\}$$
$$P_y \supseteq \{v\}$$
$$\forall w \in P_y,\ P_{w.f} \supseteq P_x$$

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

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```



- y should point to u also



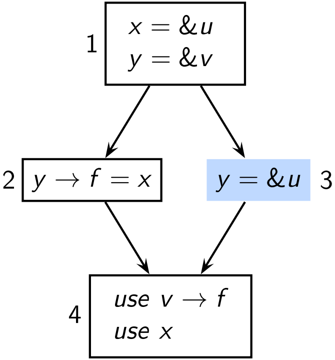
Andersen's Points-to Graph

Constraints on Points-to Sets

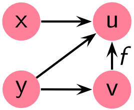
$$P_x \supseteq \{u\}$$
$$P_y \supseteq \{v\}$$
$$\forall w \in P_y,\ P_{w.f} \supseteq P_x$$
$$P_y \supseteq \{u\}$$

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

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```



- y should point to u also



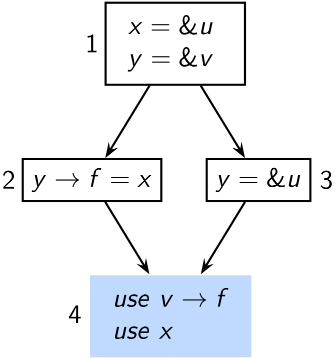
Andersen's Points-to Graph

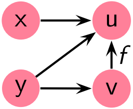
Constraints on Points-to Sets

$$
\begin{aligned}
P_x &\supseteq \{u\}\\
P_y &\supseteq \{v\}\\
\forall w \in P_y,\ P_{w.f} &\supseteq P_x\\
P_y &\supseteq \{u\}
\end{aligned}
$$

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

```
struct s {
    struct s *f;
    int n;
}  *x, *y, u, v;
```





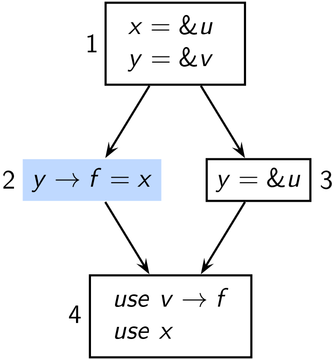
Andersen's Points-to Graph

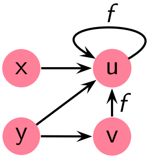
Constraints on Points-to Sets

$$
\begin{aligned}
P_x &\supseteq \{u\} \\
P_y &\supseteq \{v\} \\
\forall w \in P_y,\ P_{w.f} &\supseteq P_x \\
P_y &\supseteq \{u\}
\end{aligned}
$$

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

```
struct s {
    struct s *f;
    int n;
}  *x, *y, u, v;
```





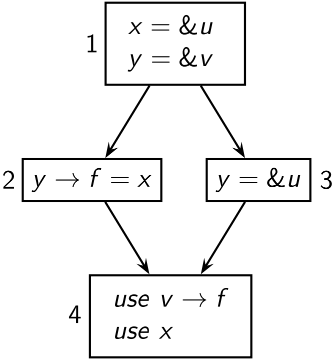
Andersen's Points-to Graph

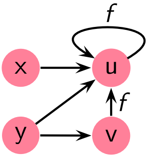
Constraints on Points-to Sets

$$P_x \supseteq \{u\}$$
$$P_y \supseteq \{v\}$$
$$\forall w \in P_y,\ P_{w.f} \supseteq P_x$$
$$P_y \supseteq \{u\}$$

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

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```





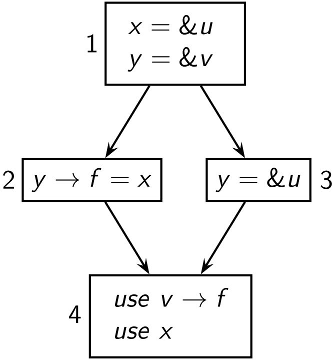
Andersen's Points-to Graph

Constraints on Points-to Sets

$$
\begin{aligned}
P_x &\supseteq \{u\} \\
P_y &\supseteq \{v\} \\
\forall w \in P_y,\ P_{w.f} &\supseteq P_x \\
P_y &\supseteq \{u\}
\end{aligned}
$$

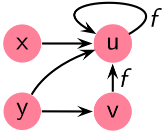
Equality Based (aka Steensgaard's) Points-to Analysis: Example 2

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```



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

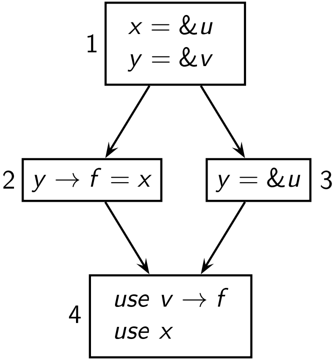
  Pointees of all equivalent locations become equivalent



Andersen's Points-to Graph

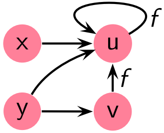
Equality Based (aka Steensgaard's) Points-to Analysis: Example 2

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```



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

  Pointees of all equivalent locations become equivalent



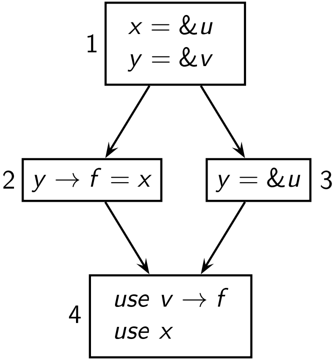
Andersen's Points-to Graph

Effective additional constraints

Unify(u, v) /* pointees of y */

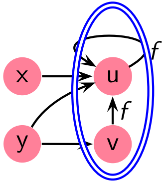
Equality Based (aka Steensgaard's) Points-to Analysis: Example 2

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```



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

  Pointees of all equivalent locations become equivalent



Steengaard's Points-to Graph

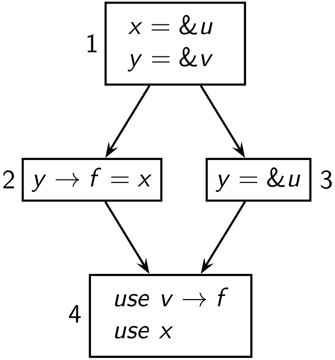
Effective additional constraints

Unify(u, v) /* pointees of y */

$\Rightarrow$ u, v are equivalent

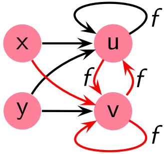
Equality Based (aka Steensgaard's) Points-to Analysis: Example 2

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```



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

  Pointees of all equivalent locations become equivalent



Steengaard's Points-to Graph

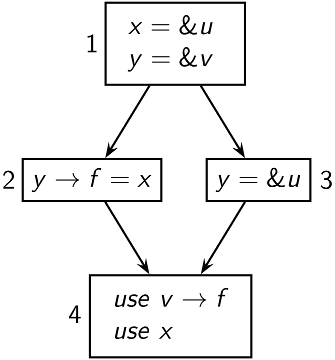
Effective additional constraints

Unify(u, v) /* pointees of y */

$\Rightarrow$ u, v are equivalent

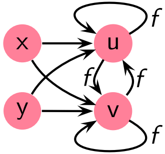
Equality Based (aka Steensgaard's) Points-to Analysis: Example 2

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```



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

  Pointees of all equivalent locations become equivalent



Steengaard's Points-to Graph

Effective additional constraints

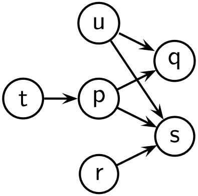
$Unify(u, v)$ /* pointees of y */

$\Rightarrow$ u, v are equivalent

Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

Program	Inclusion based	Equality based
$p = \&q$ $r = \&s$ $t = \&p$ $u = p$ $*t = r$		

Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

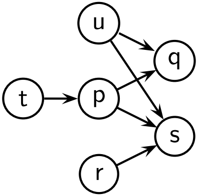
Program	Inclusion based	Equality based
$p = \&q$ $r = \&s$ $t = \&p$ $u = p$ $*t = r$		

Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

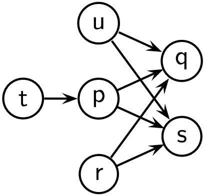
Program

p = &q
r = &s
t = &p
u = p
∗t = r

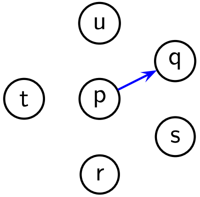
Inclusion based



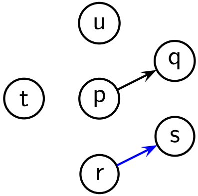
Equality based



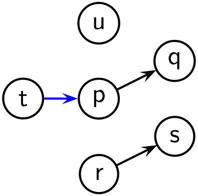
Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

Program	Inclusion based	Equality based
p = &q r = &s t = &p u = p ∗t = r		

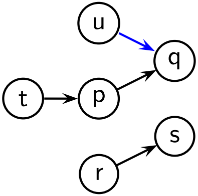
Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

Program	Inclusion based	Equality based
p = &q r = &s t = &p u = p ∗t = r		

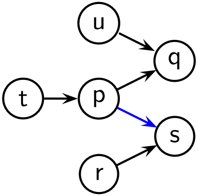
Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

Program	Inclusion based	Equality based
p = &q r = &s t = &p u = p ∗t = r		

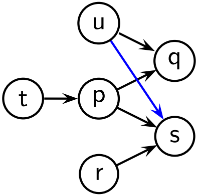
Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

Program	Inclusion based	Equality based
p = &q r = &s t = &p u = p *t = r		

Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

Program	Inclusion based	Equality based
$p = \&q$ $r = \&s$ $t = \&p$ $u = p$ $*t = r$		

Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

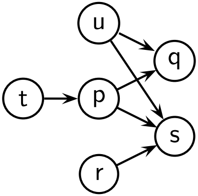
Program	Inclusion based	Equality based
p = &q r = &s t = &p u = p *t = r		

Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

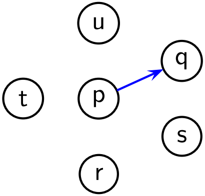
Program

$p = \&q$
$r = \&s$
$t = \&p$
$u = p$
$*t = r$

Inclusion based



Equality based

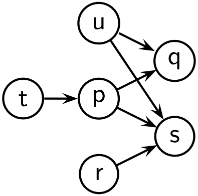


Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

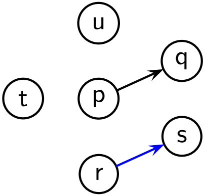
Program

```
p = &q
r = &s
t = &p
u = p
*t = r
```

Inclusion based



Equality based

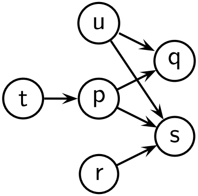


Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

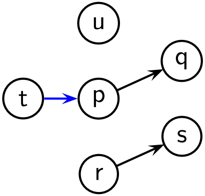
Program

```
p = &q
r = &s
t = &p
u = p
*t = r
```

Inclusion based



Equality based

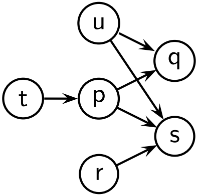


Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

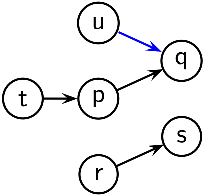
Program

p = &q
r = &s
t = &p
u = p
*t = r

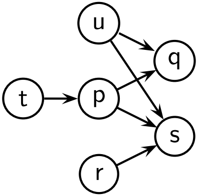
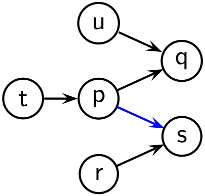
Inclusion based



Equality based



Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

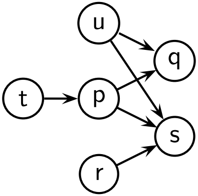
Program	Inclusion based	Equality based
$p = \&q$ $r = \&s$ $t = \&p$ $u = p$ $*t = r$		

Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

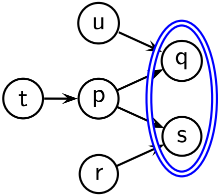
Program

```
p = &q
r = &s
t = &p
u = p
*t = r
```

Inclusion based



Equality based

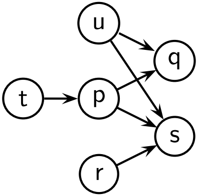


Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

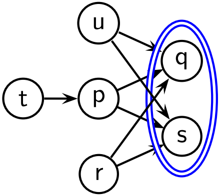
Program

```
p = &q
r = &s
t = &p
u = p
*t = r
```

Inclusion based



Equality based

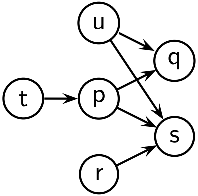


Tutorial Problem for Flow-Insensitive Pointer Analysis (1)

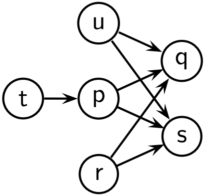
Program

```
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 = &d;
```

An Outline of Pointer Analysis Coverage

- The larger perspective
- Comparing Points-to and Alias information
- Defining Points-to Analysis
- 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

Defining Points-to Analysis: A Recap

δ: a state	RS: reachable states
FS: flow-sensitive	FI: flow-insensitive
$n \in N(p)$: nodes of procedure p	

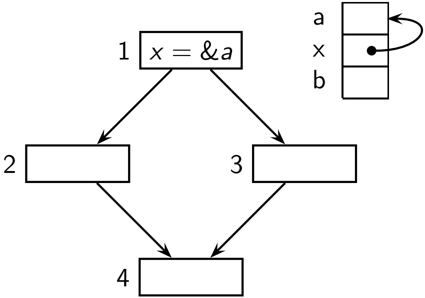
$$RS(n, p) = \{\delta \mid (n, \delta) \text{ occurs in some trace } \tau(p) \text{ of procedure } p\}$$

$$\text{FSMayPT}(n, p) \supseteq \text{IdealMayPT}(n, p) = \bigcup_{\delta \in RS(n,p)} \delta$$

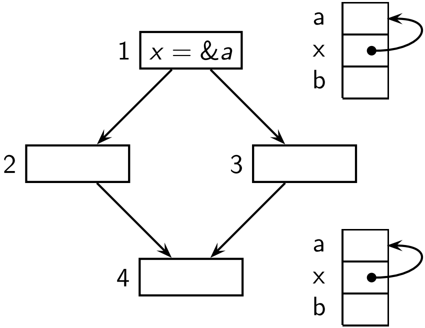
$$\text{FSMustPT}(n, p) \subseteq \text{IdealMustPT}(n, p) = \bigcap_{\delta \in RS(n,p)} \delta$$

$$\text{FIPT}(p) \supseteq \bigcup_{n \in N(p)} \text{FSMayPT}(n, p) \supseteq \bigcup_{n \in N(p)} \text{IdealMayPT}(n, p) \supseteq \bigcup_{\substack{\delta \in RS(n,p) \\ n \in N(p)}} \delta$$

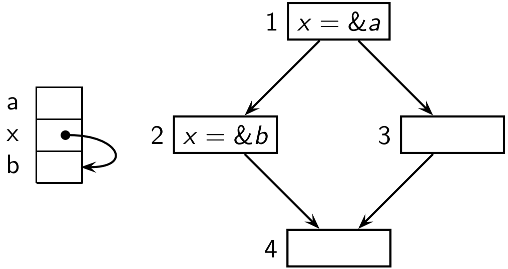
Must Points-to Information



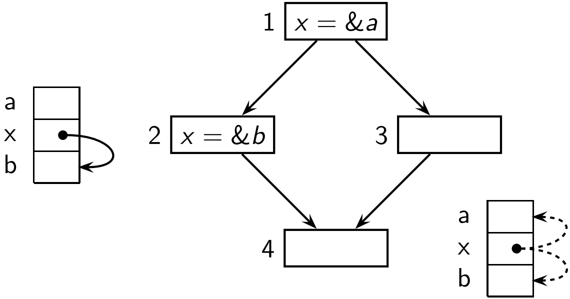
Must Points-to Information



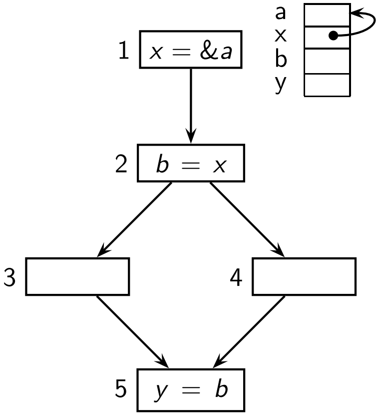
May Points-to Information



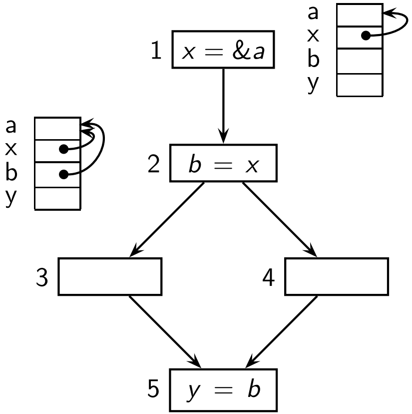
May Points-to Information



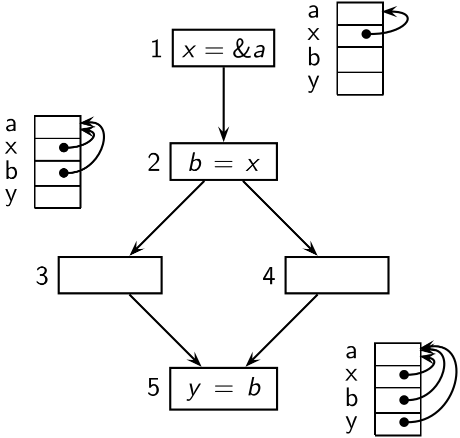
Must Alias Information



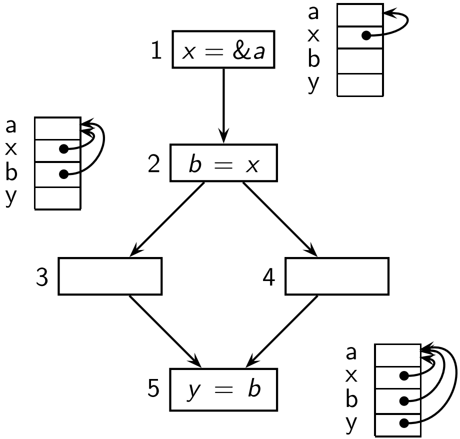
Must Alias Information



Must Alias Information

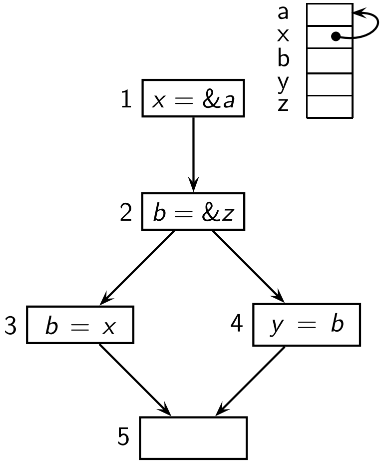


Must Alias Information

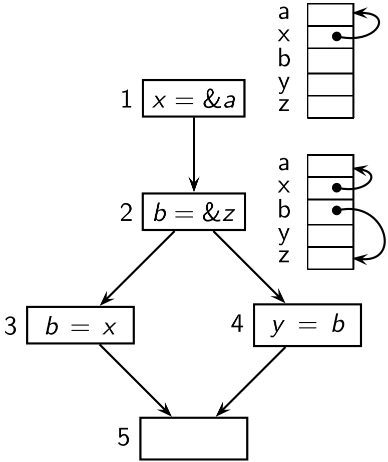


$x \overset{\circ}{=} b$ and $b \overset{\circ}{=} y \Rightarrow x \overset{\circ}{=} y$

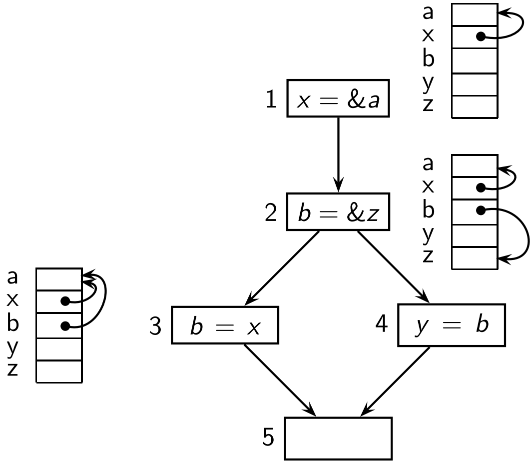
May Alias Information



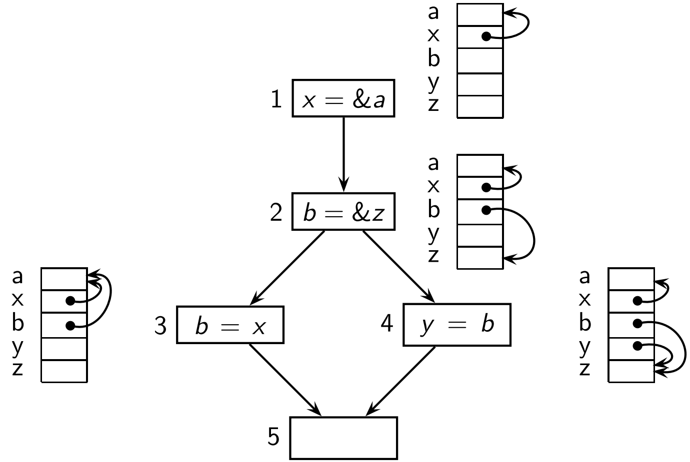
May Alias Information



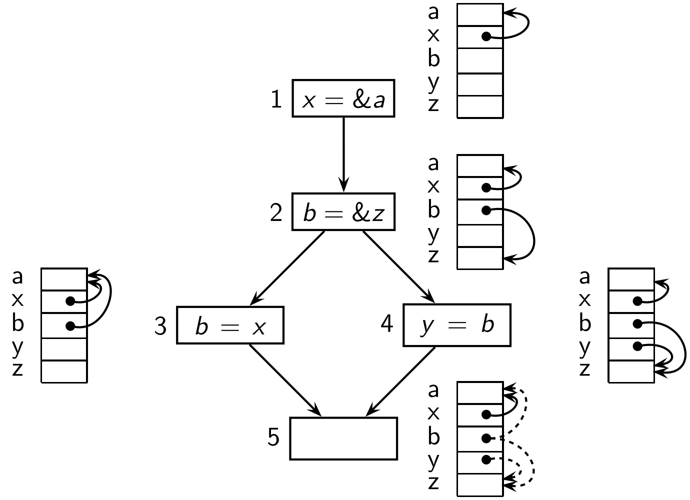
May Alias Information



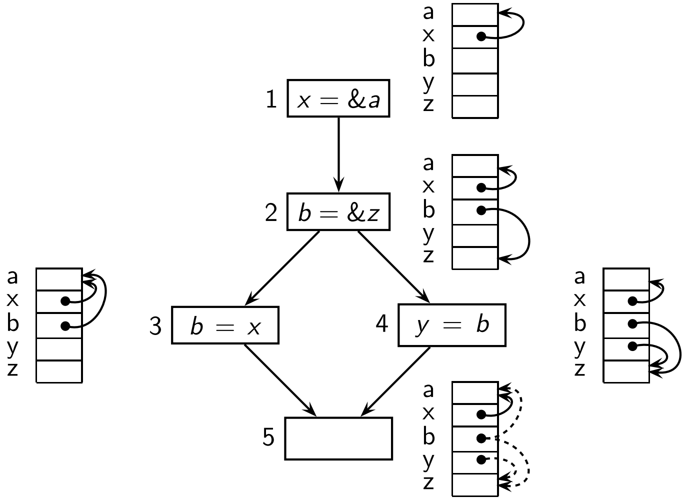
May Alias Information



May Alias Information

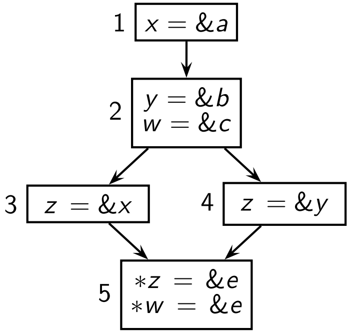


May Alias Information

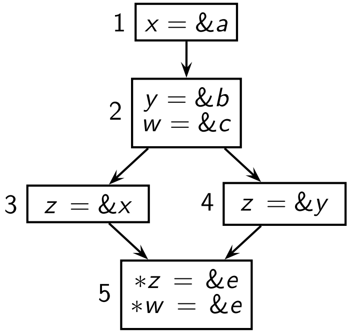


$x \mathrel{\mathring{=}} b$ and $b \mathrel{\mathring{=}} y \not\Rightarrow x \mathrel{\mathring{=}} y$

Strong and Weak Updates



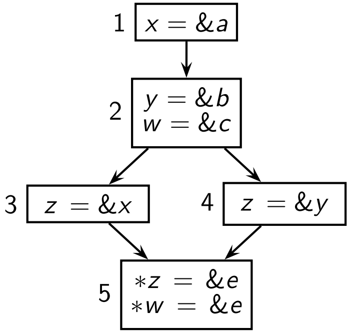
Strong and Weak Updates



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

  Only Gen, No Kill

Strong and Weak Updates



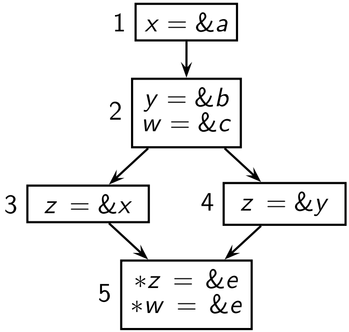
- Weak update: Modification of x or y due to $*z$ in block 5

  Only Gen, No Kill

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

  Both Gen and Kill

Strong and Weak Updates



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

  Only Gen, No Kill

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

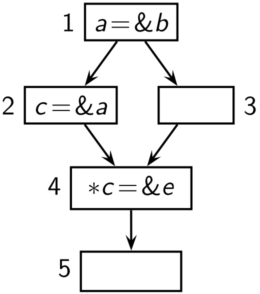
  Both Gen and Kill

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

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

May Points-to Analysis

Must Points-to Analysis



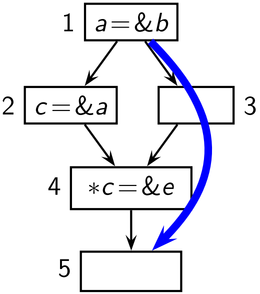
May and Must Analysis for Killing Points-to Information (1)

May Points-to Analysis

Must 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 $\emptyset$
- However, $MayIn_4$ contains (c, a)

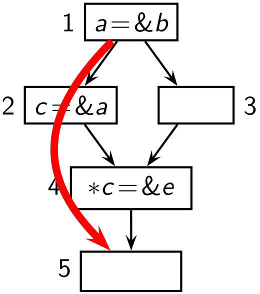


May and Must Analysis for Killing Points-to Information (1)

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 $\emptyset$
- However, $MayIn_4$ contains (c, a)



Must Points-to Analysis

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

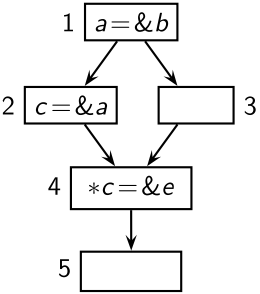
  Does not hold along path 1-2-4
- Block 4 should kill (a, b)
- Possible if pointee set of c is $\{a\}$
- However, the pointee set of c is $\emptyset$ in $MustIn_4$

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) (Use MustIn_4)



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, the pointee set of c is $\emptyset$ in MustIn_4 (Use MayIn_4)

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

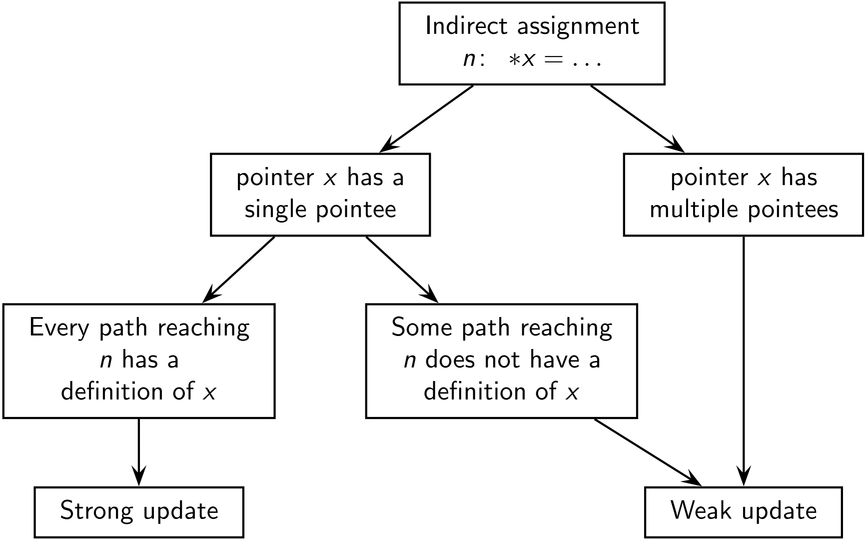
  $\Rightarrow$ should use Must Points-to information

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

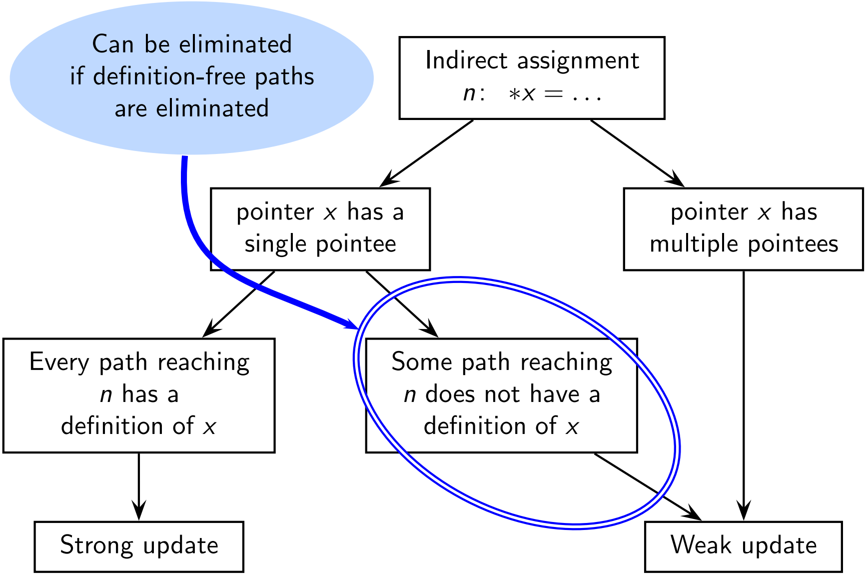
  Kill should remove ALL weak updates

  $\Rightarrow$ should use May Points-to information

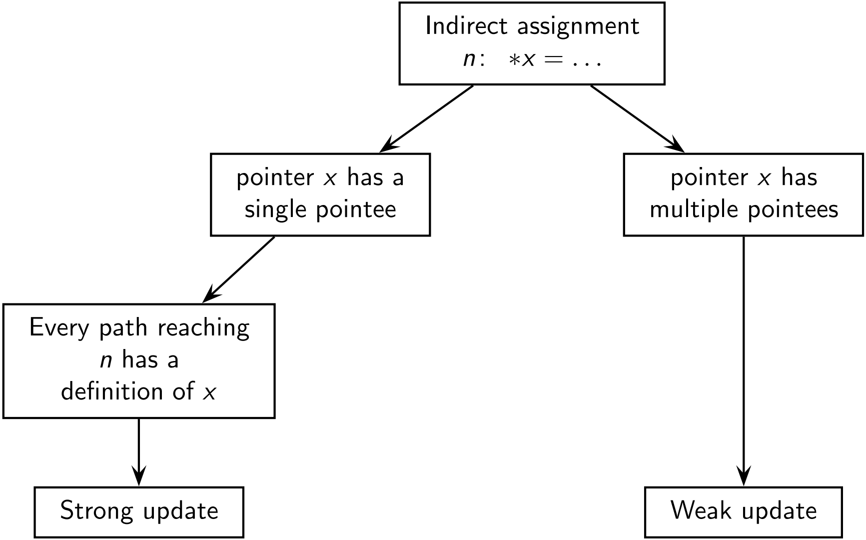
Distinguishing Between Strong and Weak Updates



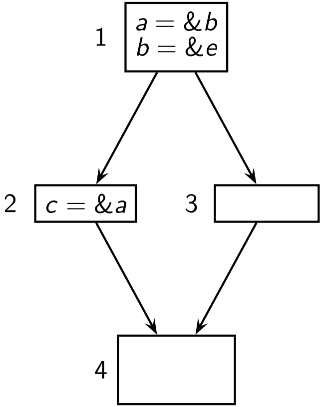
Distinguishing Between Strong and Weak Updates



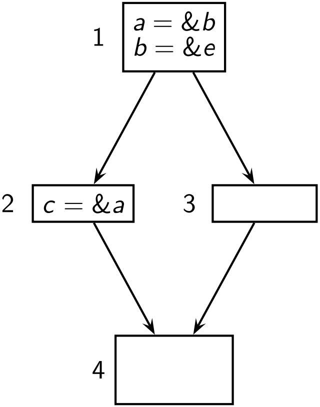
Distinguishing Between Strong and Weak Updates



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



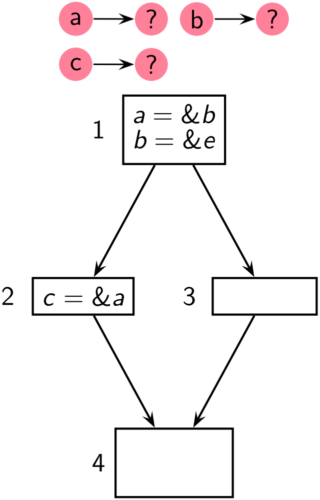
Discovering Must Points-to Information from May Points-to Information



- *BI*. every pointer points to "?"

  Assume that e is a scalar

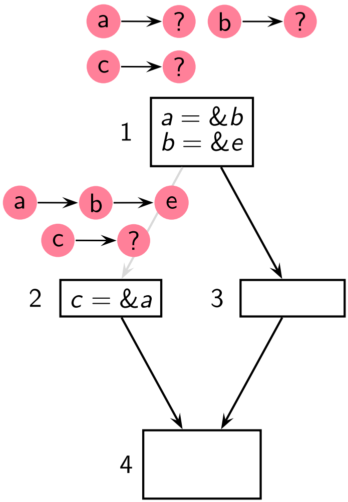
Discovering Must Points-to Information from May Points-to Information



- *BI*. every pointer points to "?"

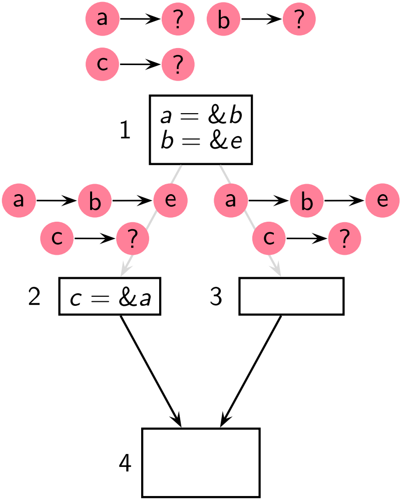
  Assume that *e* is a scalar

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



- *BI*. every pointer points to "?"
  Assume that e is a scalar
- Perform usual may points-to analysis

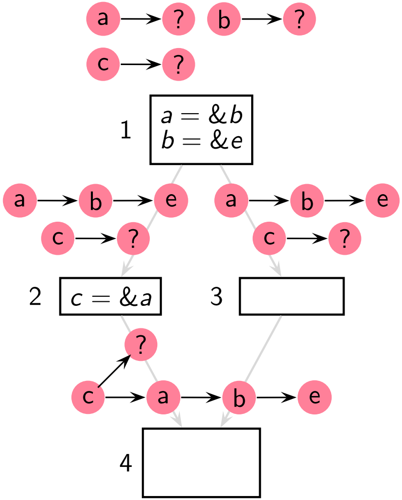
Discovering Must Points-to Information from May Points-to Information



- *BI*. every pointer points to "?"

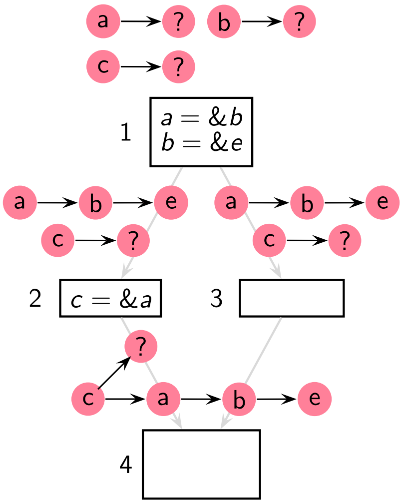
  Assume that *e* is a scalar
- Perform usual may points-to analysis

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



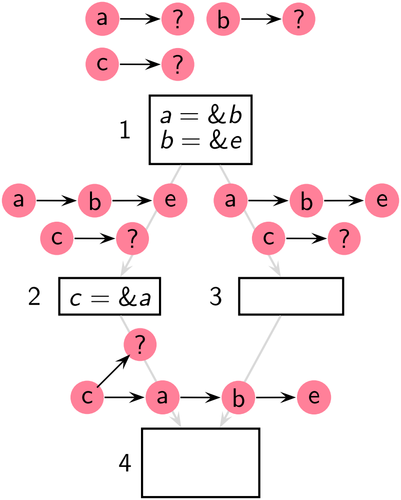
- *BI*. every pointer points to "?"
  Assume that e is a scalar
- Perform usual may points-to analysis

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



- *BI*. every pointer points to "?"
  Assume that e is a scalar
- Perform usual may points-to analysis
- Since c has multiple pointees, it is a MAY relation

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



- *BI.* every pointer points to "?"
  Assume that e is a scalar
- 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 $\mathbf{P} \subseteq V$ be the set of pointer variables
- May-points-to information: $\mathcal{A} = \left\langle 2^{\mathbf{P} \times V}, \supseteq \right\rangle$
- Standard algebraic operations on points-to relations
  Given relation $R \subseteq \mathbf{P} \times V$ and $X \subseteq \mathbf{P}$,
  - Relation *application* $R\,X = \{v \mid u \in X \wedge (u, v) \in R\}$
  - Relation *restriction* $(R|_X)$ $R|_X = \{(u, v) \in R \mid u \in X\}$

Relevant Algebraic Operations on Relations (1)

- Let $\mathbf{P} \subseteq V$ be the set of pointer variables
- May-points-to information: $\mathcal{A} = \left\langle 2^{\mathbf{P} \times V}, \supseteq \right\rangle$
- Standard algebraic operations on points-to relations
  Given relation $R \subseteq \mathbf{P} \times V$ 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* $(R|_X)$ $R|_X = \{(u, v) \in R \mid u \in X\}$

Relevant Algebraic Operations on Relations (1)

- Let $\mathbf{P} \subseteq V$ be the set of pointer variables
- May-points-to information: $\mathcal{A} = \left\langle 2^{\mathbf{P}\times V}, \supseteq \right\rangle$
- Standard algebraic operations on points-to relations
  Given relation $R \subseteq \mathbf{P} \times V$ 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

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

Then,

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

Relevant Algebraic Operations on Relations (2)

Let

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

Then,

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

Relevant Algebraic Operations on Relations (2)

Let

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

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

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

$$Aout_n = \left(Ain_n - \left(Kill_n \times V\right)\right) \cup \left(Def_n \times Pointee_n \right)$$

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

Points-to Analysis Data Flow Equations

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

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

- $Ain/Aout$: 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 for strong update

Points-to Analysis Data Flow Equations

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

$$Aout_n = \Big(Ain_n - \big(Kill_n \times V\big)\Big) \cup \Big(\boxed{Def_n} \times Pointee_n\Big)$$

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

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

Points-to Analysis Data Flow Equations

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

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

$$Aout_n = \Big(Ain_n - \big(\,Kill_n \times V\big)\Big) \cup \Big(\,Def_n \times \boxed{Pointee_n}\Big)$$

- $Ain/Aout$: 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

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

$$Aout_n = \Big(Ain_n - \big(\ Kill_n \times V\big)\Big) \cup \big(\ Def_n \times Pointee_n\ \big)$$

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

Extractor Functions for Points-to Analysis

Values defined in terms of Ain_n (denoted A)

	Def_n	$Kill_n$	$Pointee_n$
$use\ x$			
$x = \&a$			
$x = y$			
$x = *y$			
$*x = y$			
other			

Extractor Functions for Points-to Analysis

Values defined in terms of Ain_n (denoted A)

	$\boxed{Def_n}$	$Kill_n$	$Pointee_n$
use x			
$x = \&a$			
$x = y$			
$x = *y$			
$*x = y$			
other			

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	$Kill_n$	$\boxed{Pointee_n}$
use x			
$x = \&a$			
$x = y$			
$x = *y$			
$*x = y$			
other			

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

Extractor Functions for Points-to Analysis

Values defined in terms of Ain_n (denoted A)

	Def_n	$\boxed{Kill_n}$	$Pointee_n$
use x			
$x = \&a$			
$x = y$			
$x = *y$			
$*x = y$			
other			

Pointers whose points-to relations should be removed for strong update

Extractor Functions for Points-to Analysis

Values defined in terms of Ain_n (denoted A)

	Def_n	$Kill_n$	$Pointee_n$
$use\ x$	$\emptyset$	$\emptyset$	$\emptyset$
$x = \&a$			
$x = y$			
$x = *y$			
$*x = y$			
other			

Extractor Functions for Points-to Analysis

Values defined in terms of Ain_n (denoted A)

	Def_n	$Kill_n$	$Pointee_n$
use x	$\emptyset$	$\emptyset$	$\emptyset$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$
$x = y$			
$x = *y$			
$*x = y$			
other			

Extractor Functions for Points-to Analysis

Values defined in terms of Ain_n (denoted A)

	Def_n	$Kill_n$	$Pointee_n$
$use\ x$	$\emptyset$	$\emptyset$	$\emptyset$
$x = \&a$	$\{x\}$	$\{x\}$	$\{a\}$
$x = y$	$\{x\}$	$\{x\}$	$\boxed{A\{y\}}$
$x = *y$			
$*x = y$			
other			

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

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			

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			

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

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			

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

Find out must-pointees of all pointers

Extractor Functions for Points-to Analysis

Values defined in terms of Ain

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			

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

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

Extractor Functions for Points-to Analysis

Values defined in terms of Ain

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			

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

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

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

Extractor Functions for Points-to Analysis

Values defined in terms of Ain_n (denoted A)

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

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

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

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

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	$\emptyset$	$\emptyset$	$\emptyset$

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

Extractor Functions for Points-to Analysis

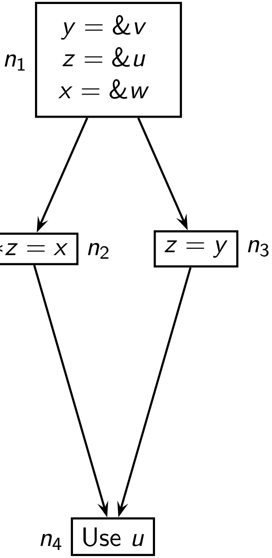
Values defined in terms of Ain_n (denoted A)

	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	$\emptyset$	$\emptyset$	$\emptyset$

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

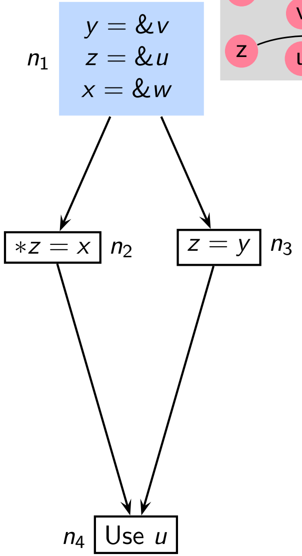
An Example of Flow-Sensitive May Points-to Analysis

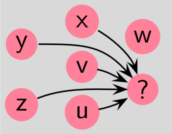
```
int w;
int *u, *v, *x;
int **y, **z;
```



An Example of Flow-Sensitive May Points-to Analysis

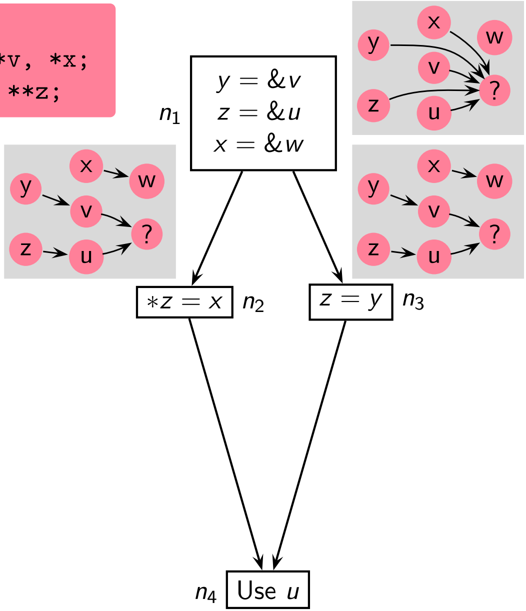
```
int w;
int *u, *v, *x;
int **y, **z;
```





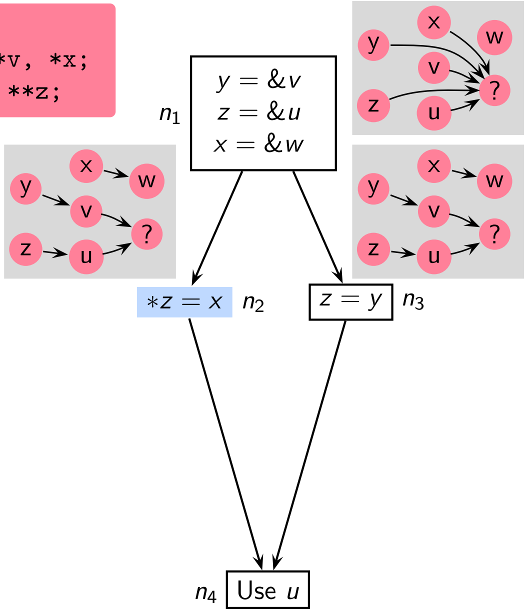
An Example of Flow-Sensitive May Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



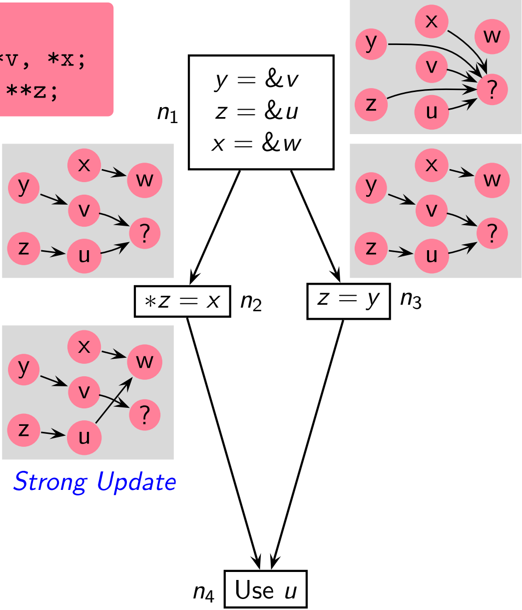
An Example of Flow-Sensitive May Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



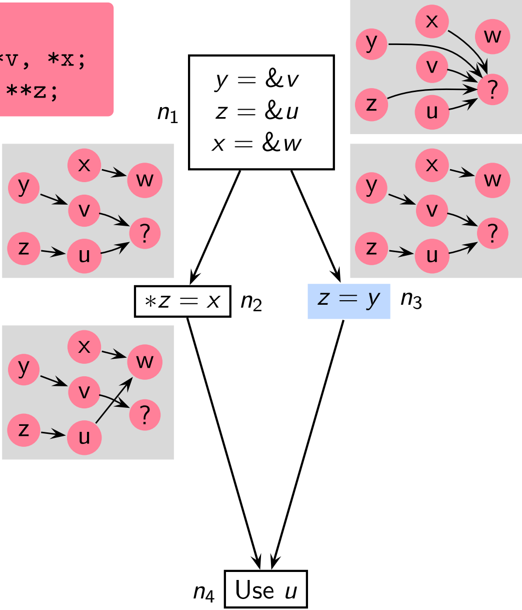
An Example of Flow-Sensitive May Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



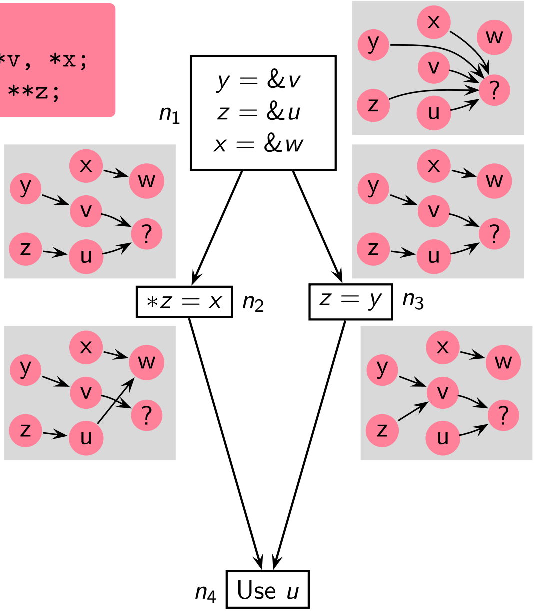
An Example of Flow-Sensitive May Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



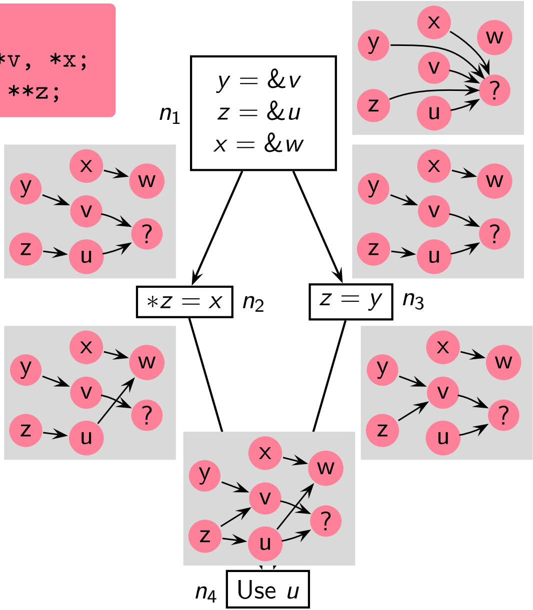
An Example of Flow-Sensitive May Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



An Example of Flow-Sensitive May Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



Extractor Functions in the Presence of Structures (1)

- We extend pointer to use field names as follows:
  - pointer x is represented by $(x, *)$, and
  - pointer field f of structure variable x is represented by (x, f)
  - points-to information is of the form $((x, f)\, y)$
- For simplicity, we
  - separate LHS and RHS assuming that
  - only legal, type-correct pointer expressions are used in a statement
- From LHS, we extract *Def* and Kill as the sets of $(x, *)$ or (a, f) (x is a pointer variable and a is a structure variable)
- From RHS, we extract *Pointee* as the sets of variables x

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*

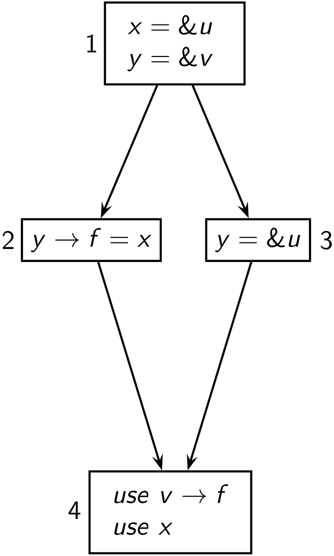
  We will later introduce heap using the allocation site based abstraction

Extractor Functions in the Presence of Structures (2)

LHS	Def_n	$Kill_n$
x	$\{(x,*)\}$	$\{(x,*)\}$
$*x$	$\{(z,*) \mid z \in A\{(x,*)\}\}$	$\{(z,*) \mid z \in Must(A)\{(x,*)\}\}$
$x \rightarrow f$	$\{(z,f) \mid z \in A\{(x,*)\}\}$	$\{(z,f) \mid z \in Must(A)\{(x,*)\}\}$
$x.f$	$\{(x,f)\}$	$\{(x,f)\}$

RHS	$Pointee_n$
$\&y$	$\{y\}$
y	$\{z \mid z \in A\{(y,*)\}\}$
$*y$	$\{z \mid z \in A\{(w,*)\}, w \in A\{(y,*)\}\}$
$y \rightarrow f$	$\{z \mid z \in A\{(w,f)\}, w \in A\{(y,*)\}\}$
$y.f$	$\{z \mid z \in A\{(y,f)\}\}$

An Example of Flow-Sensitive May Points-to Analysis



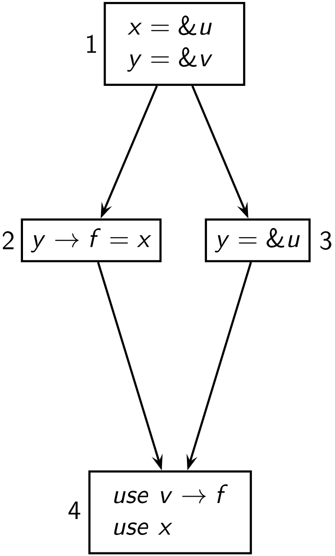
Type Information

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```

Andersen's Points-to Graph

Steensgaard's Points-to Graph

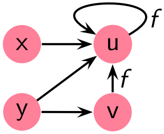
An Example of Flow-Sensitive May Points-to Analysis



Type Information

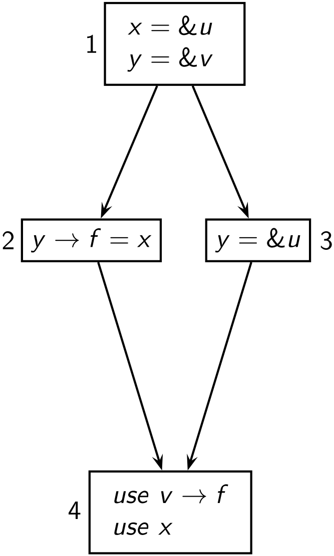
```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```

Andersen's Points-to Graph



Steensgaard's Points-to Graph

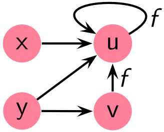
An Example of Flow-Sensitive May Points-to Analysis



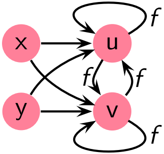
Type Information

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```

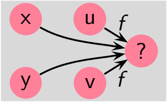
Andersen's Points-to Graph

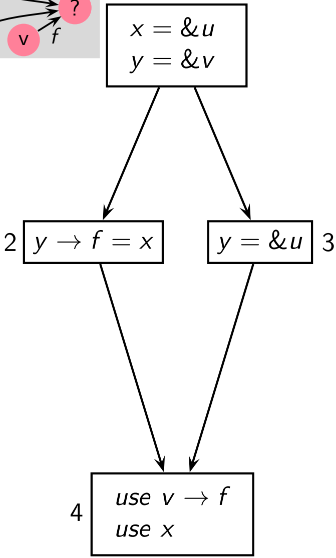


Steensgaard's Points-to Graph



An Example of Flow-Sensitive May Points-to Analysis

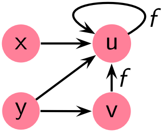




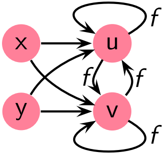
Type Information

```
struct s {
    struct s *f;
    int n;
}  *x, *y, u, v;
```

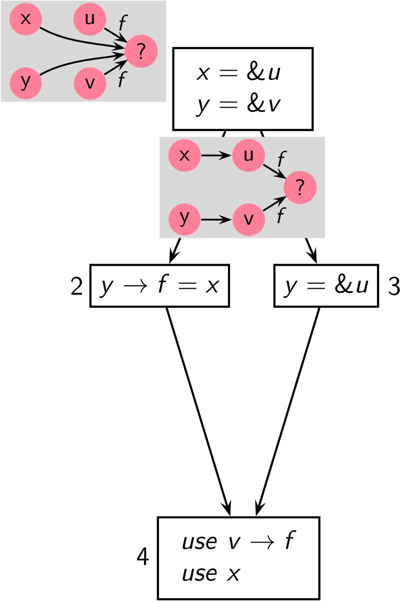
Andersen's Points-to Graph



Steensgaard's Points-to Graph



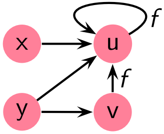
An Example of Flow-Sensitive May Points-to Analysis



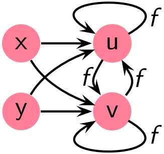
Type Information

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```

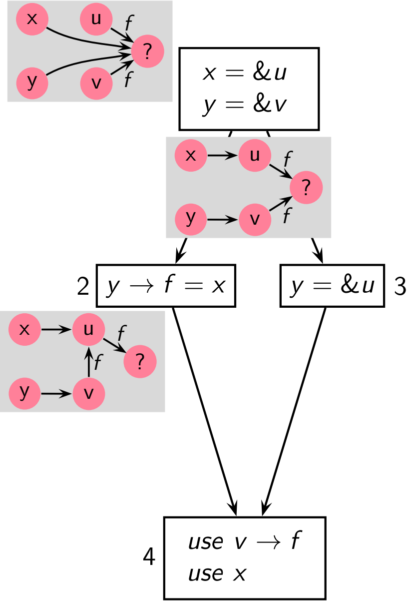
Andersen's Points-to Graph



Steensgaard's Points-to Graph



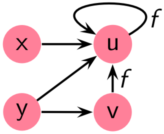
An Example of Flow-Sensitive May Points-to Analysis



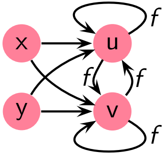
Type Information

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```

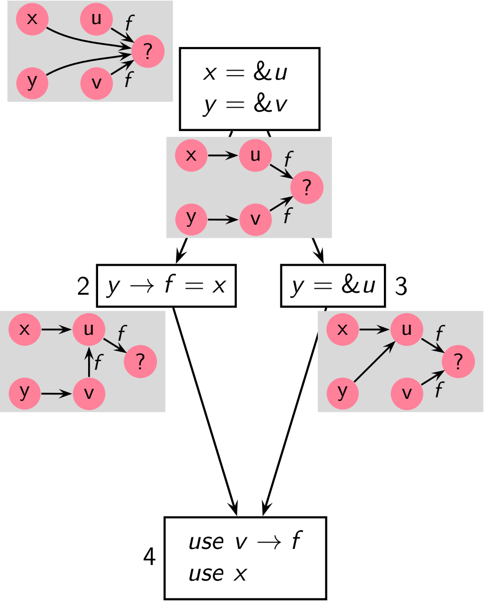
Andersen's Points-to Graph



Steensgaard's Points-to Graph



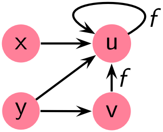
An Example of Flow-Sensitive May Points-to Analysis



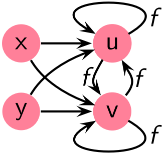
Type Information

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```

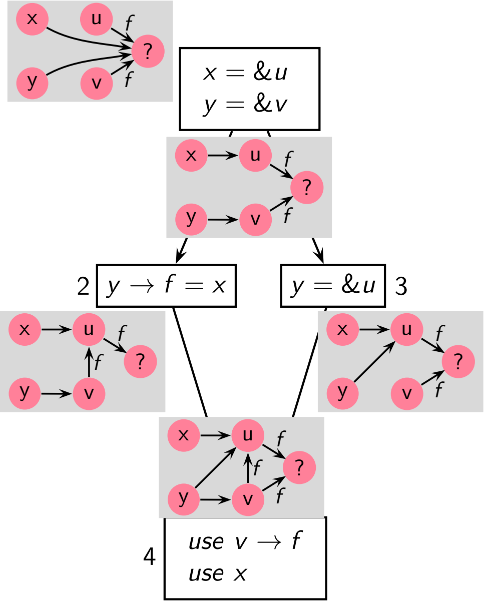
Andersen's Points-to Graph



Steensgaard's Points-to Graph



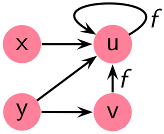
An Example of Flow-Sensitive May Points-to Analysis



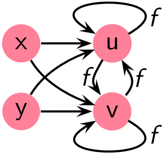
Type Information

```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```

Andersen's Points-to Graph



Steensgaard's Points-to Graph



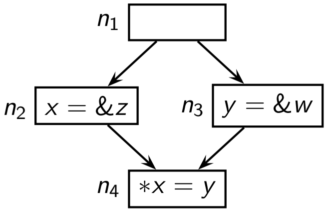
Tutorial Problems for Flow-Sensitive Pointer Analysis (2)

Compute May and Must points-to information

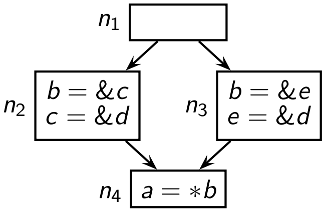
```
if (...)
        p = &x;
else
        p = &y;
x = &a;
y = &b;
*p = &c;
*y = &a;
```

Non-Distributivity of Points-to Analysis

May Points-to

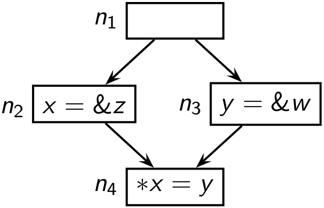


Must Points-to



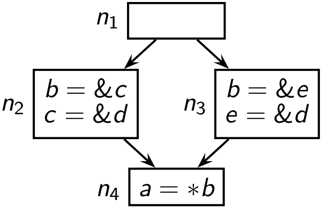
Non-Distributivity of Points-to Analysis

May Points-to



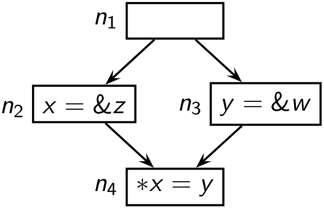
$z \rightarrowtail w$ is spurious

Must Points-to



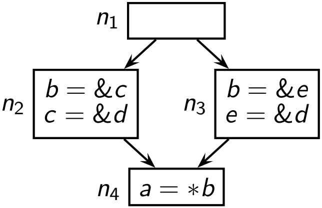
Non-Distributivity of Points-to Analysis

May Points-to



$z \rightarrowtail w$ is spurious

Must Points-to

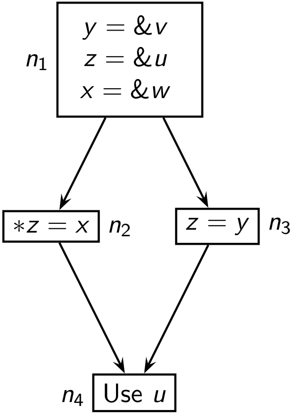


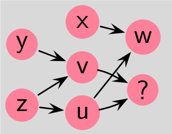
$a \rightarrowtail d$ is missing

An Outline of Pointer Analysis Coverage

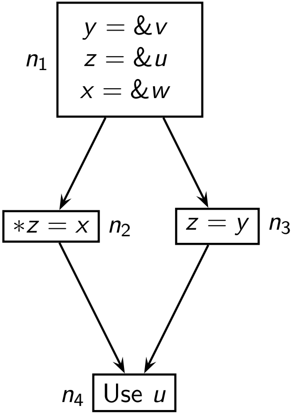
- The larger perspective
- Comparing Points-to and Alias information
- Defining Points-to Analysis
- Flow-Insensitive Points-to Analysis
- Flow-Sensitive Points-to Analysis
- Pointer Analyses: An Engineer's Landscape Next Topic
- Liveness Based Points-to Analysis
- Generalizations to Heap, Arrays, Pointer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

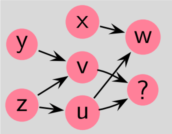
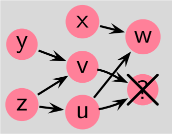
Comparing Points-to Analysis Variants for Flow Sensitivity



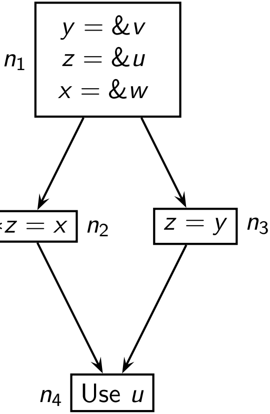
Flow sensitive may points-to graph at n_4	
Andersen's points-to graph	
Steensgaard's points-to graph	

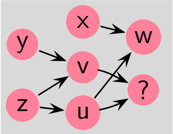
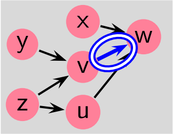
Comparing Points-to Analysis Variants for Flow Sensitivity



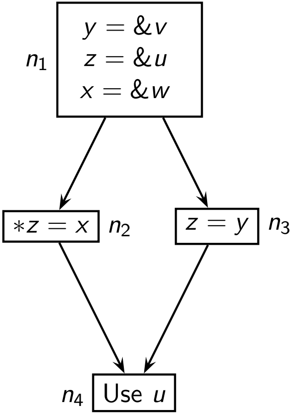
Flow sensitive may points-to graph at n_4	
Andersen's points-to graph	
Steensgaard's points-to graph	

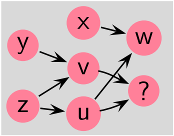
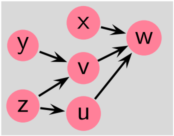
Comparing Points-to Analysis Variants for Flow Sensitivity



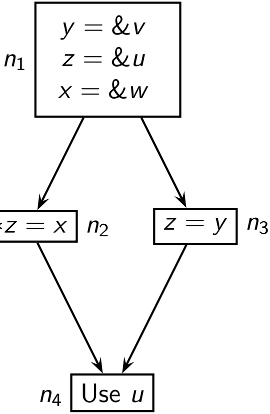
Flow sensitive may points-to graph at n_4	
Andersen's points-to graph	
Steensgaard's points-to graph	

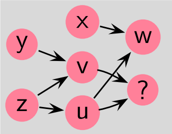
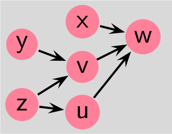
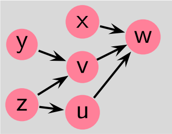
Comparing Points-to Analysis Variants for Flow Sensitivity



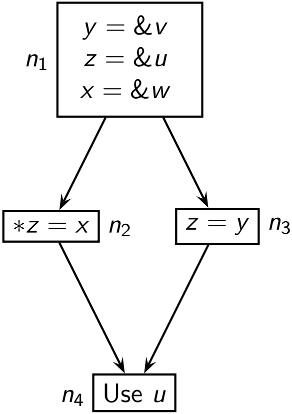
Flow sensitive may points-to graph at n_4	
Andersen's points-to graph	
Steensgaard's points-to graph	

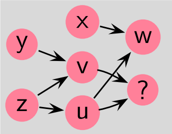
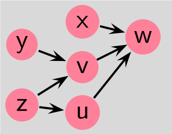
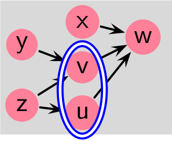
Comparing Points-to Analysis Variants for Flow Sensitivity



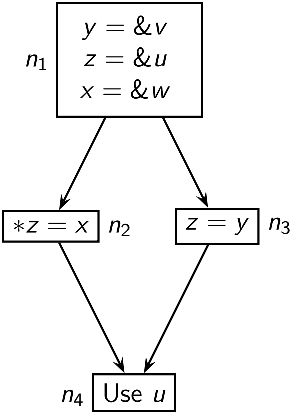
Flow sensitive may points-to graph at n_4	
Andersen's points-to graph	
Steensgaard's points-to graph	

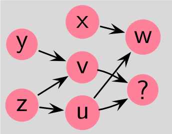
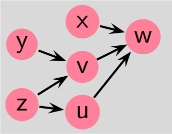
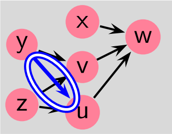
Comparing Points-to Analysis Variants for Flow Sensitivity



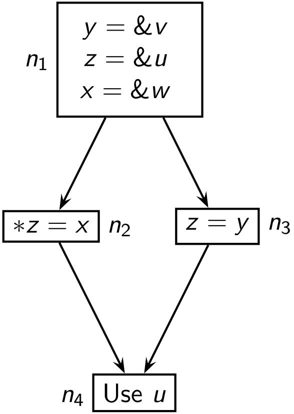
Flow sensitive may points-to graph at n_4	
Andersen's points-to graph	
Steensgaard's points-to graph	

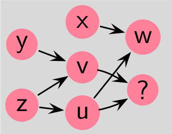
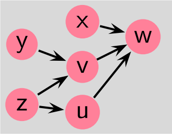
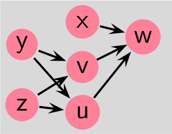
Comparing Points-to Analysis Variants for Flow Sensitivity



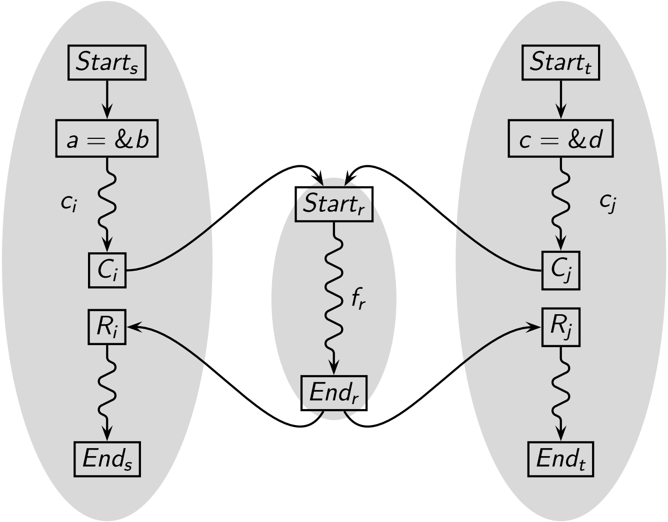
Flow sensitive may points-to graph at n_4	
Andersen's points-to graph	
Steensgaard's points-to graph	

Comparing Points-to Analysis Variants for Flow Sensitivity

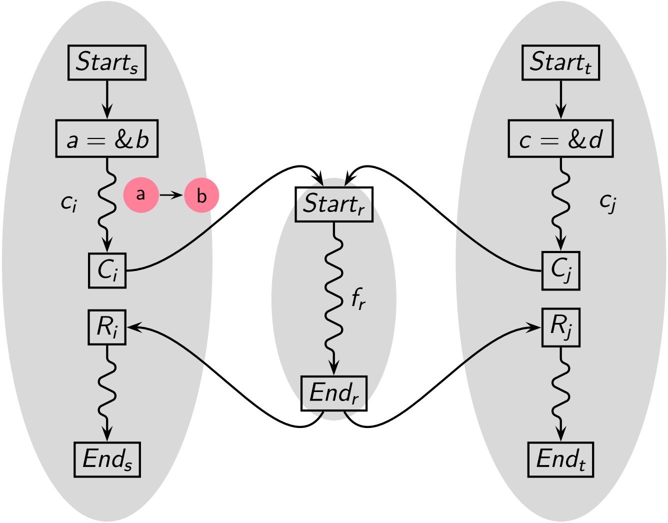


Flow sensitive may points-to graph at n_4	
Andersen's points-to graph	
Steensgaard's points-to graph	

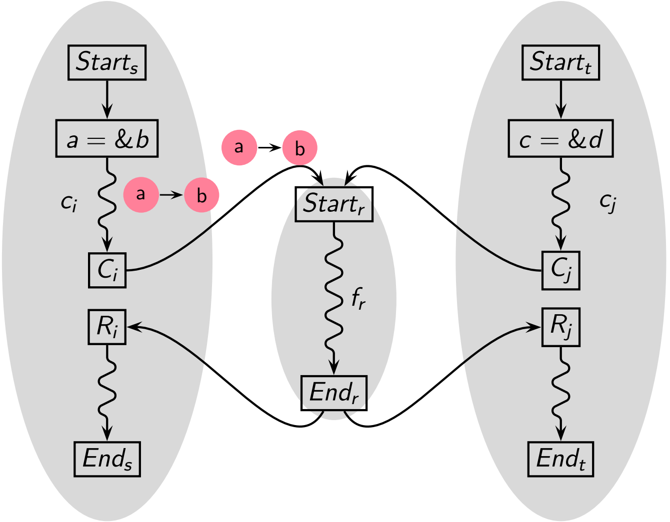
Context Sensitivity in Interprocedural Analysis



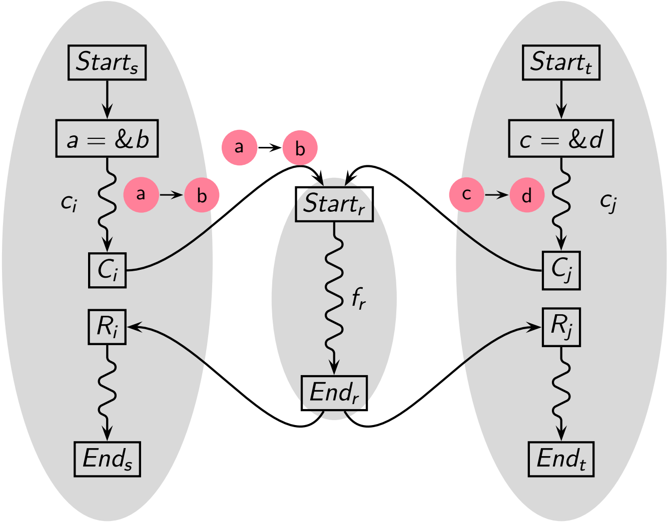
Context Sensitivity in Interprocedural Analysis



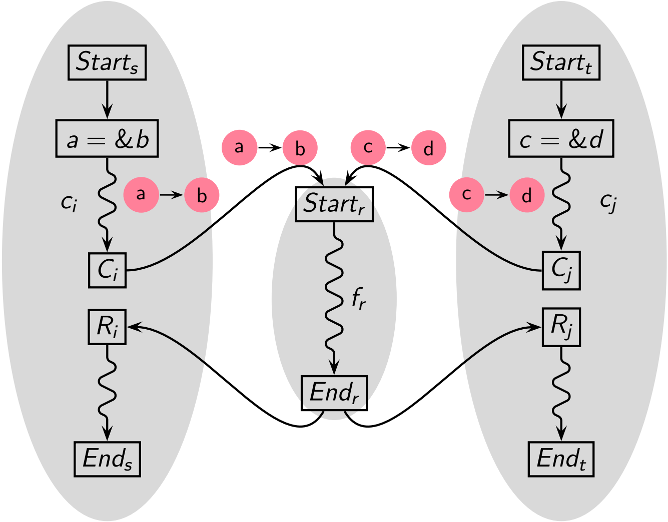
Context Sensitivity in Interprocedural Analysis



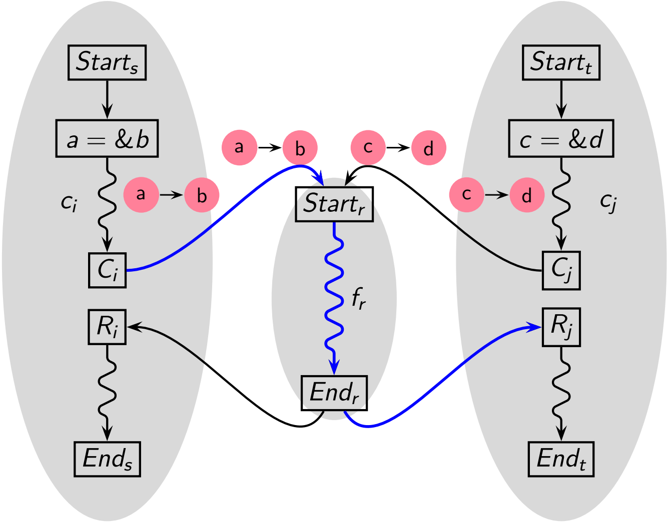
Context Sensitivity in Interprocedural Analysis



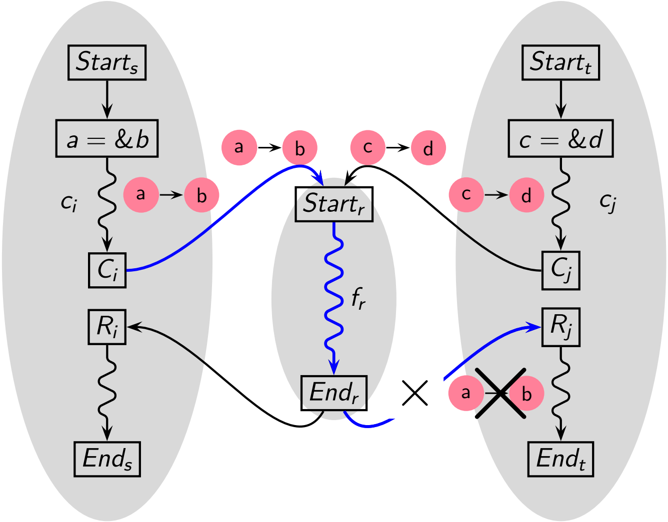
Context Sensitivity in Interprocedural Analysis



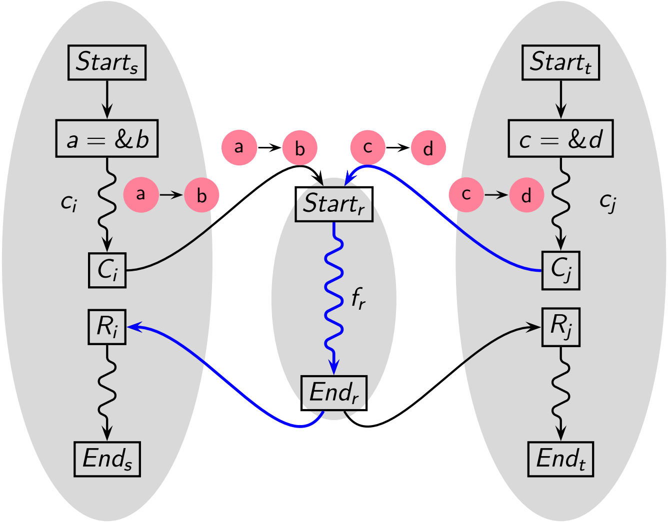
Context Sensitivity in Interprocedural Analysis



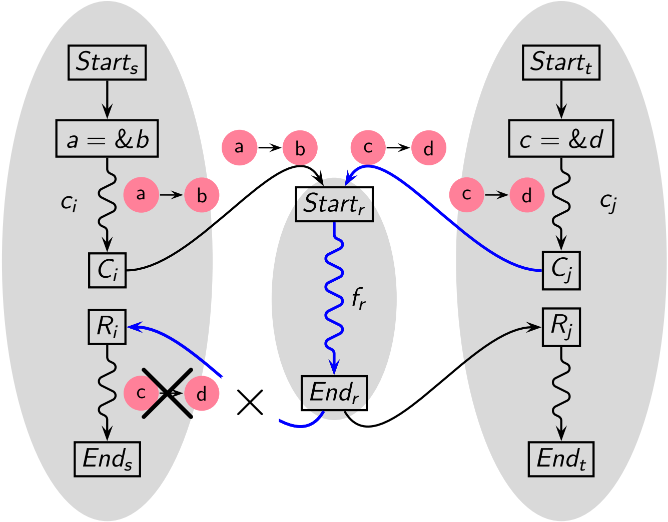
Context Sensitivity in Interprocedural Analysis



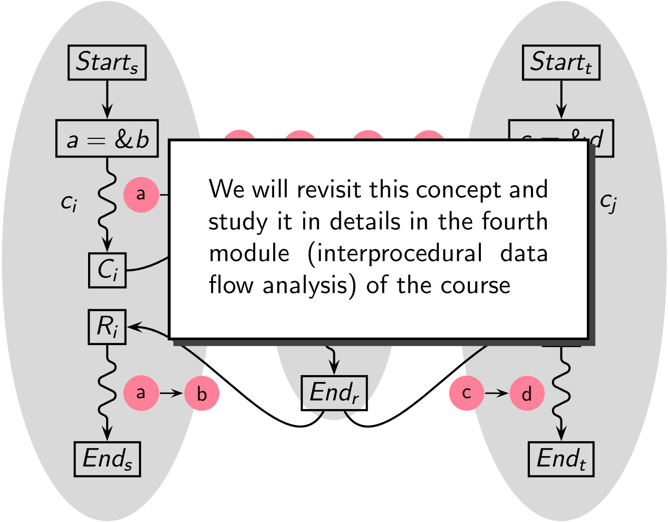
Context Sensitivity in Interprocedural Analysis



Context Sensitivity in Interprocedural Analysis

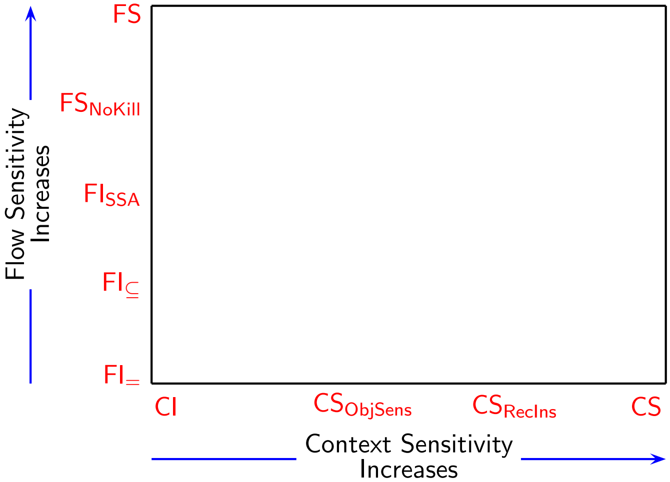


Context Sensitivity in Interprocedural Analysis

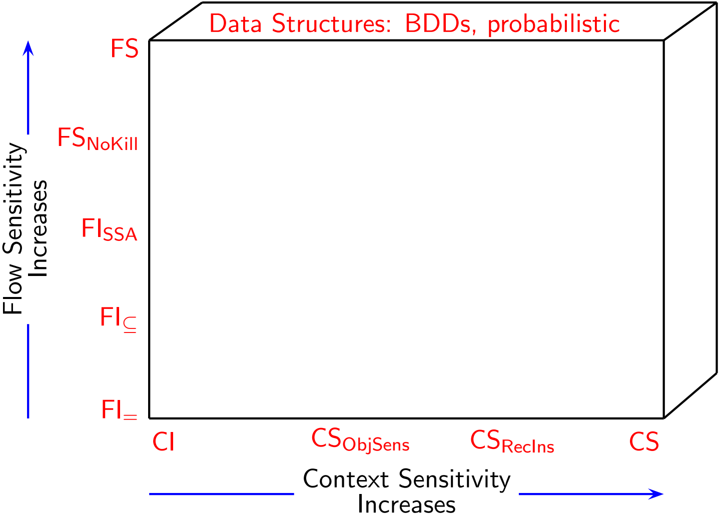


We will revisit this concept and study it in details in the fourth module (interprocedural data flow analysis) of the course

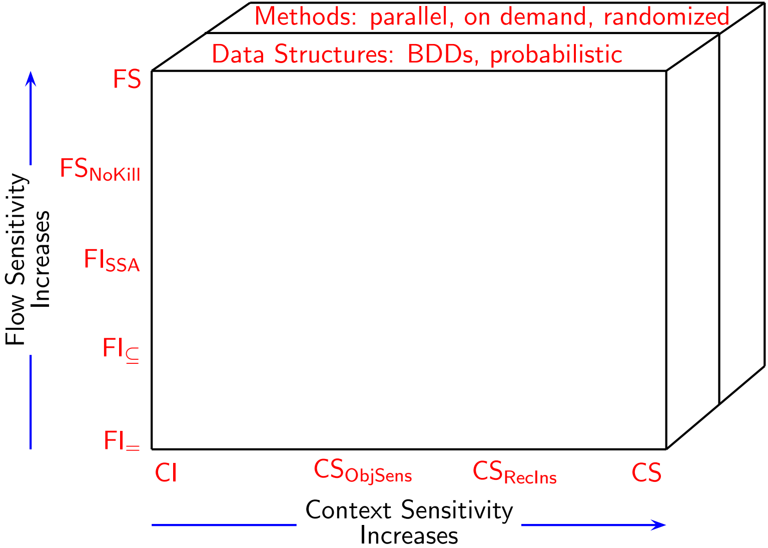
Pointer Analysis: An Engineer's Landscape



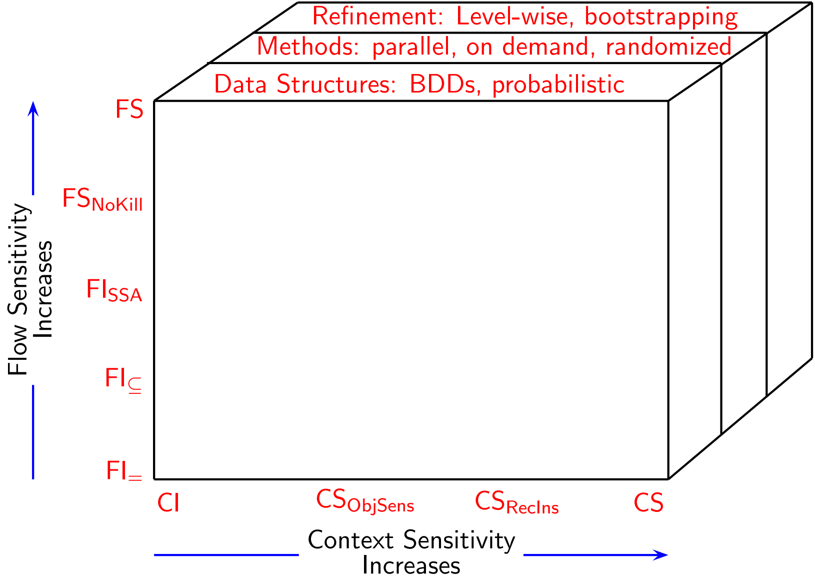
Pointer Analysis: An Engineer's Landscape



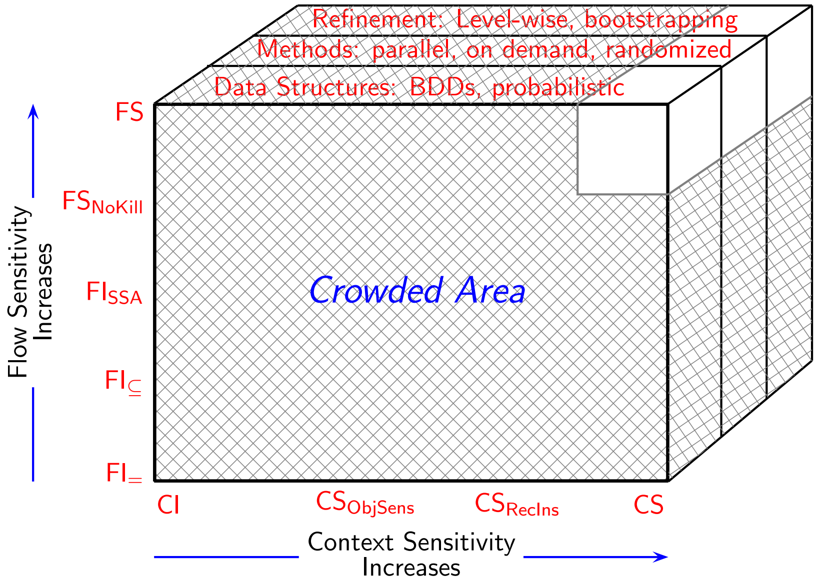
Pointer Analysis: An Engineer's Landscape



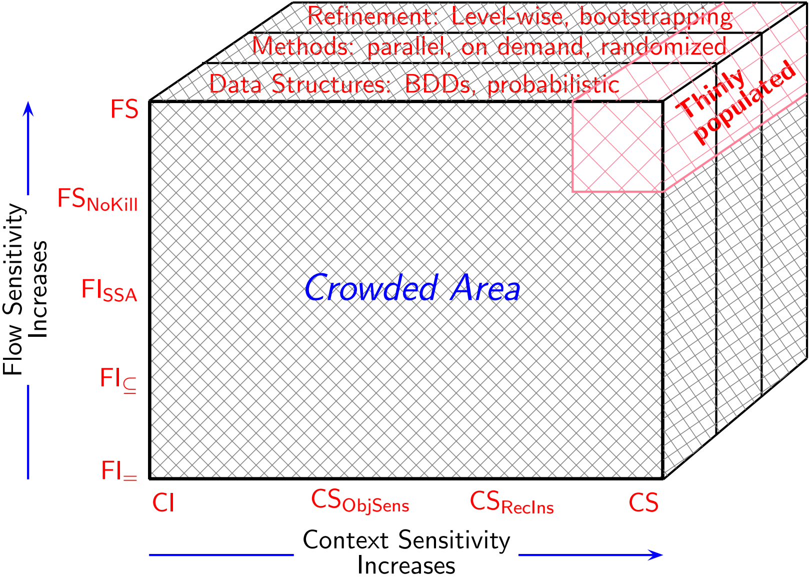
Pointer Analysis: An Engineer's Landscape



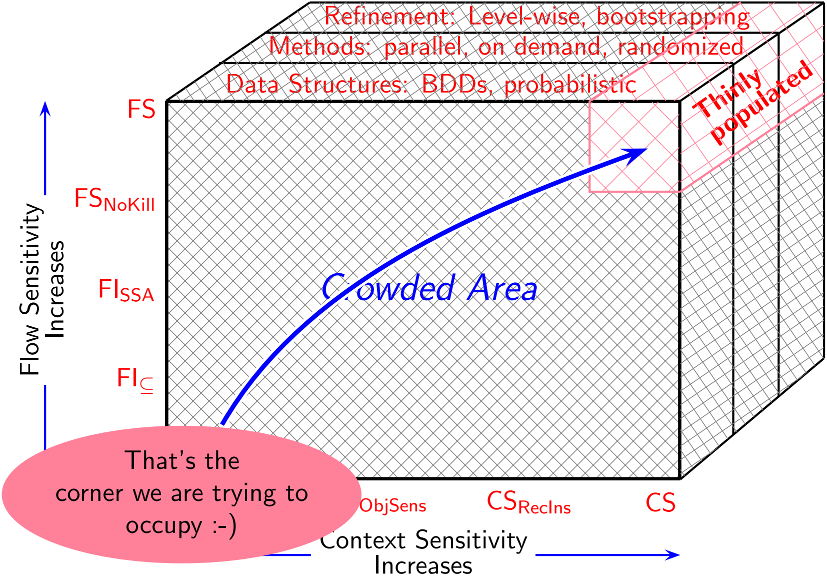
Pointer Analysis: An Engineer's Landscape



Pointer Analysis: An Engineer's Landscape



Pointer Analysis: An Engineer's Landscape

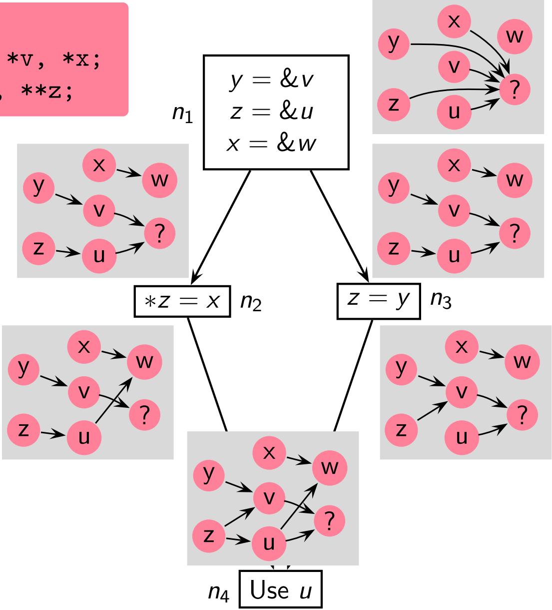


An Outline of Pointer Analysis Coverage

- The larger perspective
- Comparing Points-to and Alias information
- Defining Points-to Analysis
- 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

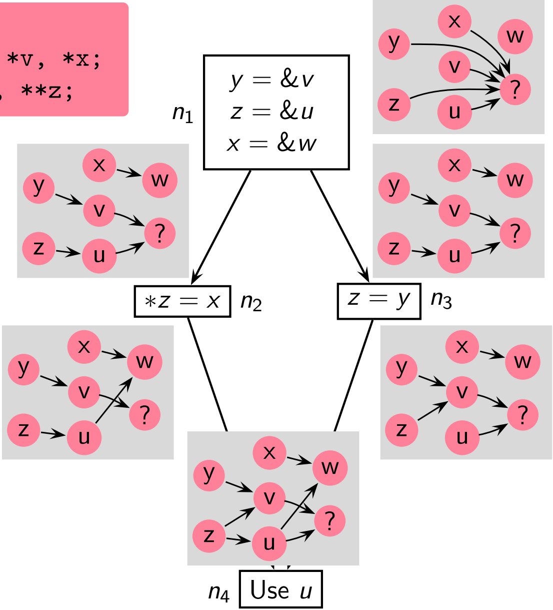
Our Motivating Example for FCPA

```
int w;
int *u, *v, *x;
int **y, **z;
```



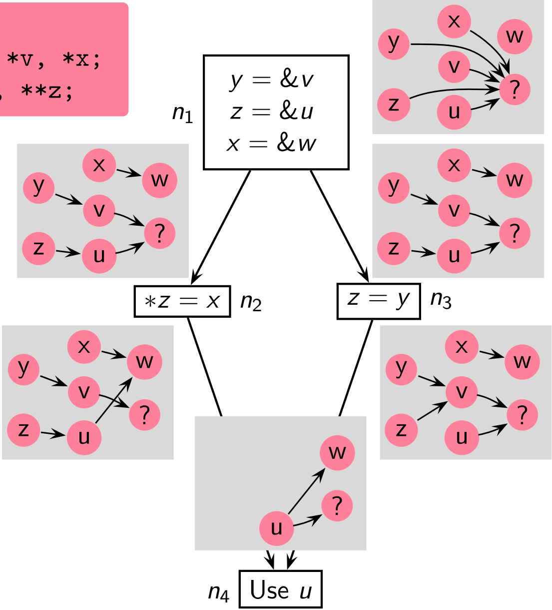
Is All This Information Useful

```
int w;
int *u, *v, *x;
int **y, **z;
```



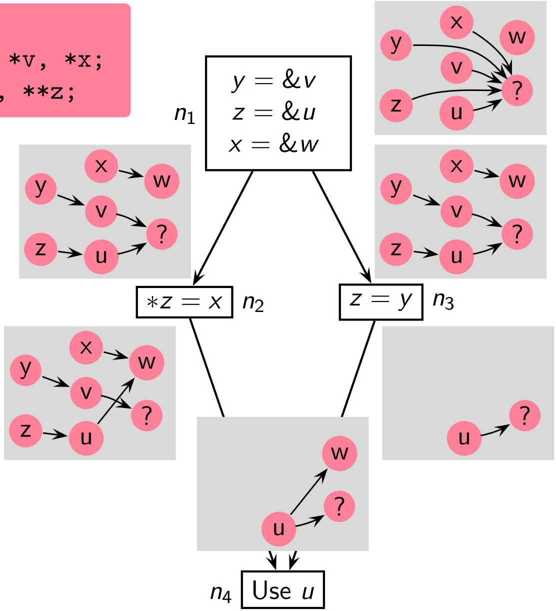
Is All This Information Useful

```
int w;
int *u, *v, *x;
int **y, **z;
```



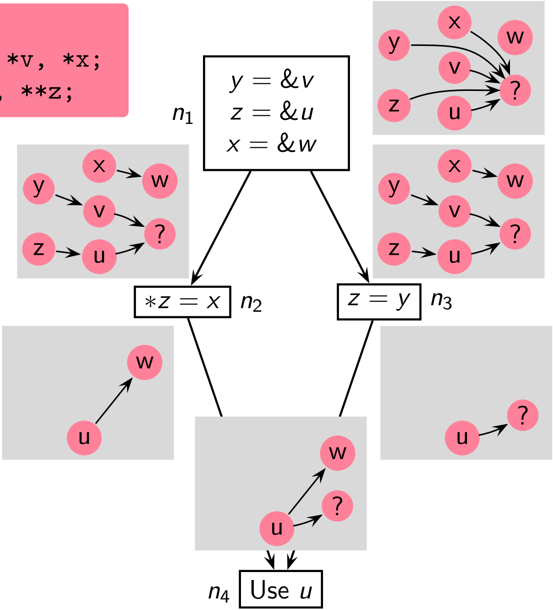
Is All This Information Useful

```
int w;
int *u, *v, *x;
int **y, **z;
```



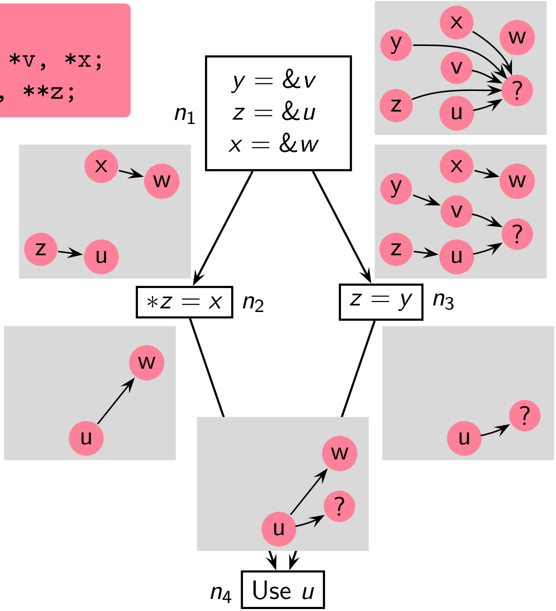
Is All This Information Useful

```
int w;
int *u, *v, *x;
int **y, **z;
```



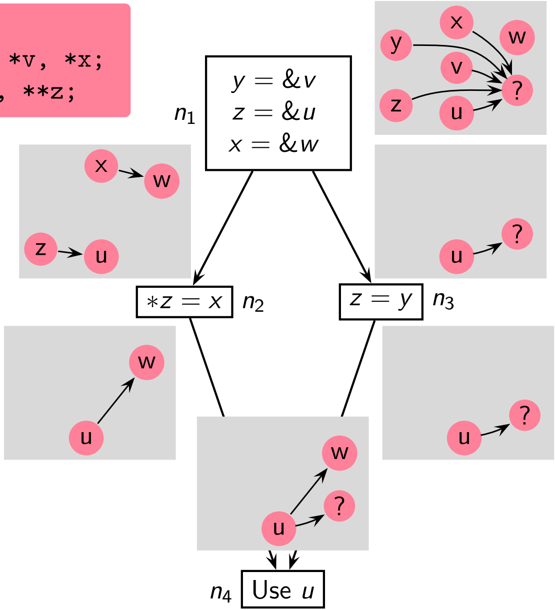
Is All This Information Useful

```
int w;
int *u, *v, *x;
int **y, **z;
```



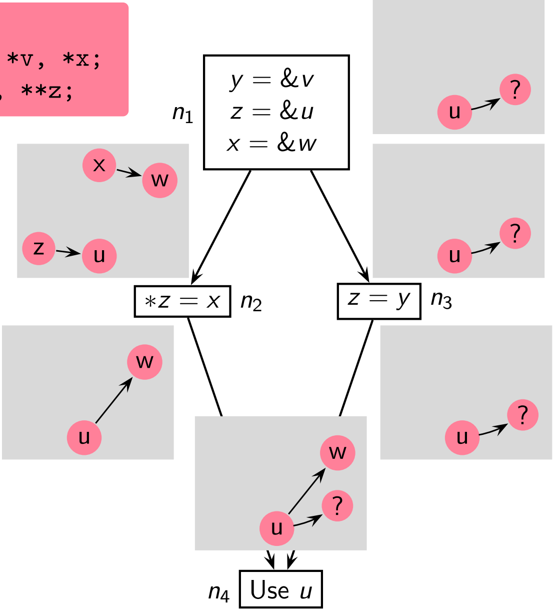
Is All This Information Useful

```
int w;
int *u, *v, *x;
int **y, **z;
```



Is All This Information Useful

```
int w;
int *u, *v, *x;
int **y, **z;
```



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]

The Role 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

The Role 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

What Generates Liveness?

- Use of a pointer in a non-assignment statement
- Indirect pointer assignment statement

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis (columns Def_n, $Kill_n$, $Pointee_n$) *Generation of strong liveness* (columns Ref_n)

	Def_n	$Kill_n$	$Pointee_n$	Ref_n: $Def_n \cap Lout_n \neq \emptyset$	Ref_n: otherwise
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	$\emptyset$	$\emptyset$	$\emptyset$		

- $Lin/Lout$: 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	$Kill_n$	$Pointee_n$	$\boxed{Ref_n}$	
				$Def_n \cap Lout_n \neq \emptyset$	otherwise
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	$\emptyset$	$\emptyset$	$\emptyset$		

Pointers that become live

Extractor Functions for LFCPA

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

	Def_n	$Kill_n$	$Pointee_n$	Ref_n	
				$\boxed{Def_n \cap Lout_n \neq \emptyset}$	*otherwise*
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	$\emptyset$	$\emptyset$	$\emptyset$		

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

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis (over columns Def_n, $Kill_n$, $Pointee_n$)

Generation of strong liveness (over column Ref_n)

	Def_n	$Kill_n$	$Pointee_n$	Ref_n	
				$Def_n \cap Lout_n \neq \emptyset$	$\boxed{otherwise}$
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	$\emptyset$	$\emptyset$	$\emptyset$		

Some pointers are unconditionally live

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis (columns Def_n, $Kill_n$, $Pointee_n$)

Generation of strong liveness (columns Ref_n)

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

x is unconditionally live

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis (columns Def_n, $Kill_n$, $Pointee_n$)

Generation of strong liveness (column Ref_n)

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

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis (Def, Kill, Pointee) — *Generation of strong liveness* (Ref)

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

y is live
if defined pointers
are live

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis (columns Def_n, $Kill_n$, $Pointee_n$) — *Generation of strong liveness* (column Ref_n)

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

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

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

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis (columns Def_n, $Kill_n$, $Pointee_n$) — *Generation of strong liveness* (columns Ref_n)

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

y is live
if defined pointers
are live

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis (columns Def_n, $Kill_n$, $Pointee_n$) — *Generation of strong liveness* (columns Ref_n)

	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	$\emptyset$	$\emptyset$	$\emptyset$		

x is unconditionally live

Extractor Functions for LFCPA

Unchanged from earlier points-to analysis (columns Def_n, $Kill_n$, $Pointee_n$) — *Generation of strong liveness* (columns Ref_n)

	Def_n	$Kill_n$	$Pointee_n$	Ref_n: $Def_n \cap Lout_n \neq \emptyset$	Ref_n: 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	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

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

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

LFCPA Data Flow Equations

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

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

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

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

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

LFCPA Data Flow Equations

$Kill_n$ defined in terms of Ain_n

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

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

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

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

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

LFCPA Data Flow Equations

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

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

Ref_n defined in terms of Ain_n and $Lout_n$

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

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

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

LFCPA Data Flow Equations

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

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

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

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

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

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

LFCPA Data Flow Equations

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

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

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

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

BI
restricted to
live pointers

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

LFCPA Data Flow Equations

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

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

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

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

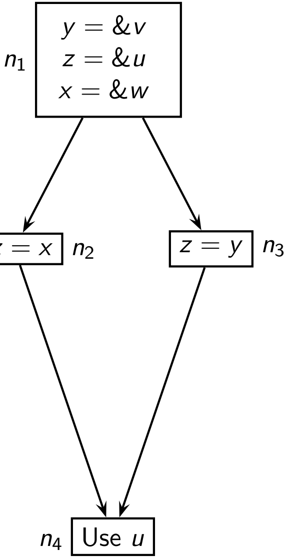
- $Lin/Lout$: set of Live pointers
- $Ain/Aout$: definitions remain unchanged except for restriction to liveness

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

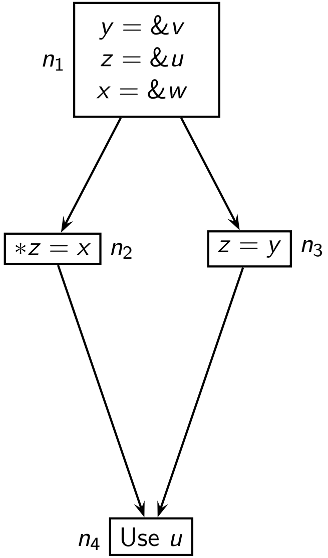
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



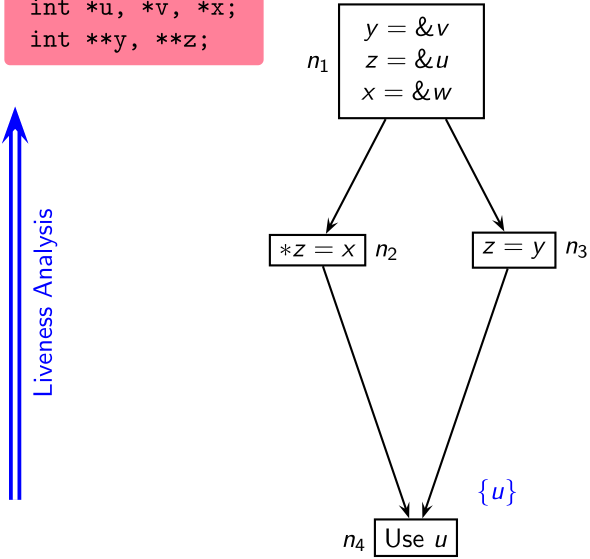
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



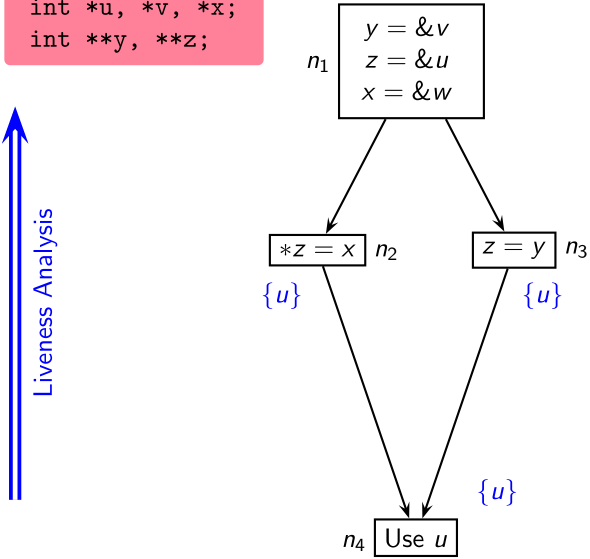
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



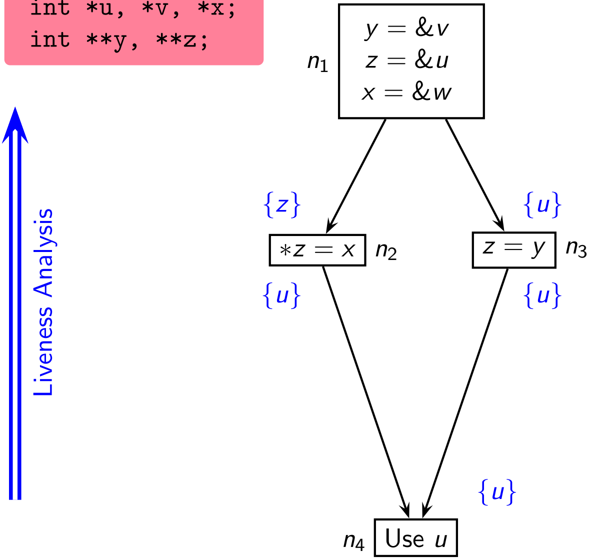
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



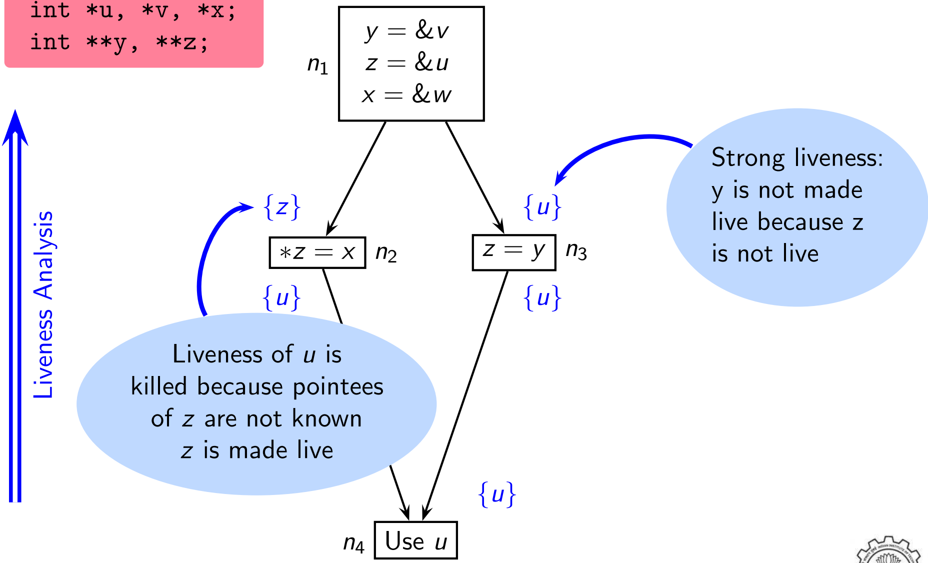
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



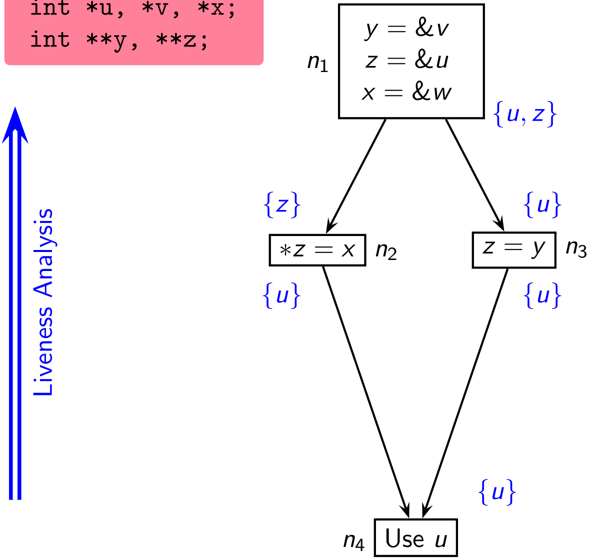
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



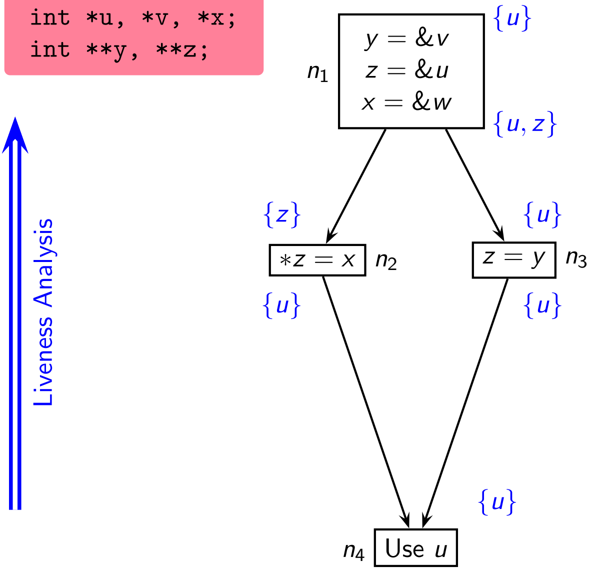
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



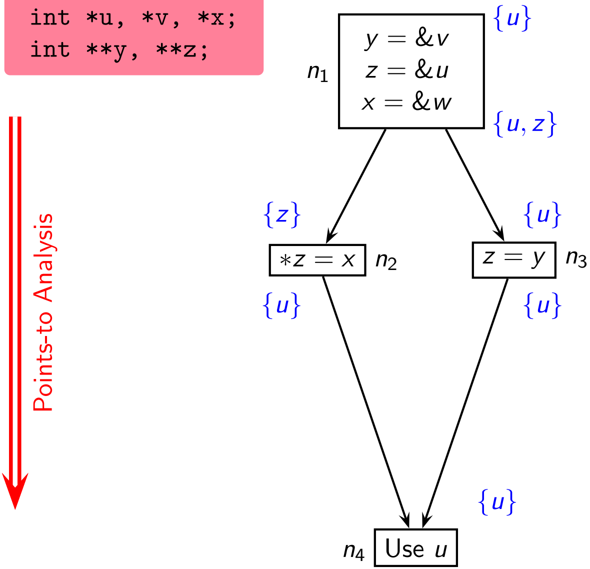
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



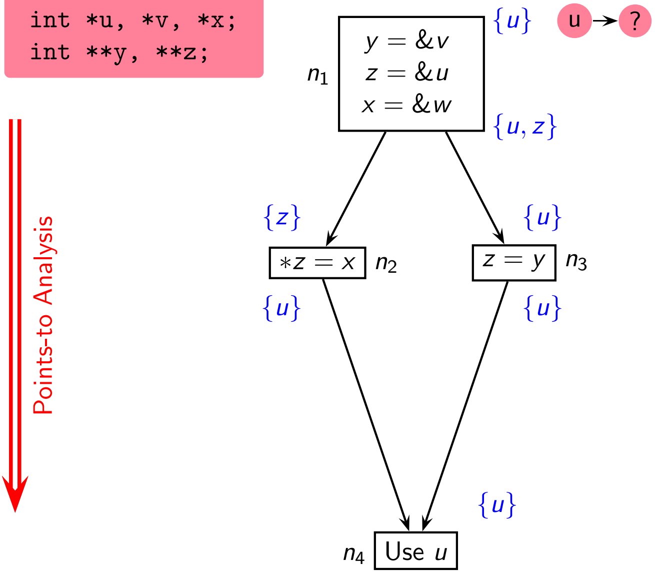
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



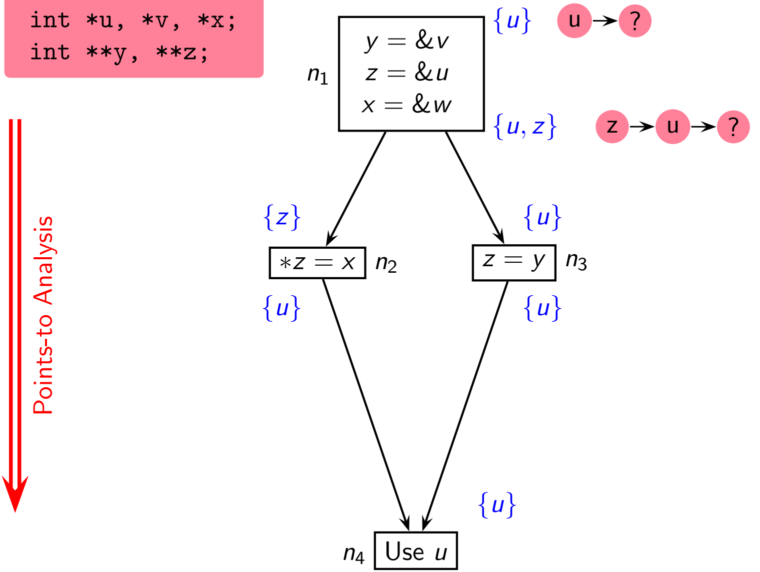
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



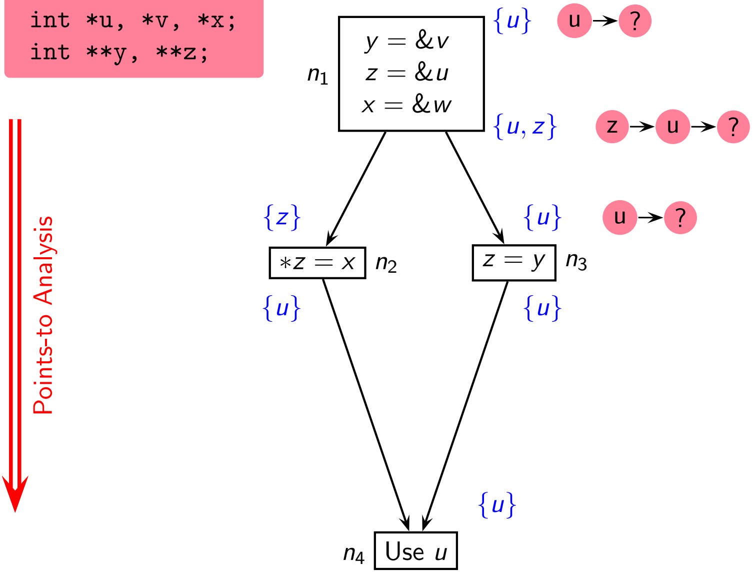
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



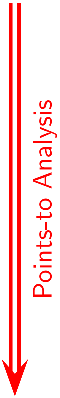
First Round of Liveness Analysis and Points-to Analysis

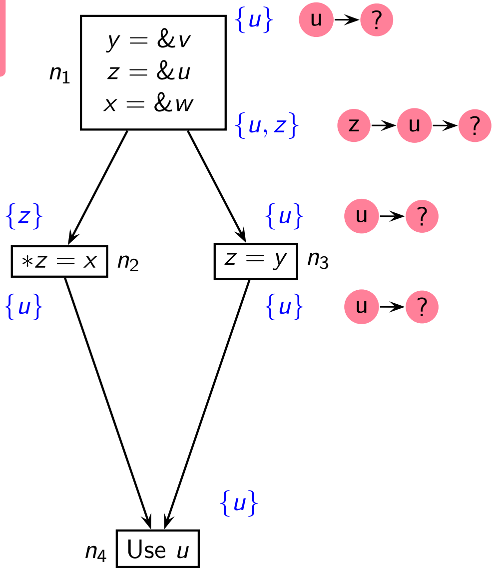
```
int w;
int *u, *v, *x;
int **y, **z;
```



First Round of Liveness Analysis and Points-to Analysis

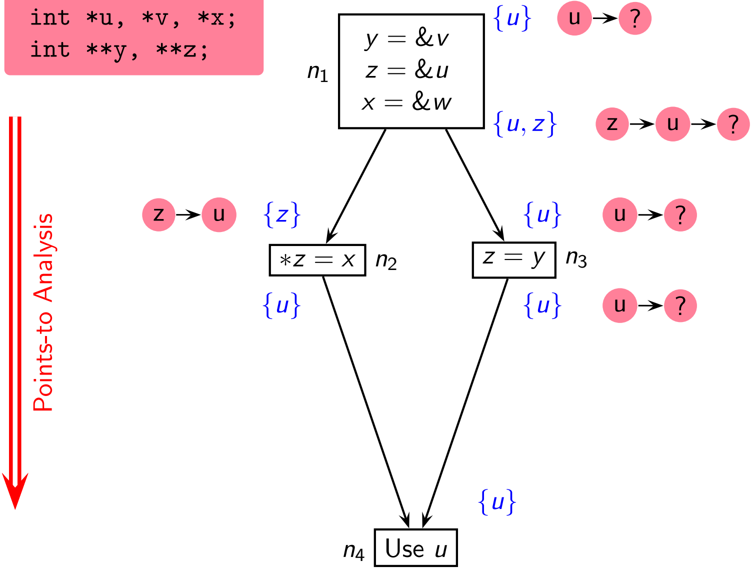
```
int w;
int *u, *v, *x;
int **y, **z;
```





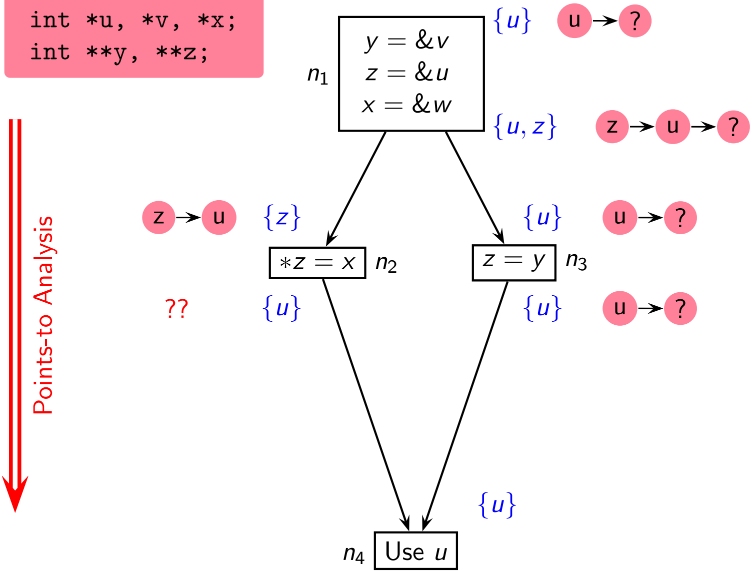
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



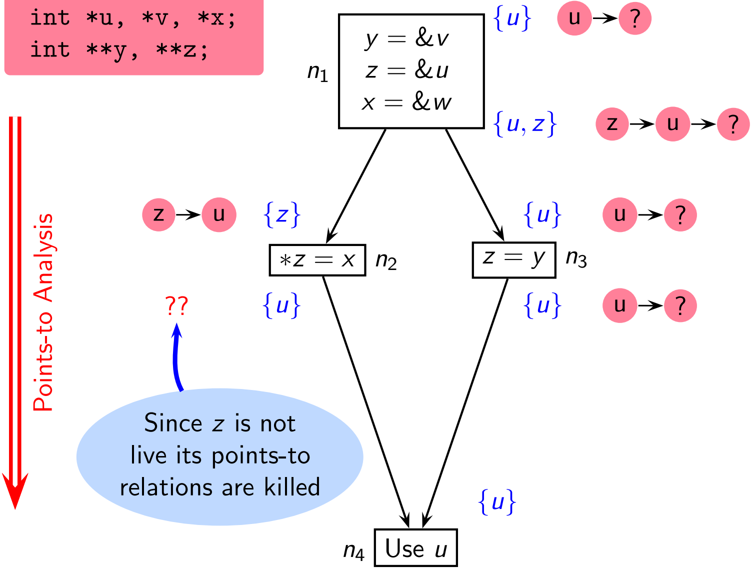
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



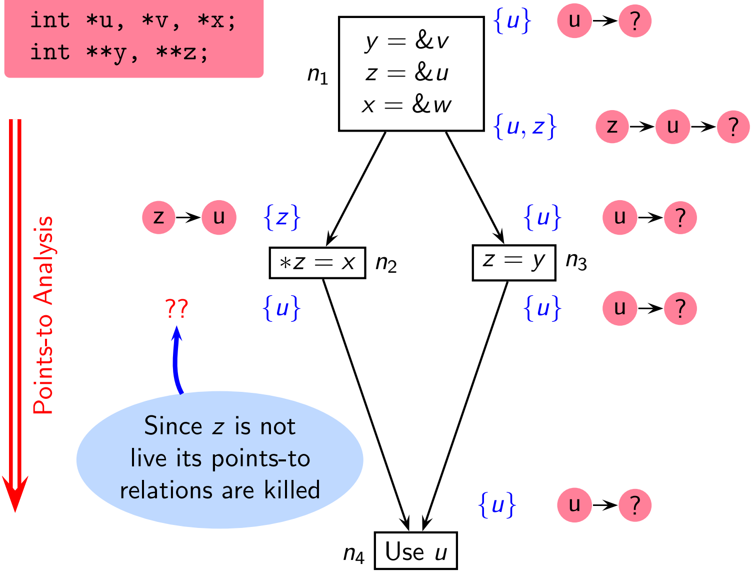
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



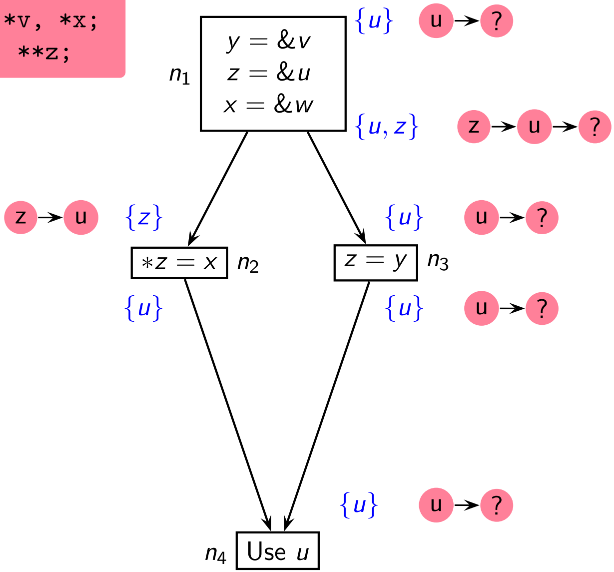
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



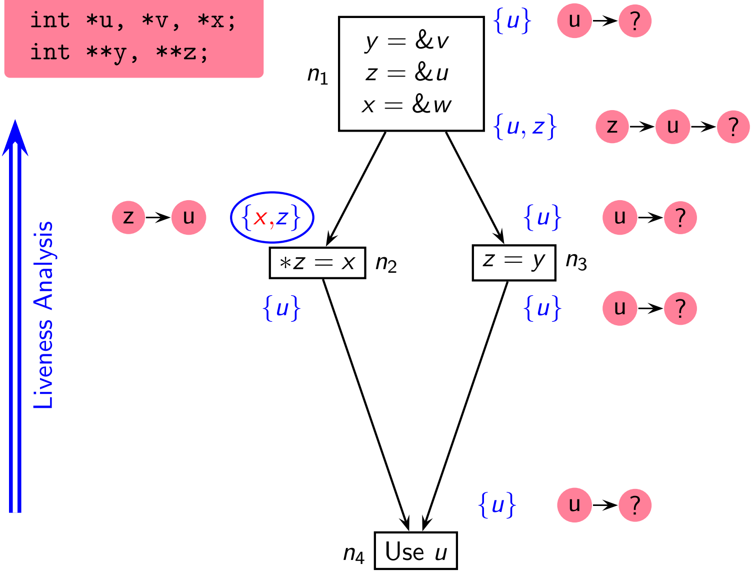
Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



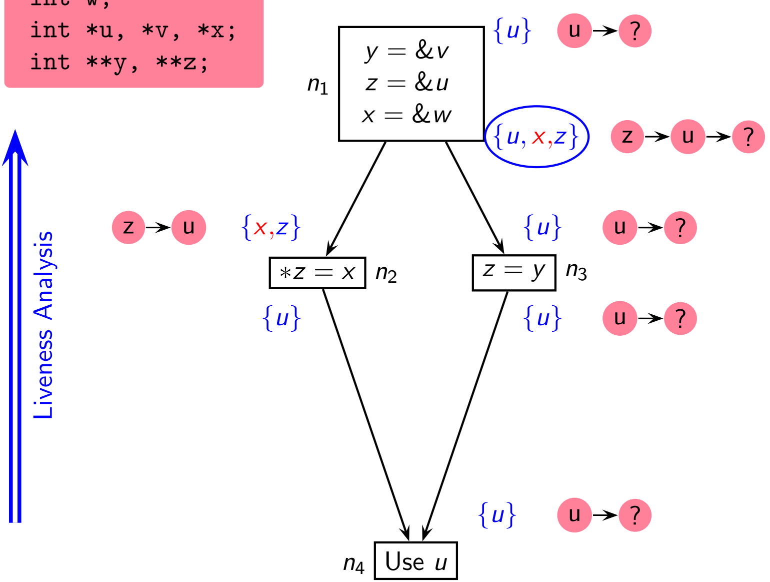
Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



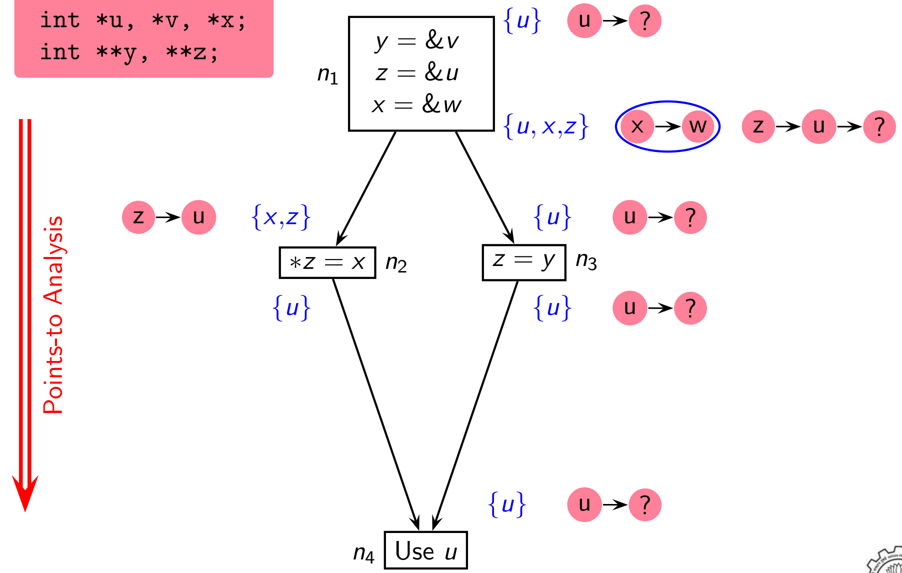
Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



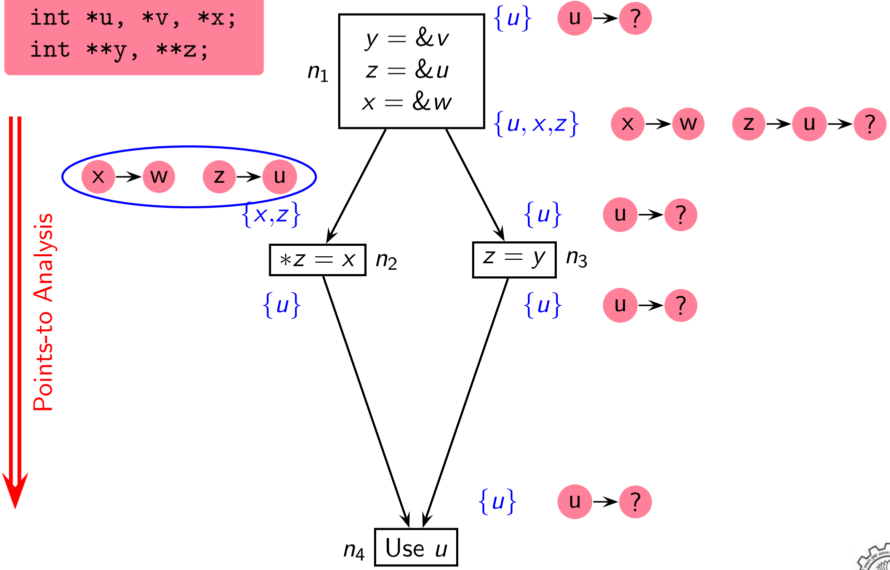
Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



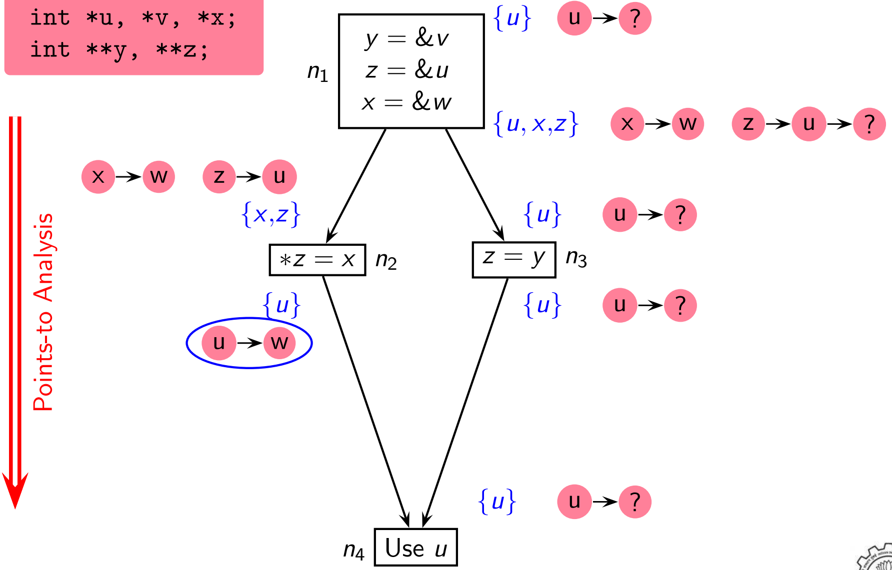
Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



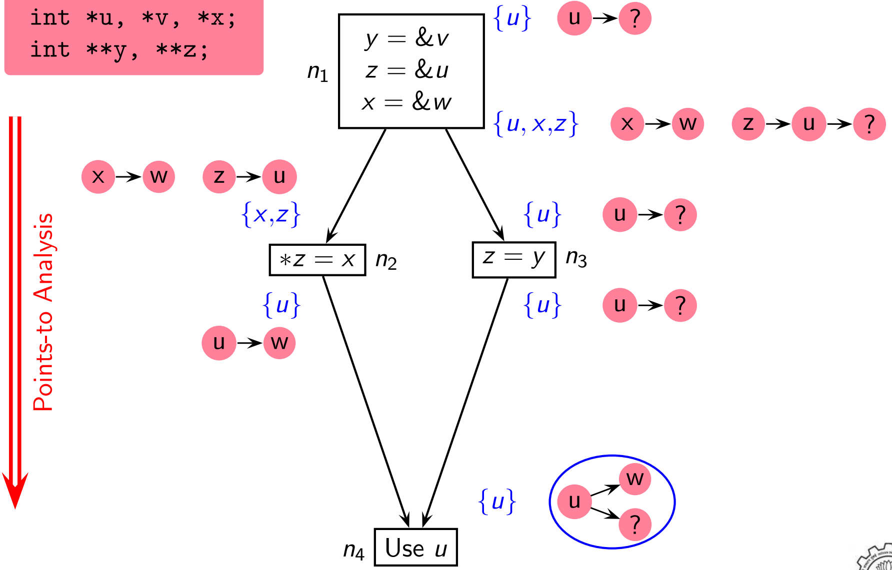
Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



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

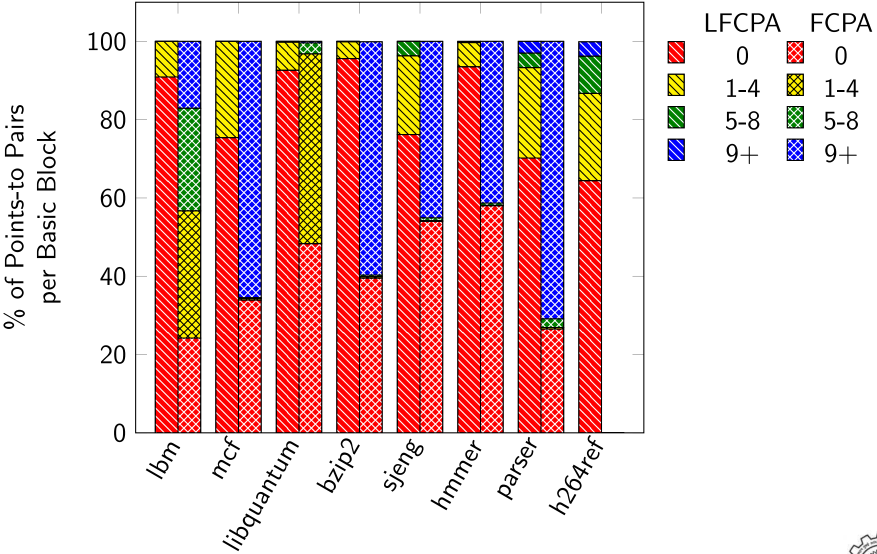
Analysis Time

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

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 ...

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

LFCPA Lessons: The Larger Perspective

exhaustive computation

computation restricted to usable information

incremental computation

demand driven computation

Maximum Computation ⟷ Minimum Computation

LFCPA Lessons: The Larger Perspective

exhaustive computation

computation restricted to usable information

incremental computation

demand driven computation

Maximum Computation ⟷ Minimum Computation

Early Computation ⟷ Late Computation

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

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?

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

← 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? Algorithm, Data Structure

LFCPA Lessons: The Larger Perspective

exhaustive computation

computation restricted to usable information

incremental computation

demand driven computation

Wha

Maximum Computation

When

Early Computation

Avoid computing some values because

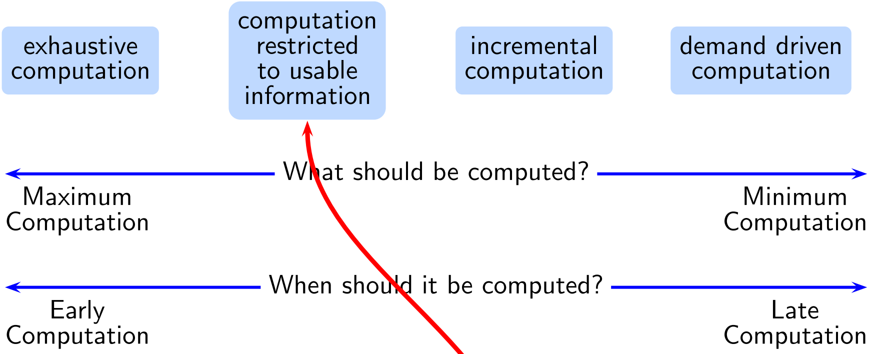
- they have been computed before, or
- they can just be "adjusted", or
- they are equivalent to some other values

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



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

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

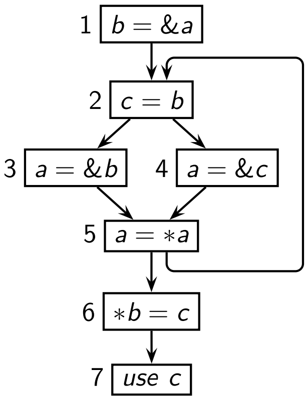
Do not compute what you don't need!

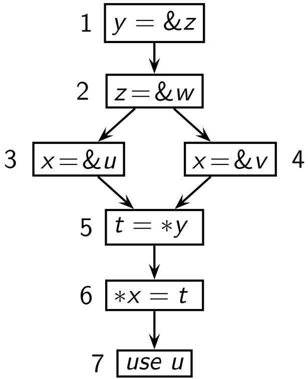
Who defines what is needed?

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
- Defining Points-to Analysis
- 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
$2^{\mathbf{P}\times V}, 2^{\mathbf{P}}$

Named locations

Variables V, 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 V, Pointers **P**, Allocation Sites H, Fields F, pF, npF, Offsets C

Generalization for Heap and Structures

- Grammar.

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

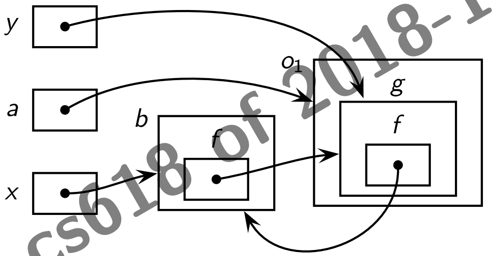
where α is a pointer expression, x is a variable, and f is a field

- Memory model: Named memory locations. No numeric addresses

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

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

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

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

Defining Extractor Functions

- Pointer assignment statement $lhs_n = rhs_n$

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

- *Use* α statement

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

- Any other statement

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

Extensions for Handling Arrays and Pointer Arithmetic

- Grammar.

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

- 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

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

Extending L-Value Computation to Arrays and Pointer Arithmetic

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

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

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

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