Pointer Analysis

Uday Khedker

(www.cse.iitb.ac.in/˜uday)

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

Dec 2019

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)

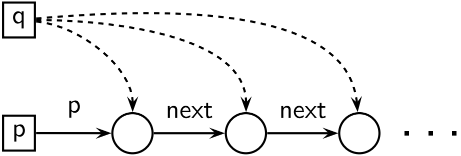
Apart from the above book, some slides are based on the material from the following book

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

An Outline of Pointer Analysis Coverage

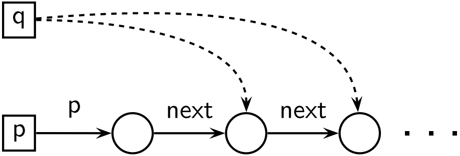
- The larger perspective
- IR for Points-to Analysis
- Flow-Insensitive Points-to Analysis
- Flow-Sensitive Points-to Analysis

Code Optimization In Presence of Pointers (1)

Program	*Memory graph at statement 5*
1. q = p; 2. while (. . .) { 3. q = q→next; 4. } 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?

Code Optimization In Presence of Pointers (1)

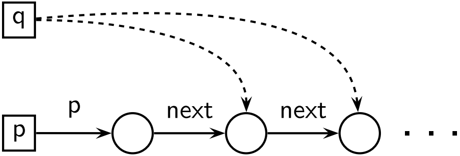
Program	*Memory graph at statement 5*
1. q = p; 2. do { 3. q = q→next; 4. } while (. . .) 5. p→data = r1; 6. print (q→data); 7. p→data = r2;	

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

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;
```



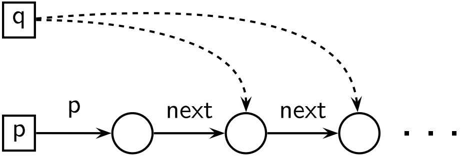
- Is p→data live at the exit of line 5? Can we delete line 5?
- We cannot delete line 5 if p and q can be possibly aliased (while loop or do-while loop with a circular list)

Code Optimization In Presence of Pointers (1)

Program

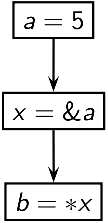
```
1.   q = p;
2.   do {
3.        q = q→next;
4.   }   while (. . . )
5.   p→data = r1;
6.   print (q→data);
7.   p→data = r2;
```

Memory graph at statement 5



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



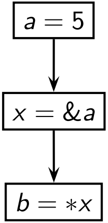
$a = 5$

$x = \&a$

$b = *x$

Original Program

Code Optimization In Presence of Pointers (2)

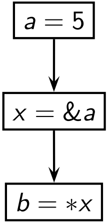


$a = 5$

$x = \&a$

$b = *x$

Original Program



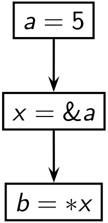
$a = 5$

$x = \&a$

$b = *x$

Constant Propagation
without aliasing

Code Optimization In Presence of Pointers (2)

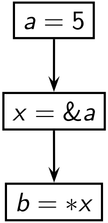


$a = 5$

$x = \&a$

$b = *x$

Original Program

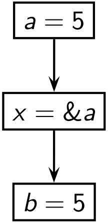


$a = 5$

$x = \&a$

$b = *x$

Constant Propagation
without aliasing



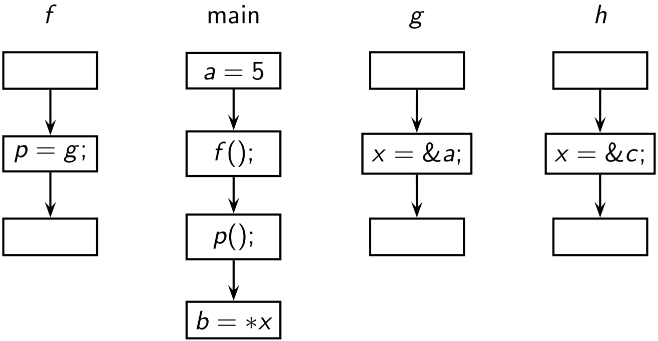
$a = 5$

$x = \&a$

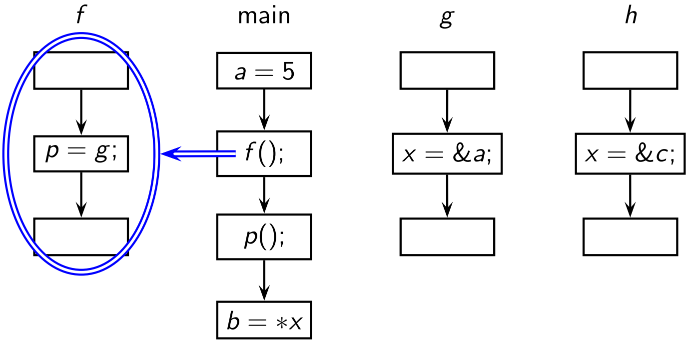
$b = 5$

Constant Propagation
with aliasing

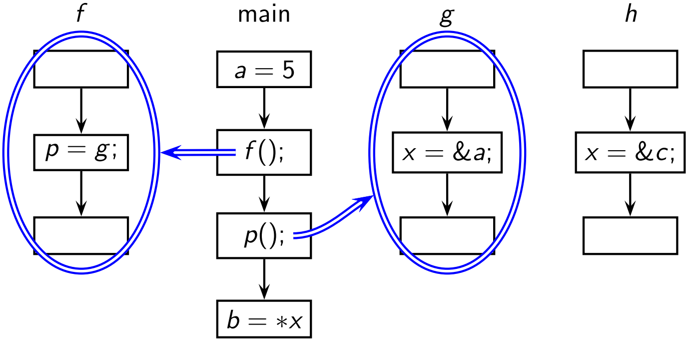
Code Optimization In Presence of Pointers (3)



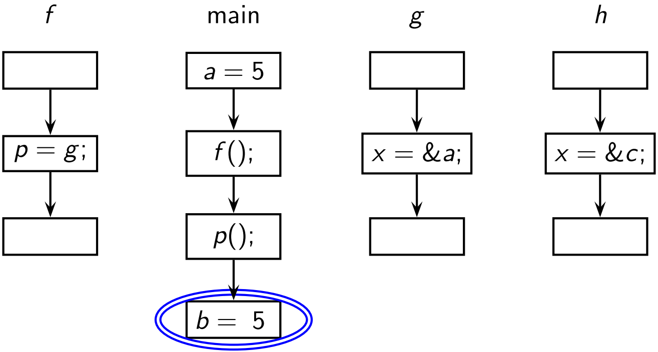
Code Optimization In Presence of Pointers (3)



Code Optimization In Presence of Pointers (3)



Code Optimization In Presence of Pointers (3)

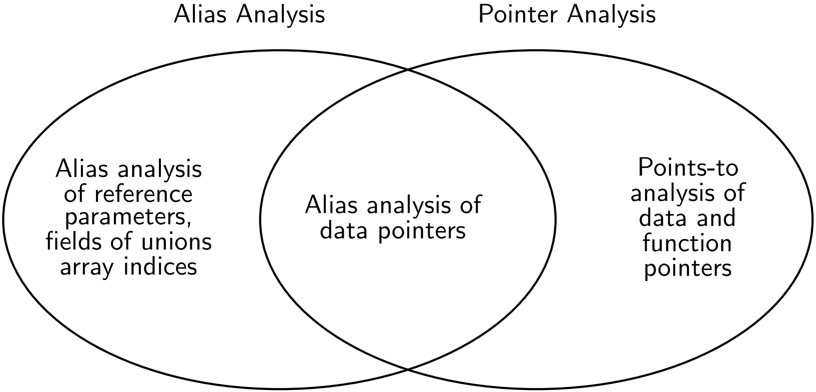


Pointer Analysis

- 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 $x \to f()$
- 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



Alias Analysis

Pointer Analysis

Alias analysis of reference parameters, fields of unions array indices

Alias analysis of data pointers

Points-to analysis of data and function pointers

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
  - 2019 . . ☹

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: Precision versus Scalability

- Ideally, an analysis should be
  - Sound
  - Precise
  - Scalable

Pointer Analysis: Precision versus Scalability

- Ideally, an analysis should be
  - Sound
  - Precise
  - Scalable

Common belief

- Precision and scalability cannot be achieved together for exhaustive analysis

Common Practice

- Trade off precision using approximations

Pointer Analysis: Precision versus Scalability

- Ideally, an analysis should be
    - Sound
    - Precise
    - Scalable
- The main factors enhancing the precision of an exhaustive (as against a demand-driven) analysis are
    - Flow sensitivity
    - Context sensitivity
    - Field sensitivity

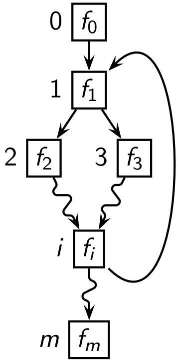
Demand-Driven Analysis Vs. Exhaustive Analysis

- Exhaustive. Compute all possible information
- Demand-Driven. Compute only the requested information (by a client)

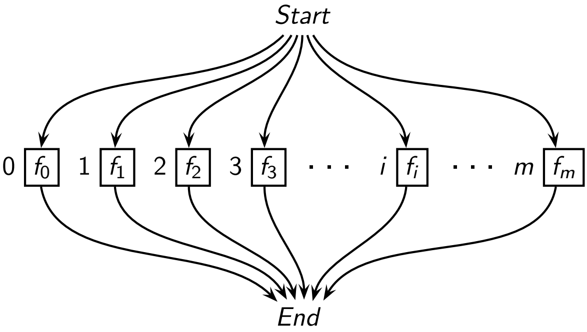
Different from incremental analysis which also computes only some information but it updates the earlier computed solution

Flow Sensitivity Vs. Flow Insensitivity

Flow Sensitive

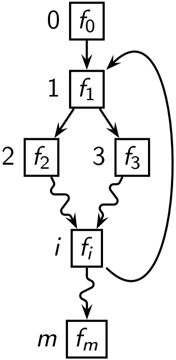


Flow Insensitive

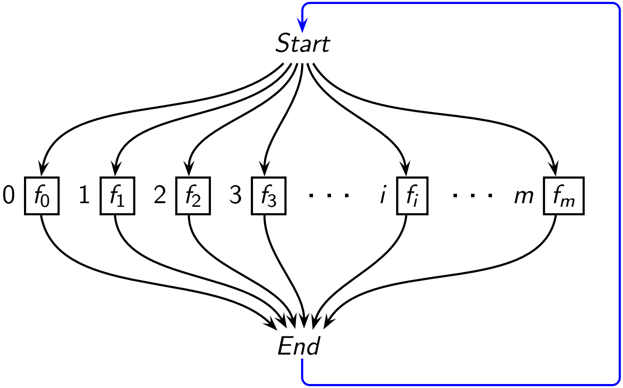


Flow Sensitivity Vs. Flow Insensitivity

Flow Sensitive



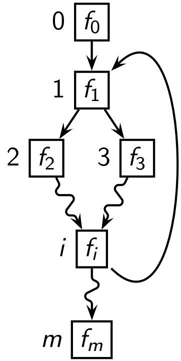
Flow Insensitive



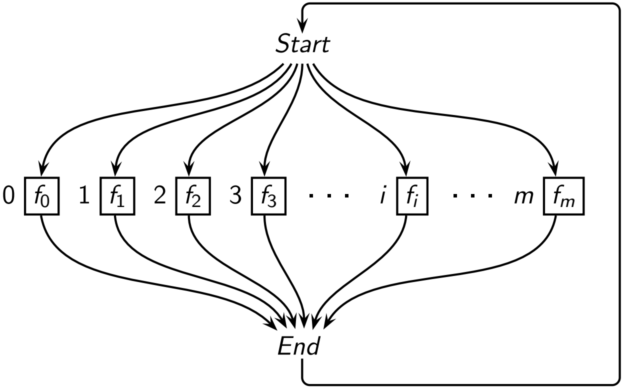
Assumption: Statements can be executed in any order

Flow Sensitivity Vs. Flow Insensitivity

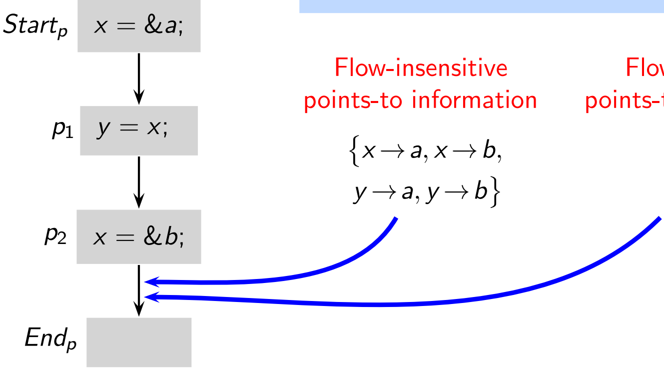
Flow Sensitive



Flow Insensitive



Flow Sensitivity Vs. Flow Insensitivity



Flow-insensitive analysis is less precise than a flow-sensitive analysis

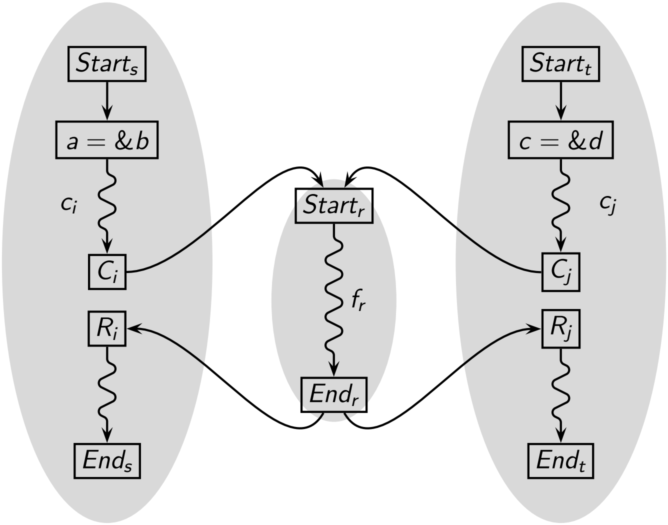
Flow-insensitive points-to information

$\{x \rightarrow a, x \rightarrow b,$
$y \rightarrow a, y \rightarrow b\}$

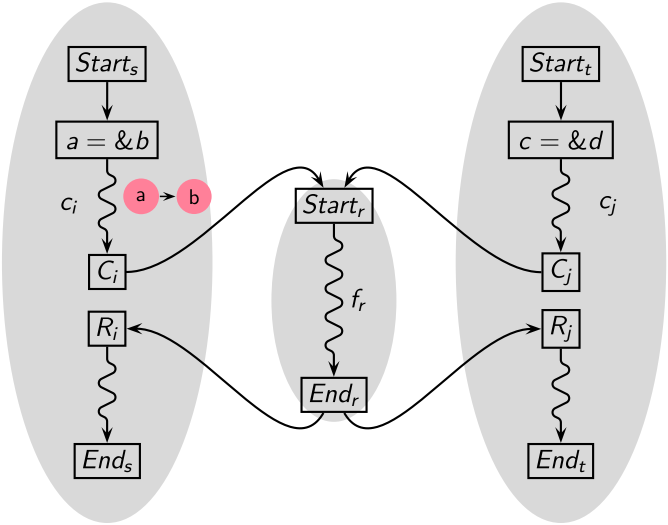
Flow-sensitive points-to information

$\{x \rightarrow b,$
$y \rightarrow a\}$

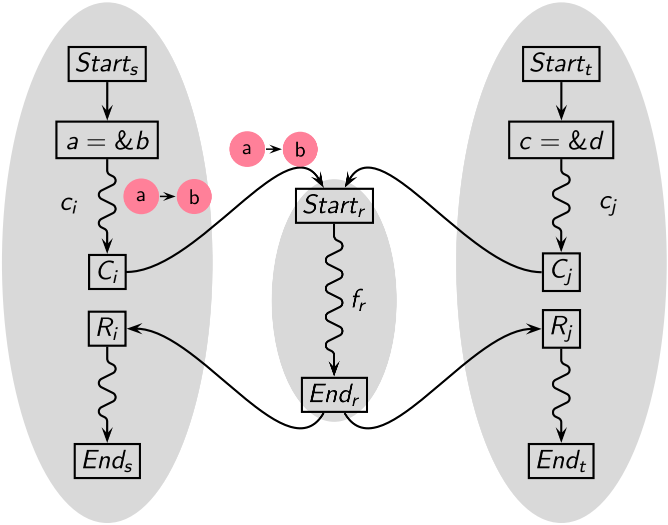
Context Sensitivity Vs. Context Insensitivity



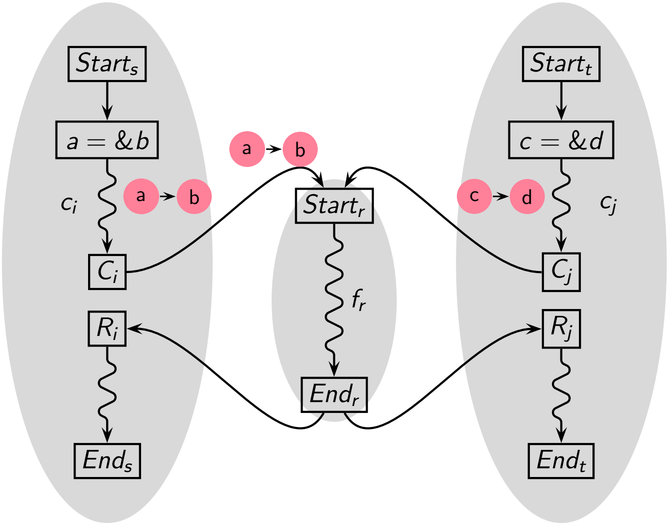
Context Sensitivity Vs. Context Insensitivity



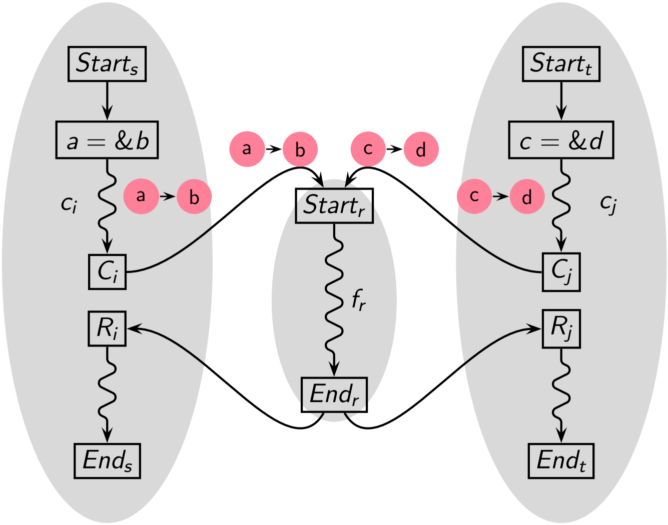
Context Sensitivity Vs. Context Insensitivity



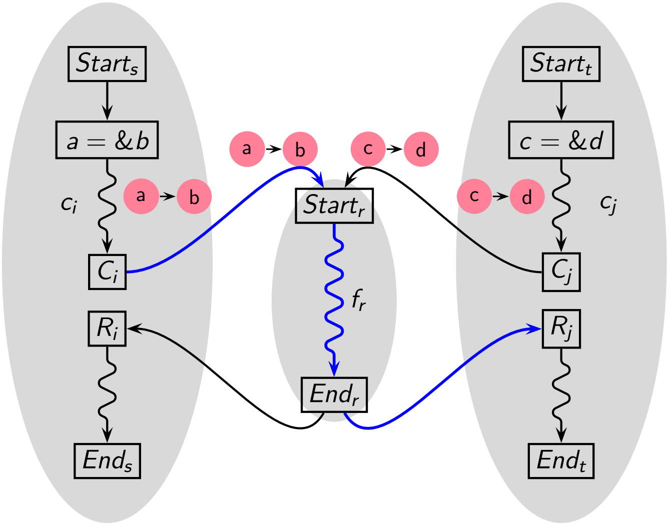
Context Sensitivity Vs. Context Insensitivity



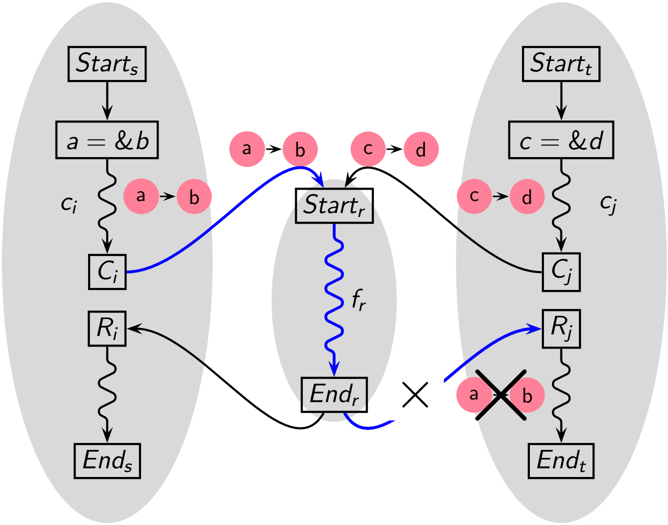
Context Sensitivity Vs. Context Insensitivity



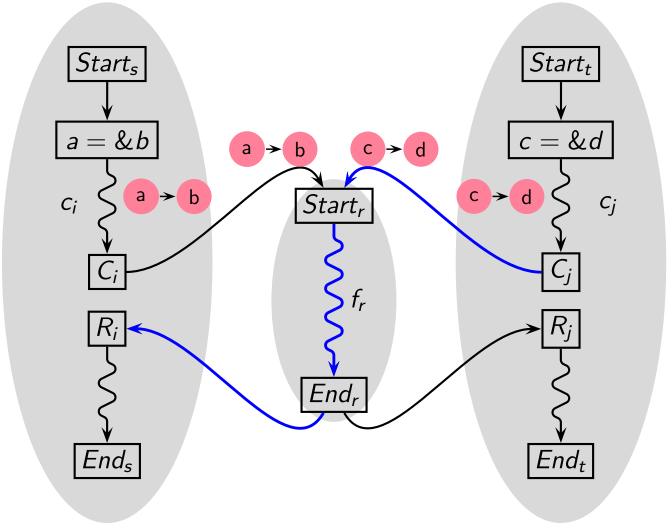
Context Sensitivity Vs. Context Insensitivity



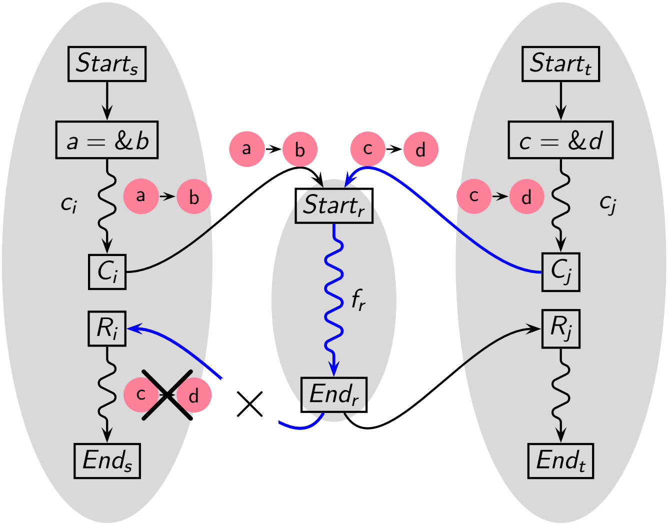
Context Sensitivity Vs. Context Insensitivity



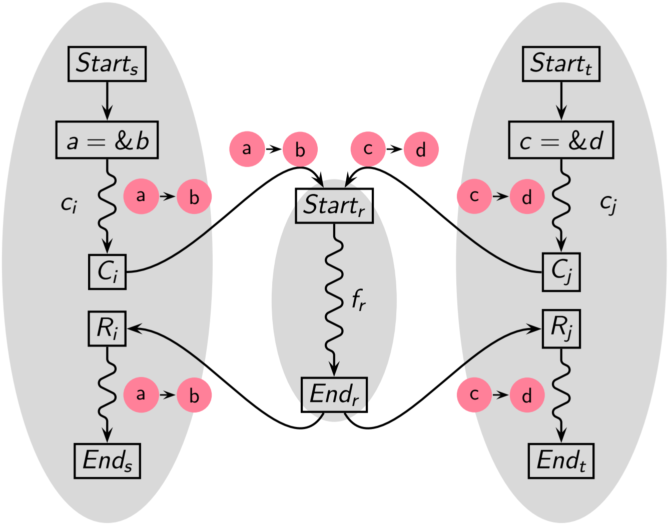
Context Sensitivity Vs. Context Insensitivity



Context Sensitivity Vs. Context Insensitivity

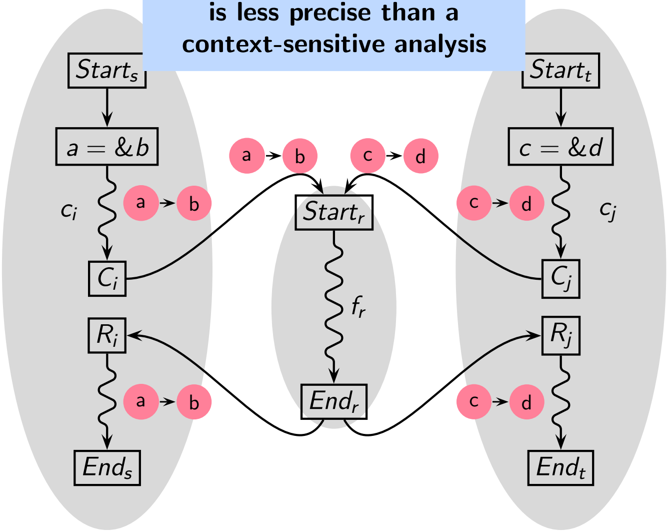


Context Sensitivity Vs. Context Insensitivity

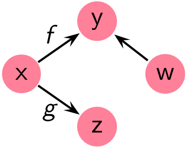


Context Sensitivity Vs. Context Insensitivity

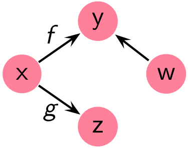
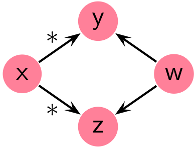
Context-insensitive analysis is less precise than a context-sensitive analysis



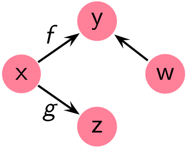
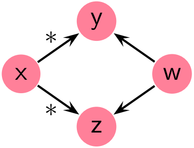
Field Sensitivity Vs. Field Insensitivity

Program	Field-sensitive points-to graph	Field-insensitive points-to graph
$x \rightarrow f = \&y$ $x \rightarrow g = \&z$ $w = x \rightarrow f$		

Field Sensitivity Vs. Field Insensitivity

Program	Field-sensitive points-to graph	Field-insensitive points-to graph
$x \rightarrow f = \&y$ $x \rightarrow g = \&z$ $w = x \rightarrow f$		

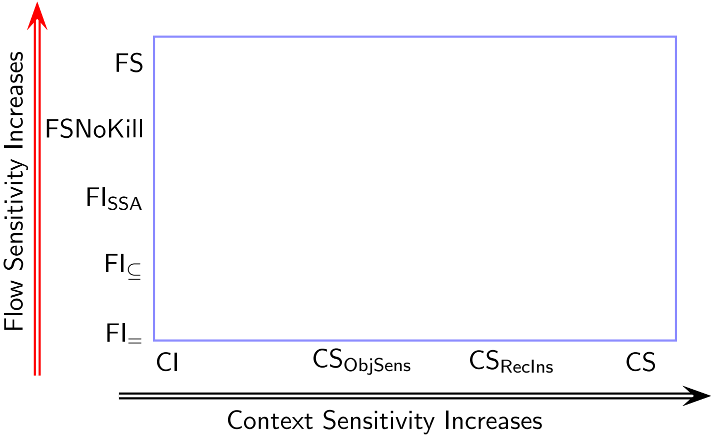
Field Sensitivity Vs. Field Insensitivity

Program	Field-sensitive points-to graph	Field-insensitive points-to graph
$x \rightarrow f = \&y$ $x \rightarrow g = \&z$ $w = x \rightarrow f$		

Field-insensitive analysis is less precise than a field-sensitive analysis

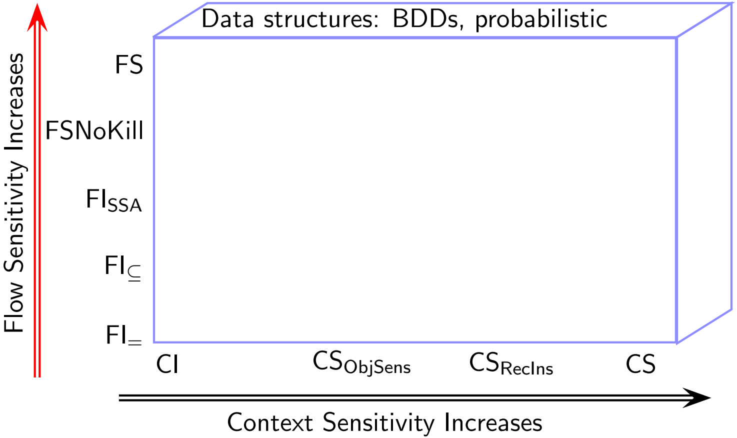
Pointer Analysis: An Engineer's Landscape

Pointer analysis is a fertile ground for research because the factors that enhance the precision of points-to analysis (flow, context, and field sensitivity), hamper scalability



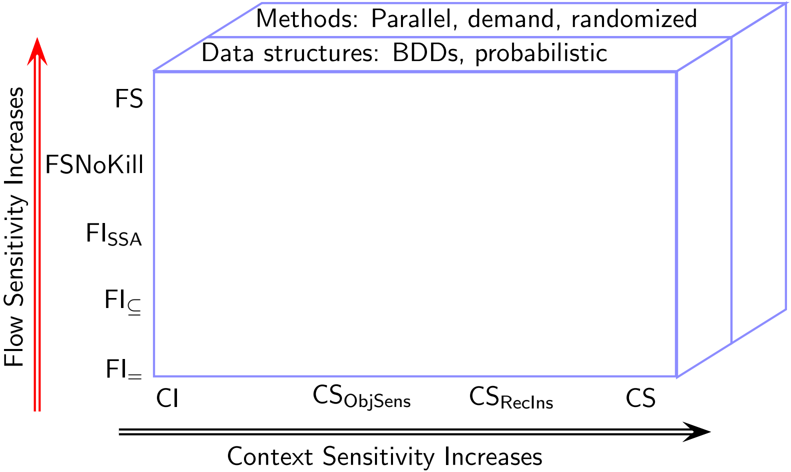
Pointer Analysis: An Engineer's Landscape

Pointer analysis is a fertile ground for research because the factors that enhance the precision of points-to analysis (flow, context, and field sensitivity), hamper scalability



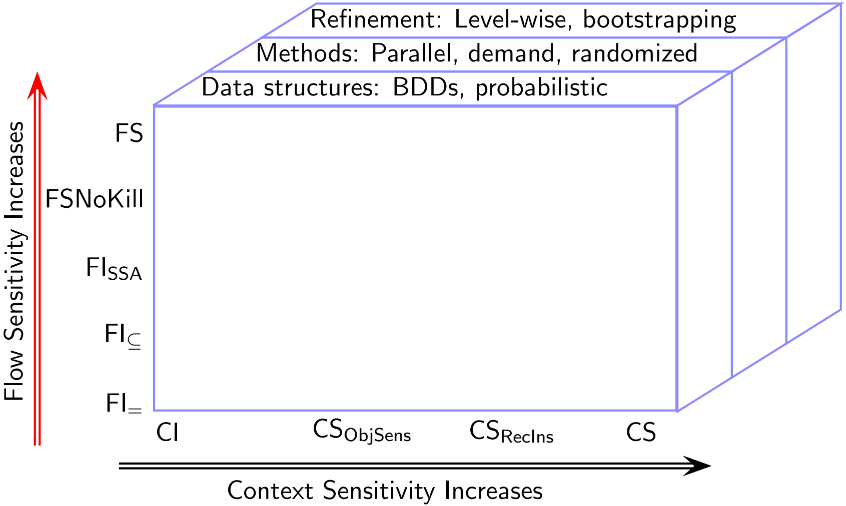
Pointer Analysis: An Engineer's Landscape

Pointer analysis is a fertile ground for research because the factors that enhance the precision of points-to analysis (flow, context, and field sensitivity), hamper scalability



Pointer Analysis: An Engineer's Landscape

Pointer analysis is a fertile ground for research because the factors that enhance the precision of points-to analysis (flow, context, and field sensitivity), hamper scalability



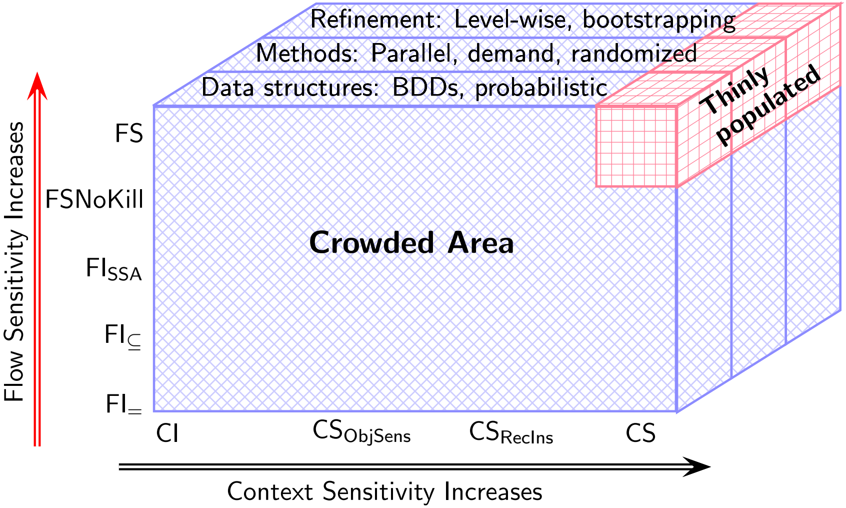
Refinement: Level-wise, bootstrapping

Methods: Parallel, demand, randomized

Data structures: BDDs, probabilistic

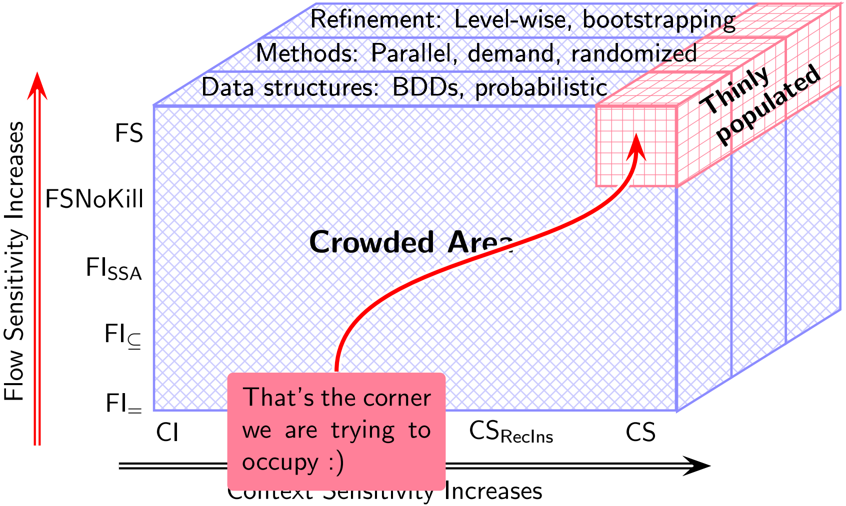
Pointer Analysis: An Engineer's Landscape

Pointer analysis is a fertile ground for research because the factors that enhance the precision of points-to analysis (flow, context, and field sensitivity), hamper scalability



Pointer Analysis: An Engineer's Landscape

Pointer analysis is a fertile ground for research because the factors that enhance the precision of points-to analysis (flow, context, and field sensitivity), hamper scalability



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

An Outline of Pointer Analysis Coverage

- The larger perspective
- IR for Points-to Analysis Next Topic
- Flow-Insensitive Points-to Analysis
- Flow-Sensitive Points-to Analysis

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 t
$(** y = a)$	$t = *y$ $*t = a$	$t = *y$ 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

An Outline of Pointer Analysis Coverage

- The larger perspective
- IR for Points-to Analysis
- Flow-Insensitive Points-to Analysis Next Topic
- Flow-Sensitive Points-to Analysis

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.

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

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?

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
- $\mathit{Unify}(x, y)$ unifies locations x and y
    - x and y are treated as equivalent locations
    - the pointees of the unified locations are also unified transitively
- $\mathit{UnifyPTS}(x, y)$ 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

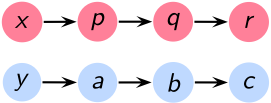
Andersen's and Steensgaard's Points-to Analysis

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

Andersen's and Steensgaard's Points-to Analysis

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

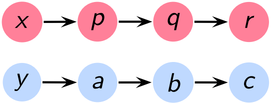
Points-to graph before the assignment



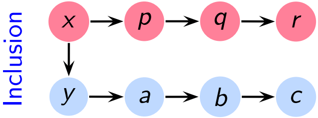
Andersen's and Steensgaard's Points-to Analysis

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

Points-to graph before the assignment



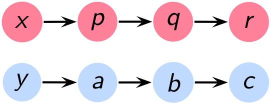
Andersen's graph after the assignment



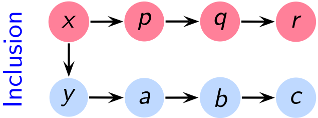
Andersen's and Steensgaard's Points-to Analysis

Statement	Andersen's Points-to Sets	Steensgaard's Points-to Sets
$\boxed{x = \&y}$	$\boxed{P_x \supseteq \{y\}}$	$\boxed{\begin{array}{l} P_x \supseteq \{y\} \\ \forall z \in P_x.\ Unify(y, z) \end{array}}$
$x = y$	$P_x \supseteq P_y$	$UnifyPTS(x, y)$
$x = *y$	$P_x \supseteq P_z.\ \forall z \in P_y$	$\forall z \in P_y.\ UnifyPTS(x, z)$
$*x = y$	$\forall z \in P_x.\ P_z \supseteq P_y$	$\forall z \in P_x.\ UnifyPTS(y, z)$

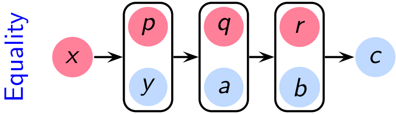
Points-to graph before the assignment



Andersen's graph after the assignment



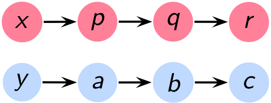
Steensgaard's graph after the assignment



Andersen's and Steensgaard's Points-to Analysis

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

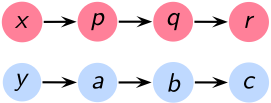
Points-to graph before the assignment



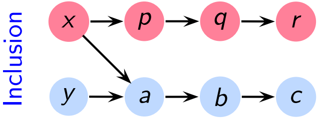
Andersen's and Steensgaard's Points-to Analysis

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

Points-to graph before the assignment



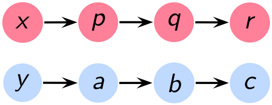
Andersen's graph after the assignment



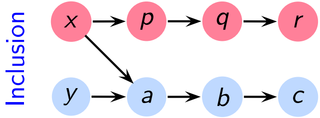
Andersen's and Steensgaard's Points-to Analysis

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

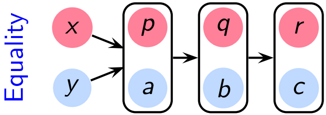
Points-to graph before the assignment



Andersen's graph after the assignment



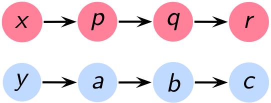
Steensgaard's graph after the assignment



Andersen's and Steensgaard's Points-to Analysis

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

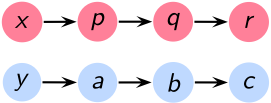
Points-to graph before the assignment



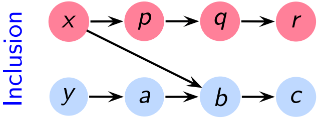
Andersen's and Steensgaard's Points-to Analysis

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

Points-to graph before the assignment



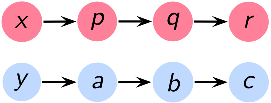
Andersen's graph after the assignment



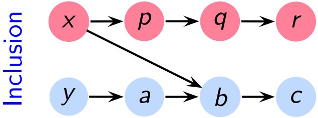
Andersen's and Steensgaard's Points-to Analysis

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

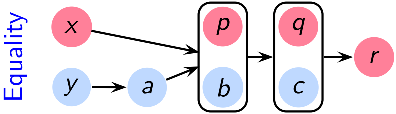
Points-to graph before the assignment



Andersen's graph after the assignment



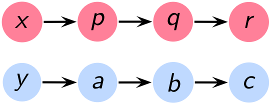
Steensgaard's graph after the assignment



Andersen's and Steensgaard's Points-to Analysis

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

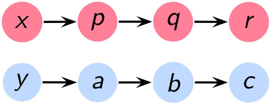
Points-to graph before the assignment



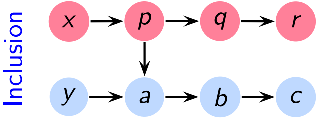
Andersen's and Steensgaard's Points-to Analysis

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

Points-to graph before the assignment



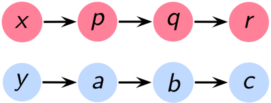
Andersen's graph after the assignment



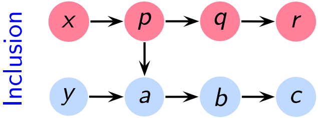
Andersen's and Steensgaard's Points-to Analysis

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

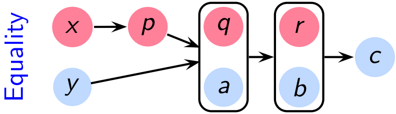
Points-to graph before the assignment



Andersen's graph after the assignment



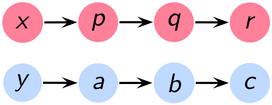
Steensgaard's graph after the assignment



Andersen's and Steensgaard's Points-to Analysis

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

Points-to graph before the assignment



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

Program

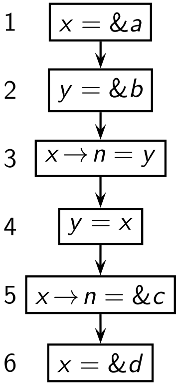
1 $x = \&a$
2 $y = \&b$
3 $x \rightarrow n = y$
4 $y = x$
5 $x \rightarrow n = \&c$
6 $x = \&d$

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

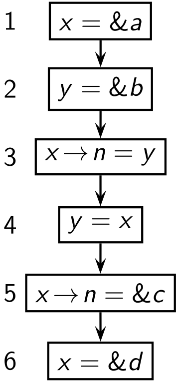
Program



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	$P_x \supseteq \{d\}$

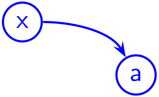
Example of Inclusion Based (aka Andersen's) Points-to Analysis

Program

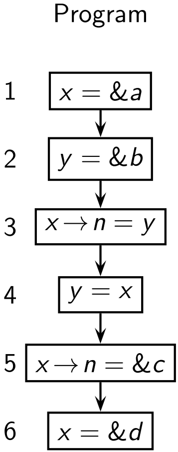


Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$

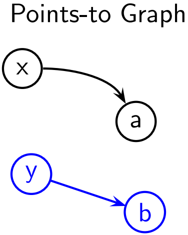
Points-to Graph



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

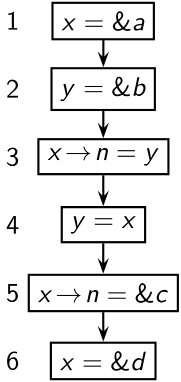


Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$



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

Program



1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

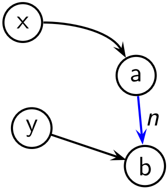
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

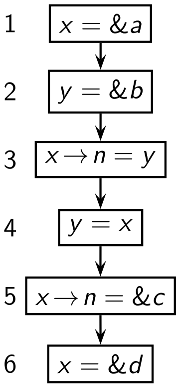
Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$

Points-to Graph



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

Program



1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

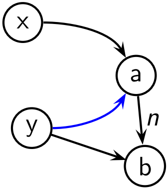
4 $y = x$

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 $x \rightarrow n = y$

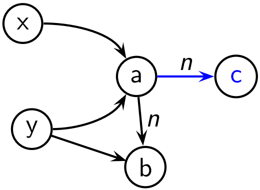
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

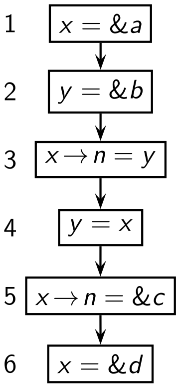
Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$

Points-to Graph



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

Program



1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

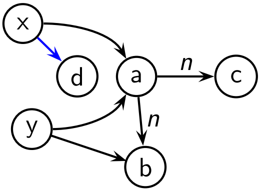
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

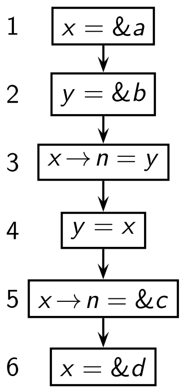
Node	Constraint
1	$P_x \supseteq \{a\}$
2	$P_y \supseteq \{b\}$
3	$\forall z \in P_x, P_{z.n} \supseteq P_y$
4	$P_y \supseteq P_x$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$
6	$P_x \supseteq \{d\}$

Points-to Graph



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

Program


1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

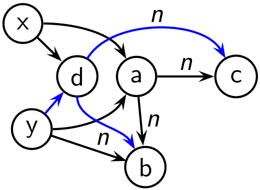
4 $y = x$

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	$P_x \supseteq \{d\}$

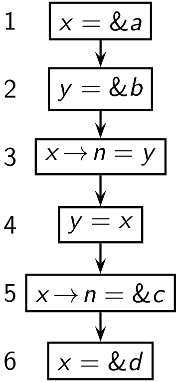
Points-to Graph



- 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

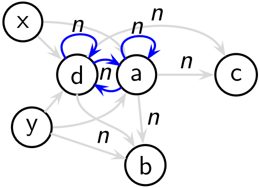
Example of Inclusion Based (aka Andersen's) Points-to Analysis

Program



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	$P_x \supseteq \{d\}$

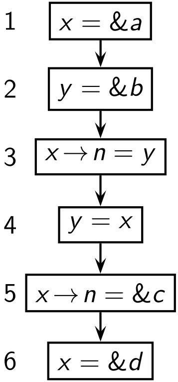
Points-to Graph



- 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 $x \rightarrow n = y$

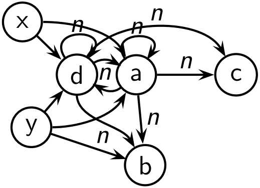
4 $y = x$

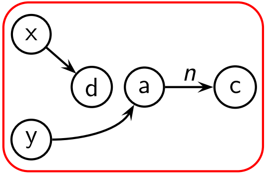
5 $x \rightarrow n = \&c$

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

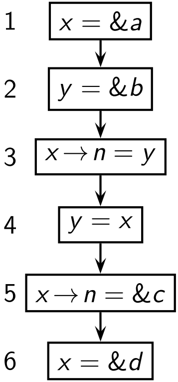




- Actual graph after statement 6 (red box on the right) is much simpler with many edges killed
- y does not point to d any time in the execution

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

Program



1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

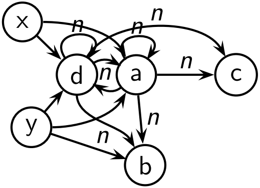
4 $y = x$

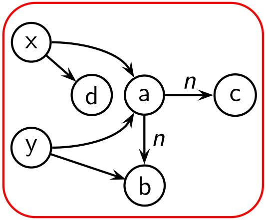
5 $x \rightarrow n = \&c$

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

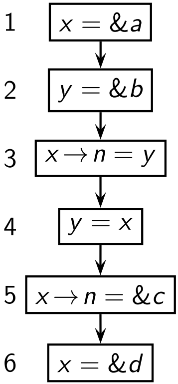




- A union of all graphs at each program point
- *y* does not point to *d* any time in the execution

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

Program



1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

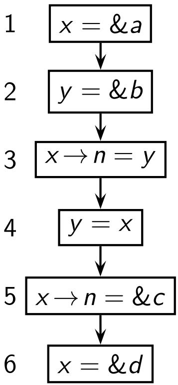
4 $y = x$

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 $x \rightarrow n = y$

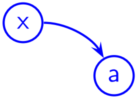
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

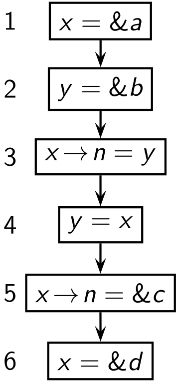
Node	Constraint
1	$P_x \supseteq \{a\}$ $\forall z \in P_x, Unify(a, z)$
2	$P_y \supseteq \{b\}$ $\forall z \in P_y, Unify(b, z)$
3	$\forall z \in P_x, UnifyPTS(y, z.n)$
4	$UnifyPTS(x, y)$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$ $\forall w \in P_{z.n}, Unify(w, c)$
6	$P_x \supseteq \{d\}$ $\forall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

Points-to Graph



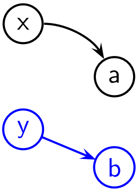
Example of Equality Based (aka Steensgaard's) Points-to Analysis

Program



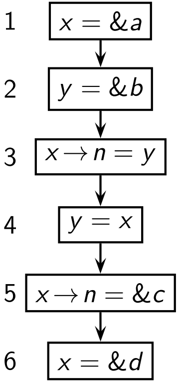
Node	Constraint
1	$P_x \supseteq \{a\}$ $\forall z \in P_x, Unify(a, z)$
2	$P_y \supseteq \{b\}$ $\forall z \in P_y, Unify(b, z)$
3	$\forall z \in P_x, UnifyPTS(y, z.n)$
4	$UnifyPTS(x, y)$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$ $\forall w \in P_{z.n}, Unify(w, c)$
6	$P_x \supseteq \{d\}$ $\forall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

Points-to Graph



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

Program



1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

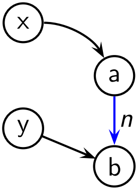
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

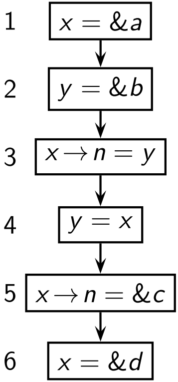
Node	Constraint
1	$P_x \supseteq \{a\}$ $\forall z \in P_x, Unify(a, z)$
2	$P_y \supseteq \{b\}$ $\forall z \in P_y, Unify(b, z)$
3	$\forall z \in P_x, UnifyPTS(y, z.n)$
4	$UnifyPTS(x, y)$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$ $\forall w \in P_{z.n}, Unify(w, c)$
6	$P_x \supseteq \{d\}$ $\forall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

Points-to Graph



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

Program



1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

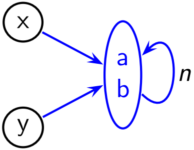
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

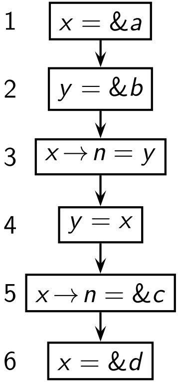
Node	Constraint
1	$P_x \supseteq \{a\}$ $\forall z \in P_x, Unify(a, z)$
2	$P_y \supseteq \{b\}$ $\forall z \in P_y, Unify(b, z)$
3	$\forall z \in P_x, UnifyPTS(y, z.n)$
4	$UnifyPTS(x, y)$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$ $\forall w \in P_{z.n}, Unify(w, c)$
6	$P_x \supseteq \{d\}$ $\forall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

Points-to Graph



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

Program



1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

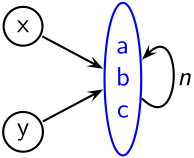
4 $y = x$

5 $x \rightarrow n = \&c$

6 $x = \&d$

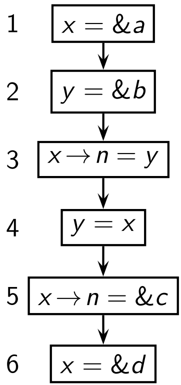
Node	Constraint
1	$P_x \supseteq \{a\}$ $\forall z \in P_x, Unify(a, z)$
2	$P_y \supseteq \{b\}$ $\forall z \in P_y, Unify(b, z)$
3	$\forall z \in P_x, UnifyPTS(y, z.n)$
4	$UnifyPTS(x, y)$
5	$\forall z \in P_x, P_{z.n} \supseteq \{c\}$ $\forall w \in P_{z.n}, Unify(w, c)$
6	$P_x \supseteq \{d\}$ $\forall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

Points-to Graph



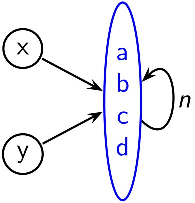
Example of Equality Based (aka Steensgaard's) Points-to Analysis

Program



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 $x \rightarrow n = y$

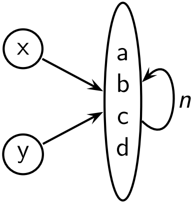
4 $y = x$

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	$P_x \supseteq \{d\}$ $\forall z \in P_x, Unify(d, z)$

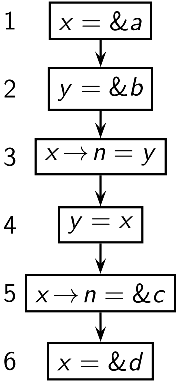
Points-to Graph



No further change

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

Program



1 $x = \&a$

2 $y = \&b$

3 $x \rightarrow n = y$

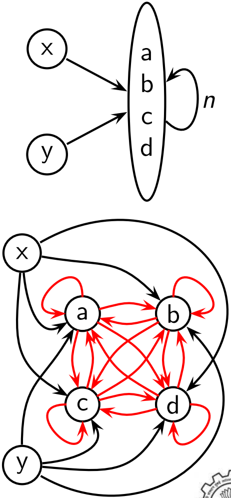
4 $y = x$

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

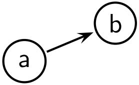
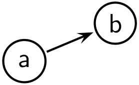
- Andersen's algorithm is cubic in number of pointers
- Steensgaard's algorithm is nearly linear in number of pointers
  - How can it be more efficient by an orders of magnitude?

Efficiency of Equality Based Approach

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

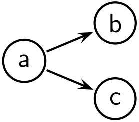
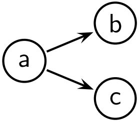
- Andersen's inclusion based wisdom:
    - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
    - Merge multiple successors and maintain a single successor of any node

Efficiency of Equality Based Approach

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

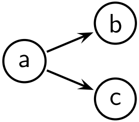
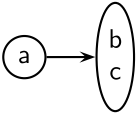
- Andersen's inclusion based wisdom:
  - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
  - Merge multiple successors and maintain a single successor of any node

Efficiency of Equality Based Approach

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

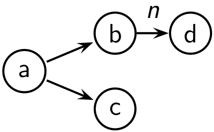
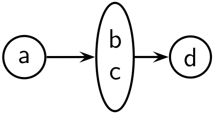
- Andersen's inclusion based wisdom:
    - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
    - Merge multiple successors and maintain a single successor of any node

Efficiency of Equality Based Approach

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

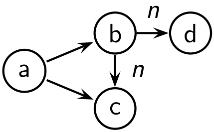
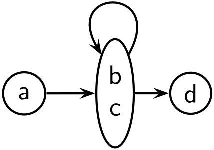
- Andersen's inclusion based wisdom:
  - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
  - Merge multiple successors and maintain a single successor of any node

Efficiency of Equality Based Approach

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

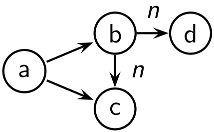
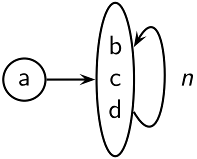
- Andersen's inclusion based wisdom:
  - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
  - Merge multiple successors and maintain a single successor of any node

Efficiency of Equality Based Approach

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

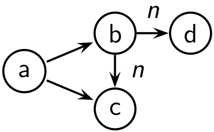
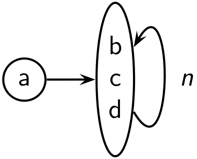
- Andersen's inclusion based wisdom:
    - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
    - Merge multiple successors and maintain a single successor of any node

Efficiency of Equality Based Approach

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

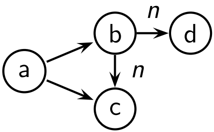
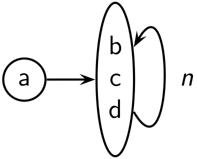
- Andersen's inclusion based wisdom:
    - Add edges and let the number of successors increase
- Steensgaard's equality based wisdom:
    - Merge multiple successors and maintain a single successor of any node

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

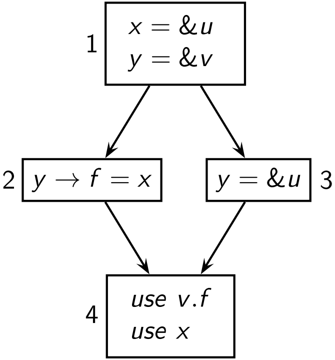
Efficiency of Equality Based Approach

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

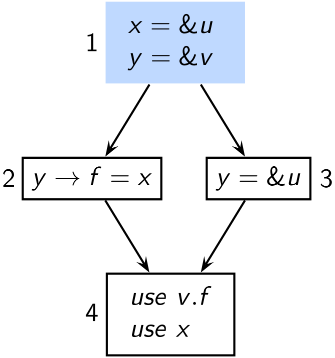
Inclusion Based (aka Andersen's) Points-to Analysis: Example 2

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

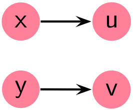


Inclusion Based (aka Andersen's) Points-to Analysis: Example 2

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



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



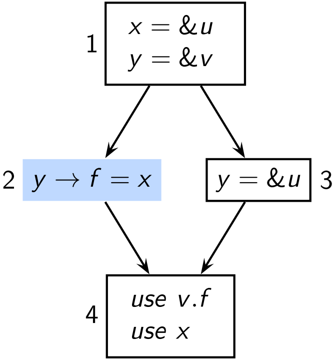
Andersen's Points-to Graph

Constraints on Points-to Sets

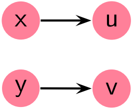
$$P_x \supseteq \{u\}$$
$$P_y \supseteq \{v\}$$

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



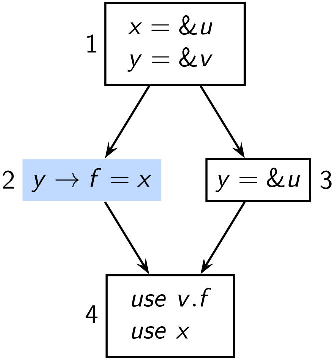
Andersen's Points-to Graph

Constraints on Points-to Sets

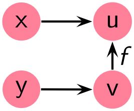
$$P_x \supseteq \{u\}$$
$$P_y \supseteq \{v\}$$
$$\forall w \in P_y,\ P_{w.f} \supseteq P_x$$

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



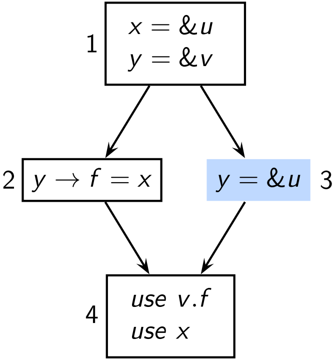
Andersen's Points-to Graph

Constraints on Points-to Sets

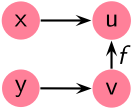
$$P_x \supseteq \{u\}$$
$$P_y \supseteq \{v\}$$
$$\forall w \in P_y,\ P_{w.f} \supseteq P_x$$

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



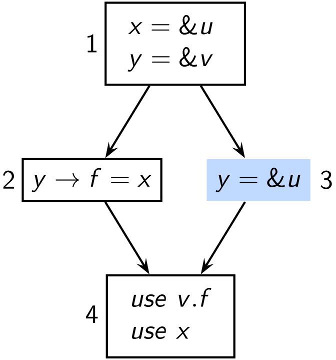
Andersen's Points-to Graph

Constraints on Points-to Sets

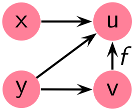
$$
\begin{aligned}
P_x &\supseteq \{u\} \\
P_y &\supseteq \{v\} \\
\forall w \in P_y,\ P_{w.f} &\supseteq P_x \\
P_y &\supseteq \{u\}
\end{aligned}
$$

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

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



- y should point to u also



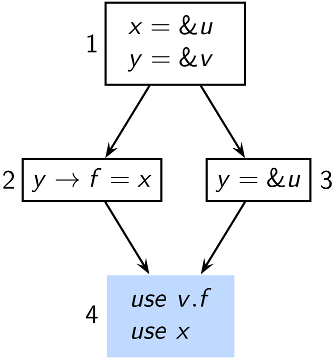
Andersen's Points-to Graph

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;
```

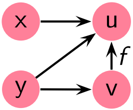


1 $x = \&u$ $y = \&v$

2 $y \rightarrow f = x$

3 $y = \&u$

4 $use\ v.f$ $use\ x$



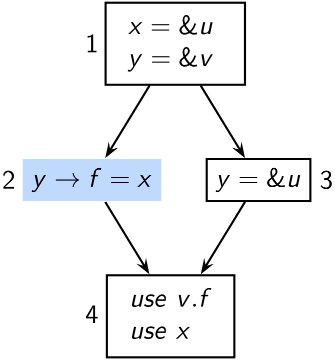
Andersen's Points-to Graph

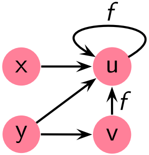
Constraints on Points-to Sets

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

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

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





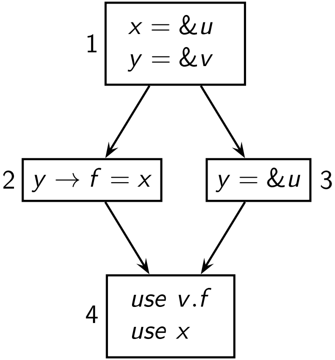
Andersen's Points-to Graph

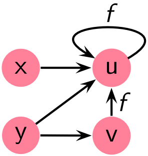
Constraints on Points-to Sets

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

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

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





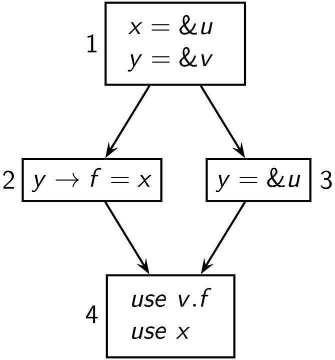
Andersen's Points-to Graph

Constraints on Points-to Sets

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

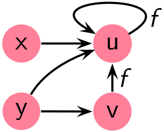
Equality Based (aka Steensgaard's) Points-to Analysis: Example 2

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



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

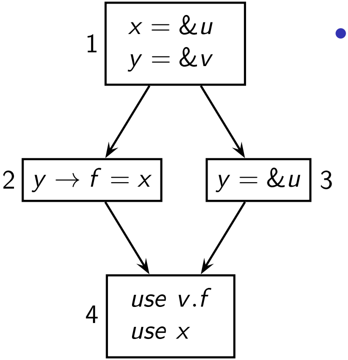
  Pointees of all equivalent locations become equivalent



Andersen's Points-to Graph

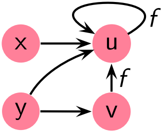
Equality Based (aka Steensgaard's) Points-to Analysis: Example 2

```
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

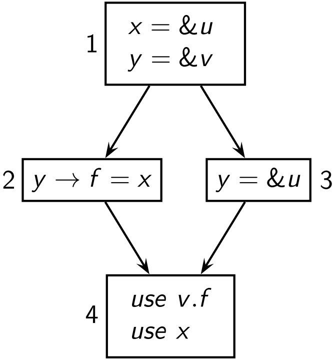
Effective additional constraints

$Unify(u, v)$

/* pointees of y */

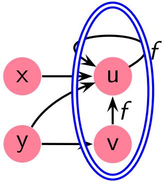
Equality Based (aka Steensgaard's) Points-to Analysis: Example 2

```
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

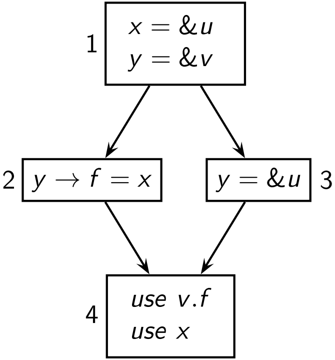
Effective additional constraints

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

$\Rightarrow u, v$ are equivalent

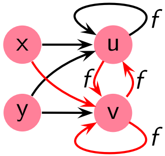
Equality Based (aka Steensgaard's) Points-to Analysis: Example 2

```
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

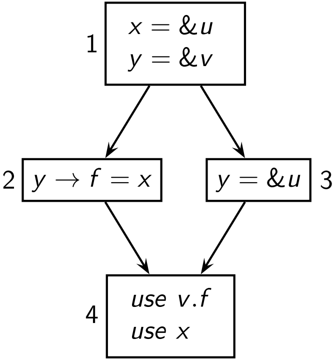
Effective additional constraints

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

⇒ u, v are equivalent

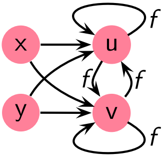
Equality Based (aka Steensgaard's) Points-to Analysis: Example 2

```
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

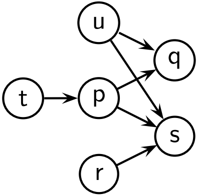
Unify(u, v)
/* pointees of y */

$\Rightarrow$ u, v are equivalent

Tutorial Problem for Flow-Insensitive Pointer Analysis

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

Tutorial Problem for Flow-Insensitive Pointer Analysis

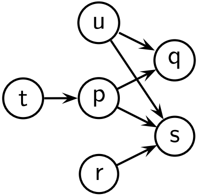
Program	Inclusion based	Equality based
p = &q r = &s t = &p u = p ∗t = r		

Tutorial Problem for Flow-Insensitive Pointer Analysis

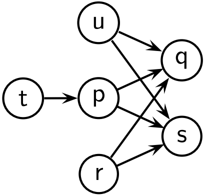
Program

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

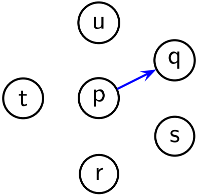
Inclusion based



Equality based



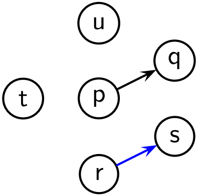
Tutorial Problem for Flow-Insensitive Pointer Analysis

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

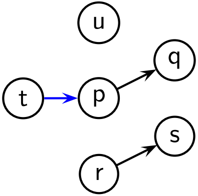
Tutorial Problem for Flow-Insensitive Pointer Analysis

Program | Inclusion based | Equality based

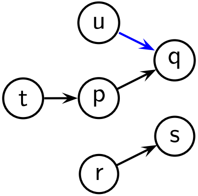
```
p = &q
r = &s
t = &p
u = p
*t = r
```



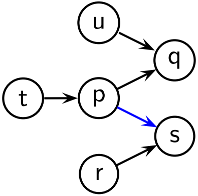
Tutorial Problem for Flow-Insensitive Pointer Analysis

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

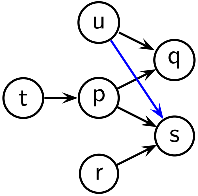
Tutorial Problem for Flow-Insensitive Pointer Analysis

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

Tutorial Problem for Flow-Insensitive Pointer Analysis

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

Tutorial Problem for Flow-Insensitive Pointer Analysis

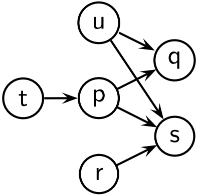
Program	Inclusion based	Equality based
$p = \&q$ $r = \&s$ $t = \&p$ $u = p$ $*t = r$		

Tutorial Problem for Flow-Insensitive Pointer Analysis

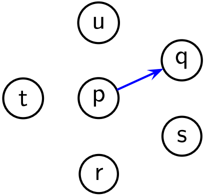
Program

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

Inclusion based



Equality based

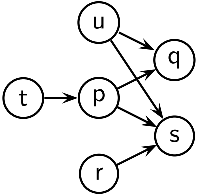


Tutorial Problem for Flow-Insensitive Pointer Analysis

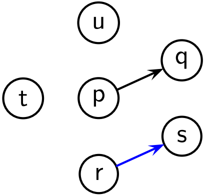
Program

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

Inclusion based



Equality based

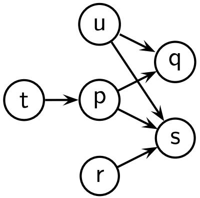


Tutorial Problem for Flow-Insensitive Pointer Analysis

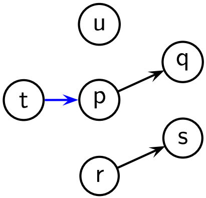
Program

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

Inclusion based



Equality based

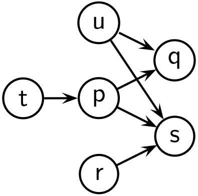


Tutorial Problem for Flow-Insensitive Pointer Analysis

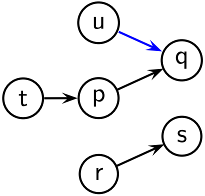
Program

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

Inclusion based



Equality based

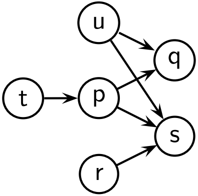


Tutorial Problem for Flow-Insensitive Pointer Analysis

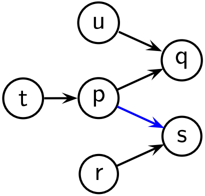
Program

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

Inclusion based



Equality based

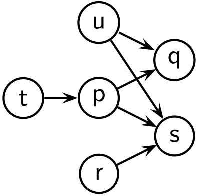


Tutorial Problem for Flow-Insensitive Pointer Analysis

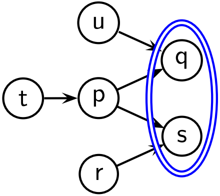
Program

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

Inclusion based



Equality based

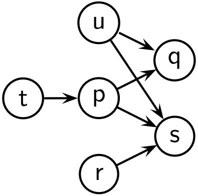


Tutorial Problem for Flow-Insensitive Pointer Analysis

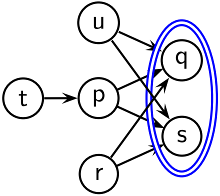
Program

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

Inclusion based



Equality based

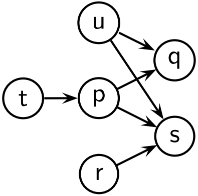


Tutorial Problem for Flow-Insensitive Pointer Analysis

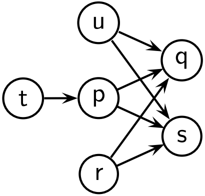
Program

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

Inclusion based



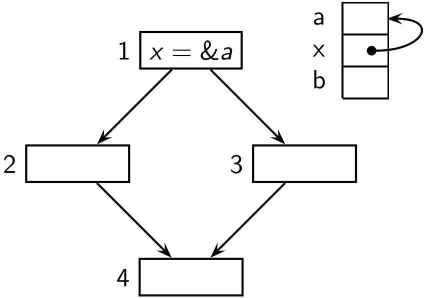
Equality based



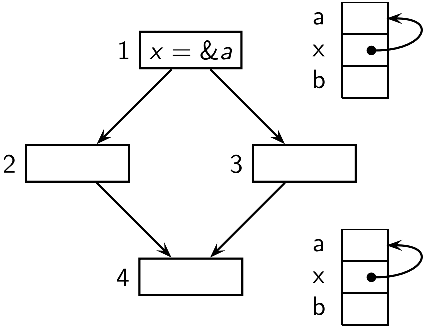
An Outline of Pointer Analysis Coverage

- The larger perspective
- IR for Points-to Analysis
- Flow-Insensitive Points-to Analysis
- Flow-Sensitive Points-to Analysis Next Topic

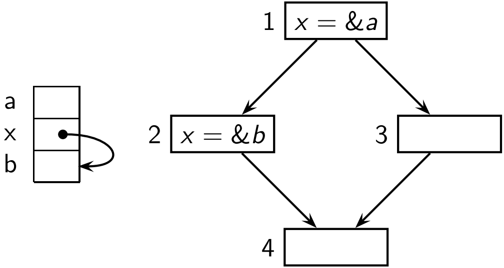
Must Points-to Information



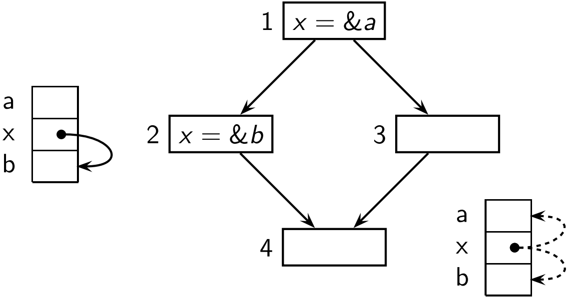
Must Points-to Information



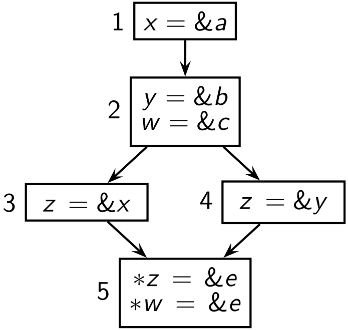
May Points-to Information



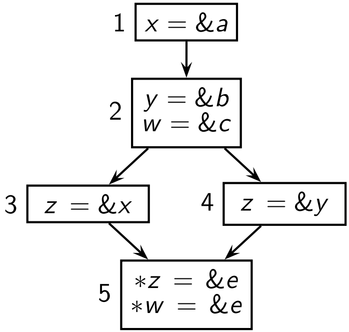
May Points-to Information



Strong and Weak Updates



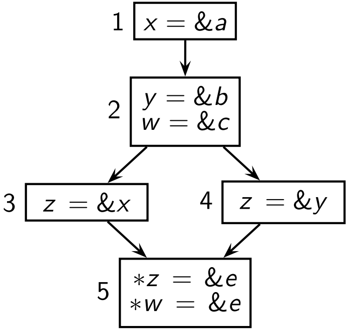
Strong and Weak Updates



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

  Only Gen, No Kill

Strong and Weak Updates



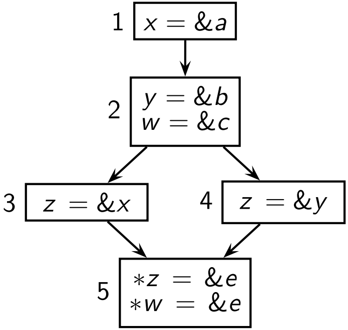
- Weak update: Modification of x or y due to $*z$ in block 5

  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

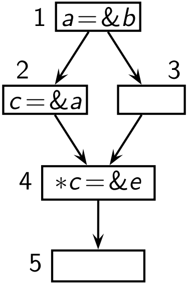
  Both Gen and Kill

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

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

MFP of May Points-to Analysis

MFP of Must Points-to Analysis



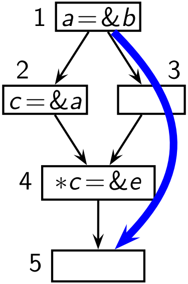
May and Must Analysis for Killing Points-to Information (1)

MFP of Must Points-to Analysis

MFP of May Points-to Analysis

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

  Holds along path 1-3-4
- (a, b) should not be killed in node 4
- Possible if pointee set of c is $\emptyset$
- However, $MayIn_4$ contains (c, a)

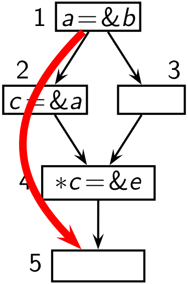


May and Must Analysis for Killing Points-to Information (1)

MFP of May Points-to Analysis

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

  Holds along path 1-3-4
- (a, b) should not be killed in node 4
- Possible if pointee set of c is $\emptyset$
- However, $MayIn_4$ contains (c, a)



MFP of Must Points-to Analysis

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

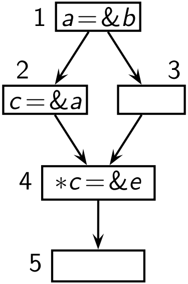
  Does not hold along path 1-2-4
- (a, b) should be killed in node 4
- Possible if pointee set of c is $\{a\}$
- However, the pointee set of c is $\emptyset$ in $MustIn_4$

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

MFP of May Points-to Analysis

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

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



MFP of Must Points-to Analysis

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

  Does not hold along path 1-2-4
- (a, b) should be killed in node 4
- 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

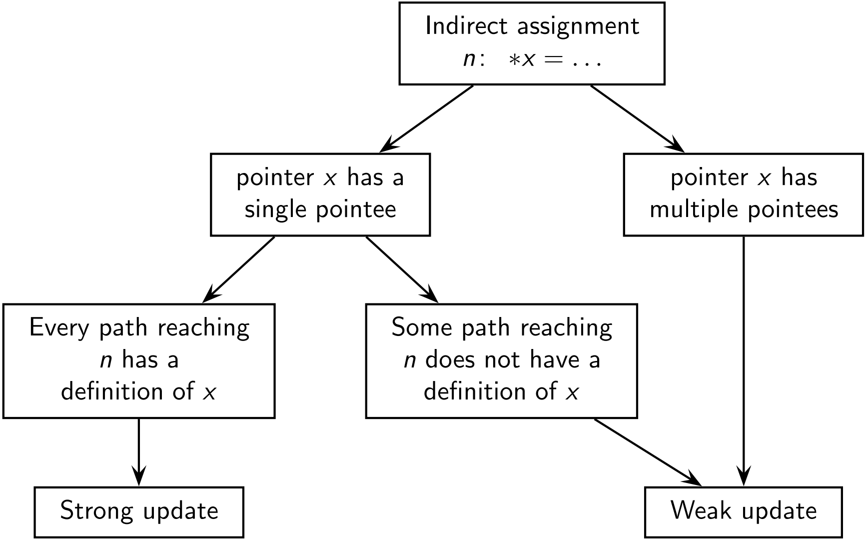
  ⇒ should use Must Points-to information

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

  Kill should remove ALL weak updates

  ⇒ should use May Points-to information

Distinguishing Between Strong and Weak Updates



Indirect assignment
$n: \quad *x = \ldots$

pointer x has a single pointee

pointer x has multiple pointees

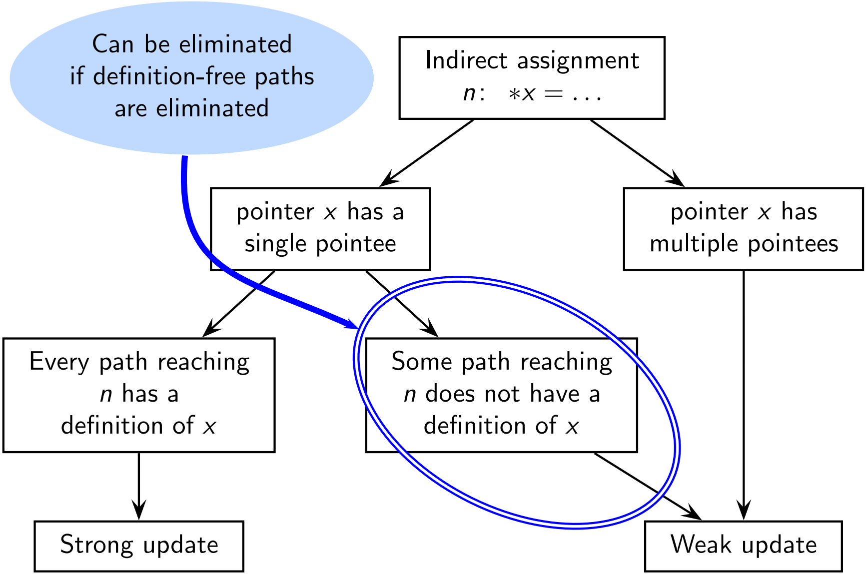
Every path reaching n has a definition of x

Some path reaching n does not have a definition of x

Strong update

Weak update

Distinguishing Between Strong and Weak Updates



Can be eliminated if definition-free paths are eliminated

Indirect assignment $n: \;\; *x = \ldots$

pointer x has a single pointee

pointer x has multiple pointees

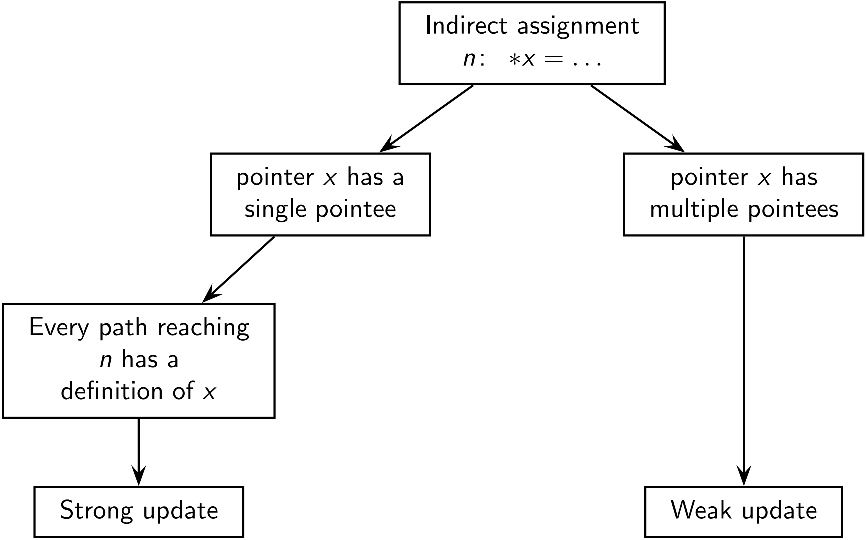
Every path reaching n has a definition of x

Some path reaching n does not have a definition of x

Strong update

Weak update

Distinguishing Between Strong and Weak Updates



Indirect assignment
$n: \ *x = \ldots$

pointer x has a single pointee

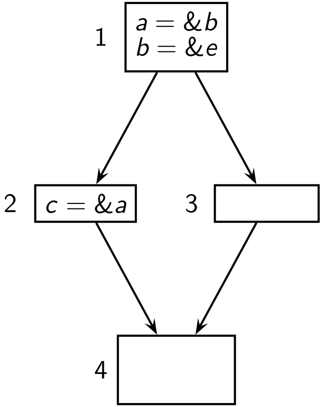
pointer x has multiple pointees

Every path reaching n has a definition of x

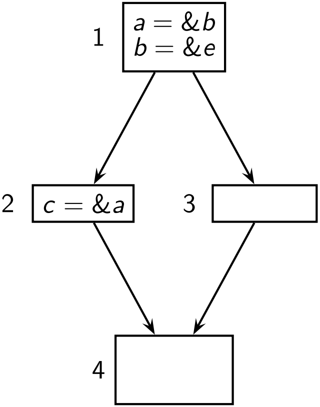
Strong update

Weak update

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



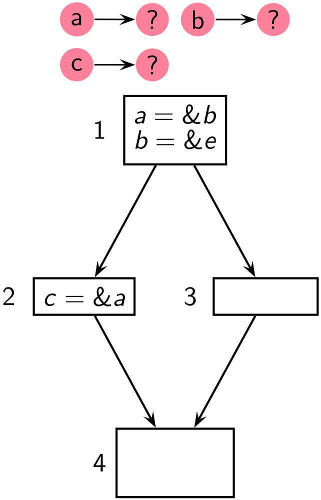
Discovering Must Points-to Information from May Points-to Information



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

  Assume that e is a scalar

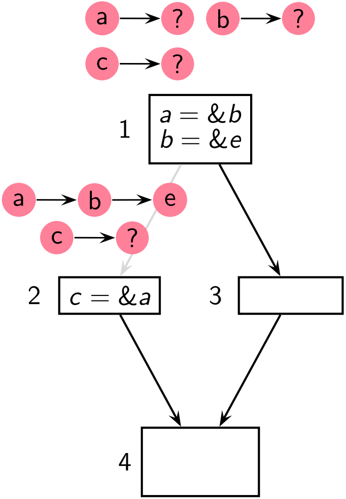
Discovering Must Points-to Information from May Points-to Information



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

  Assume that e is a scalar

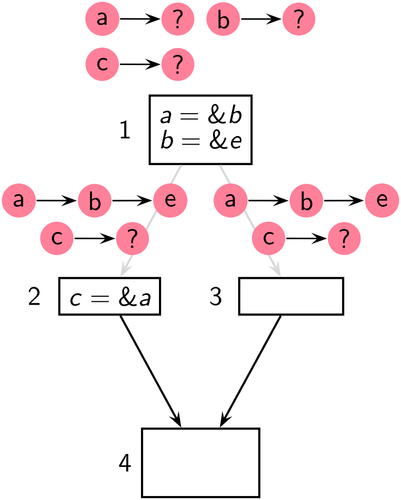
Discovering Must Points-to Information from May Points-to Information



- *BI*. every pointer points to “?”

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

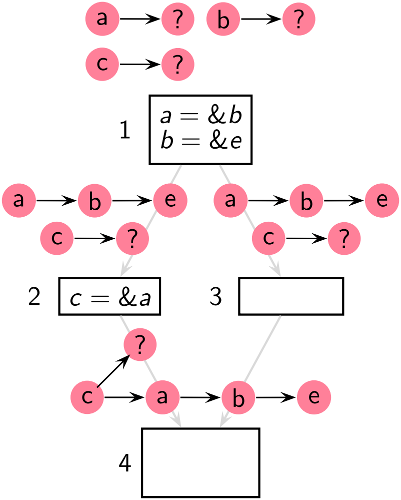
Discovering Must Points-to Information from May Points-to Information



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

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

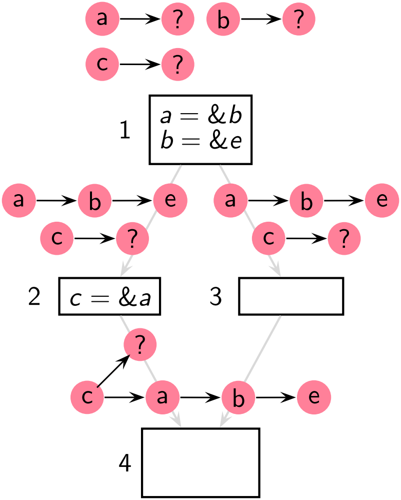
Discovering Must Points-to Information from May Points-to Information



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

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

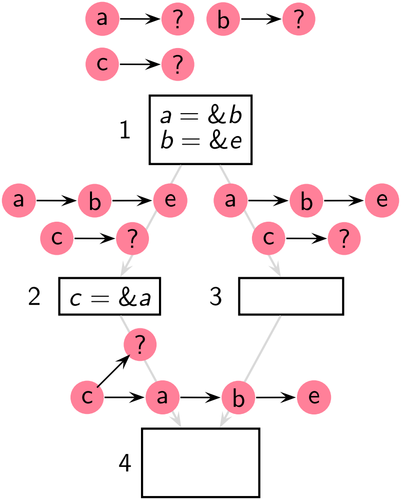
Discovering Must Points-to Information from May Points-to Information



- *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

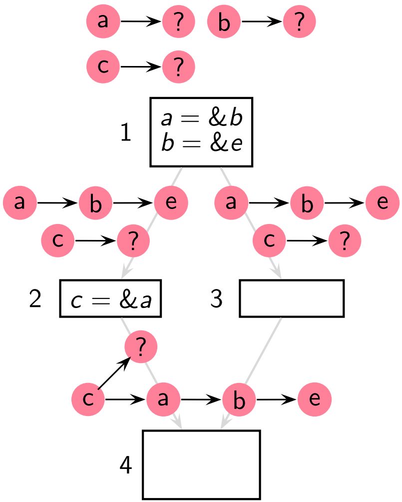
Discovering Must Points-to Information from May Points-to Information



- *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

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



The use of “?” to derive Must is valid under the following conditions

If there is a definition free path from *Start* to node i for pointer x, then $(x, ?)$ must reach In_i during the very first visit to node i in the analysis.

Conversely, if there is no definition free path from *Start* to node i for pointer x, then $(x, ?)$ must *not* reach In_i during the very first visit to node i in the analysis.

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

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

Then,

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

Relevant Algebraic Operations on Relations (2)

Let

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

Then,

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

Relevant Algebraic Operations on Relations (2)

Let

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

Then,

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

Points-to Analysis Data Flow Equations

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

$$Pout_n = \Big(Pin_n - \big(\ Kill_n \times V\big)\Big) \cup \big(\ Def_n \times Pointee_n\ \big)$$

- $Pin/Pout$: sets of may points-to pairs
- $Kill_n$, Def_n, and $Pointee_n$ are defined in terms of Pin_n

Points-to Analysis Data Flow Equations

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

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

- $Pin/Pout$: sets of may points-to pairs
- $Kill_n$, Def_n, and $Pointee_n$ are defined in terms of Pin_n

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

Points-to Analysis Data Flow Equations

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

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

- $Pin/Pout$: sets of may points-to pairs
- $Kill_n$, Def_n, and $Pointee_n$ are defined in terms of Pin_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)

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

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

- *Pin*/*Pout*: sets of may points-to pairs
- $Kill_n$, Def_n, and $Pointee_n$ are defined in terms of Pin_n

Points-to Analysis Data Flow Equations

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

$$
Pout_n = \Big(Pin_n - \big(\ Kill_n \ \times \ V\big)\Big) \cup \big(\ Def_n \ \times \ Pointee_n\ \big)
$$

- $Pin/Pout$: sets of may points-to pairs
- $Kill_n$, Def_n, and $Pointee_n$ are defined in terms of Pin_n

Extractor Functions for Points-to Analysis

Values defined in terms of Pin_n (denoted P)

	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 Pin_n (denoted P)

	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 Pin_n (denoted P)

	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 Pin_n (denoted P)

	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 Pin_n (denoted P)

	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 Pin_n (denoted P)

	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 Pin_n (denoted P)

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

Pointees of y in Pin_n are the targets of defined pointers

Extractor Functions for Points-to Analysis

Values defined in terms of Pin_n (denoted P)

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

Pointees of those pointees of y in Pin_n which are pointers

Extractor Functions for Points-to Analysis

Values defined in terms of Pin_n (denoted P)

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

Pointees of x in Pin_n receive new addresses

Extractor Functions for Points-to Analysis

Values defined in terms of *Pi*

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

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

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

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

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

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

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

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

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

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

Extractor Functions for Points-to Analysis

Values defined in terms of Pin_n (denoted P)

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

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

Pointees of y in Pin_n are the targets of defined pointers

Extractor Functions for Points-to Analysis

Values defined in terms of Pin_n (denoted P)

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

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

Extractor Functions for Points-to Analysis

Values defined in terms of Pin_n (denoted P)

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

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

Extractor Functions for Points-to Analysis

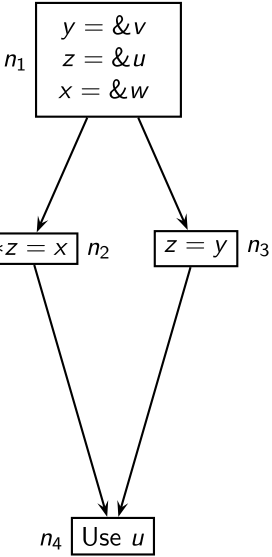
Values defined in terms of Pin_n (denoted P)

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

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

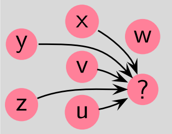
An Example of Flow-Sensitive May Points-to Analysis

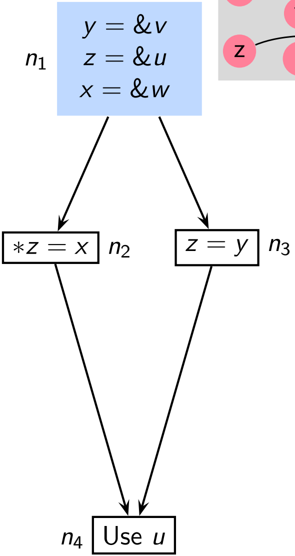
```
int w;
int *u, *v, *x;
int **y, **z;
```



An Example of Flow-Sensitive May Points-to Analysis

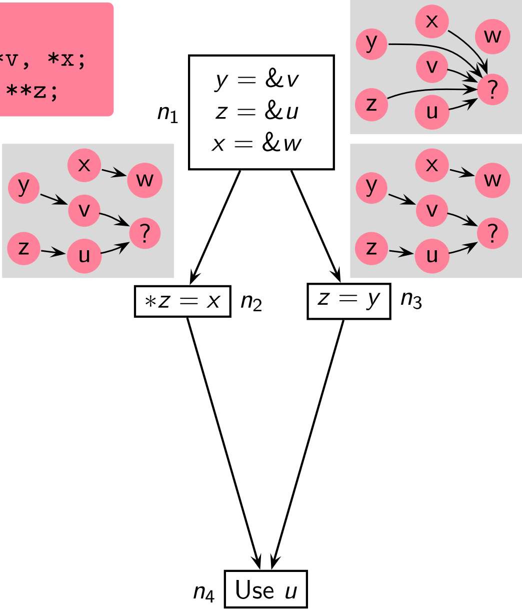
```
int w;
int *u, *v, *x;
int **y, **z;
```





An Example of Flow-Sensitive May Points-to Analysis

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



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

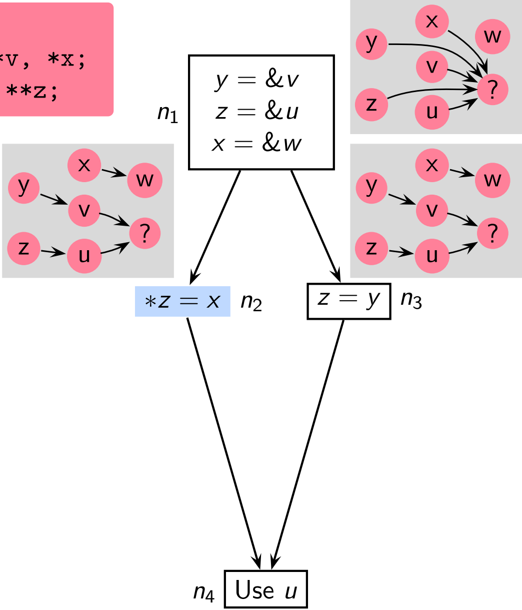
n_2: $*z = x$

n_3: $z = y$

n_4: Use u

An Example of Flow-Sensitive May Points-to Analysis

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



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

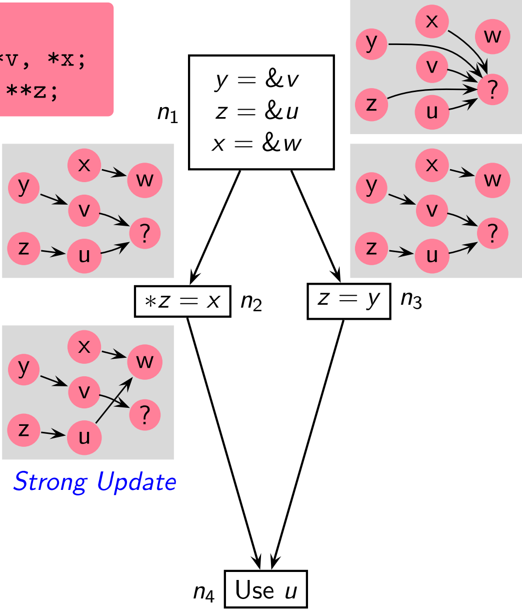
n_2: $*z = x$

n_3: $z = y$

n_4: Use u

An Example of Flow-Sensitive May Points-to Analysis

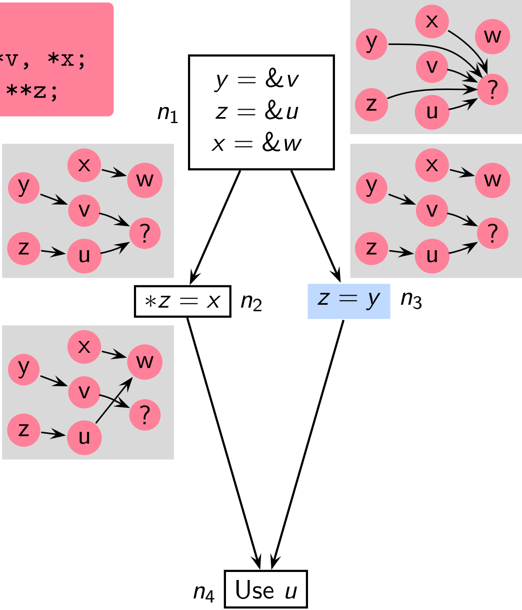
```
int w;
int *u, *v, *x;
int **y, **z;
```



Strong Update

An Example of Flow-Sensitive May Points-to Analysis

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



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

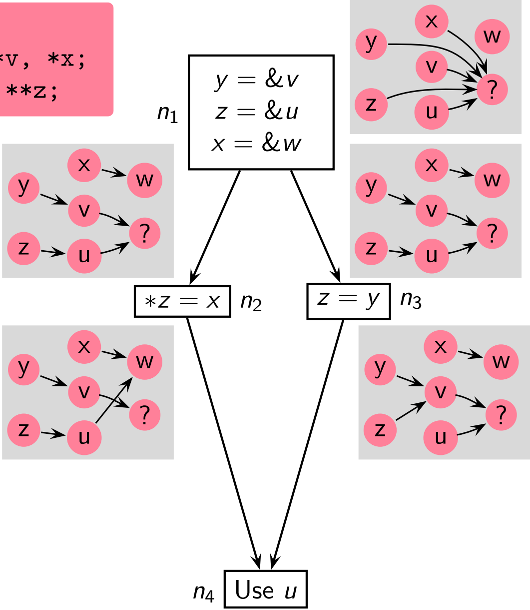
n_2: $*z = x$

n_3: $z = y$

n_4: Use u

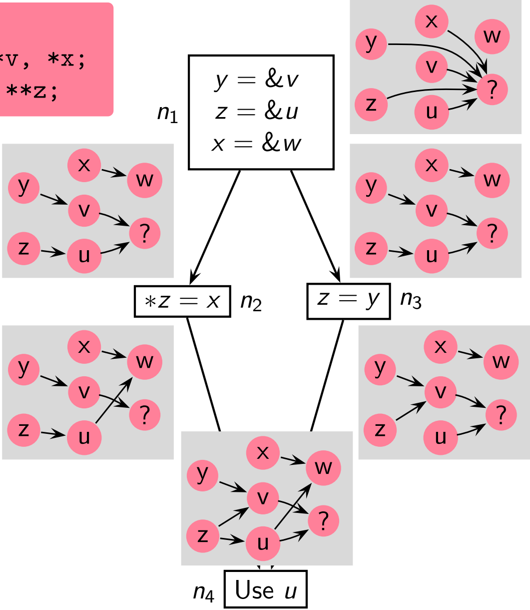
An Example of Flow-Sensitive May Points-to Analysis

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



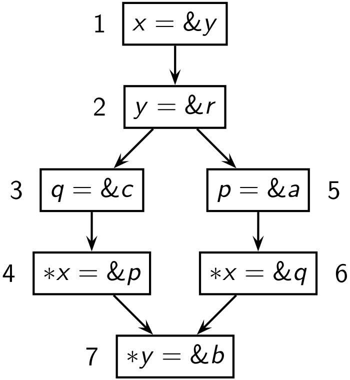
An Example of Flow-Sensitive May Points-to Analysis

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



Tutorial Problem for Flow-Sensitive Pointer Analysis

int a, b, c, $*p$, $*q$, $*r$;
int $**y$, $***x$;



Solution of Tutorial Problem

	Pin_n	$Pout_n$
1	$\{(p,?),(q,?),(r,?),(x,?),(y,?)\}$	$\{(p,?),(q,?),(r,?),(x,y),(y,?)\}$
2	$\{(p,?),(q,?),(r,?),(x,y),(y,?)\}$	$\{(p,?),(q,?),(r,?),(x,y),(y,r)\}$
3	$\{(p,?),(q,?),(r,?),(x,y),(y,r)\}$	$\{(p,?),(q,c),(r,?),(x,y),(y,r)\}$
4	$\{(p,?),(q,c),(r,?),(x,y),(y,r)\}$	$\{(p,?),(q,c),(r,?),(x,y),(y,p)\}$
5	$\{(p,?),(q,?),(r,?),(x,y),(y,r)\}$	$\{(p,a),(q,?),(r,?),(x,y),(y,r)\}$
6	$\{(p,a),(q,?),(r,?),(x,y),(y,r)\}$	$\{(p,a),(q,?),(r,?),(x,y),(y,q)\}$
7	$\{(p,?),(p,a),(q,?),(q,c),$ $(r,?),(x,y),(y,p)(y,q)\}$	$\{(p,?),(p,a),(p,b),(q,?),(q,c),(q,b),$ $(r,?),(x,y),(y,p)(y,q)\}$

Extractor Functions in the Presence of Structures (1)

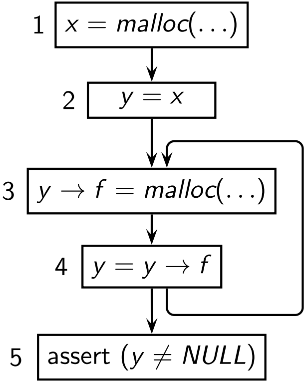
- We extend pointer to use field names as follows:
  - pointer x is represented by $(x, *)$, and
  - pointer field f of structure variable x is represented by (x, f)
  - points-to information is of the form $((x, f)\, y)$
- For simplicity, we
  - separate LHS and RHS assuming that
  - only legal, type-correct 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 of pointer expression: Usually based on the length of pointer expression

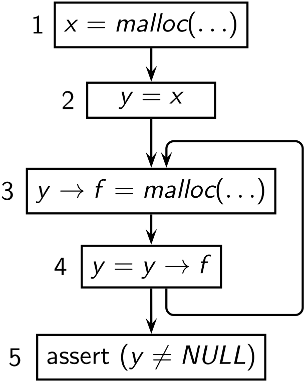
Allocation Site Based Abstraction of Points-to Graph

Program

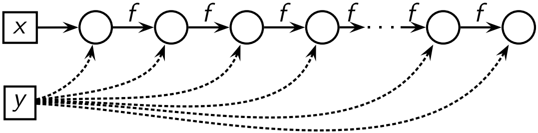


Allocation Site Based Abstraction of Points-to Graph

Program

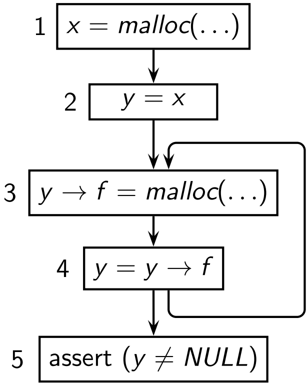


Memory graph representing multiple executions

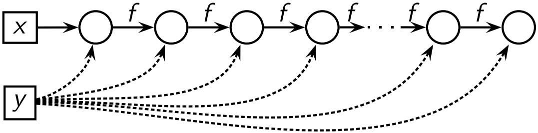


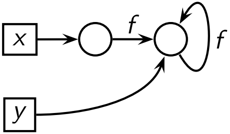
Allocation Site Based Abstraction of Points-to Graph

Program



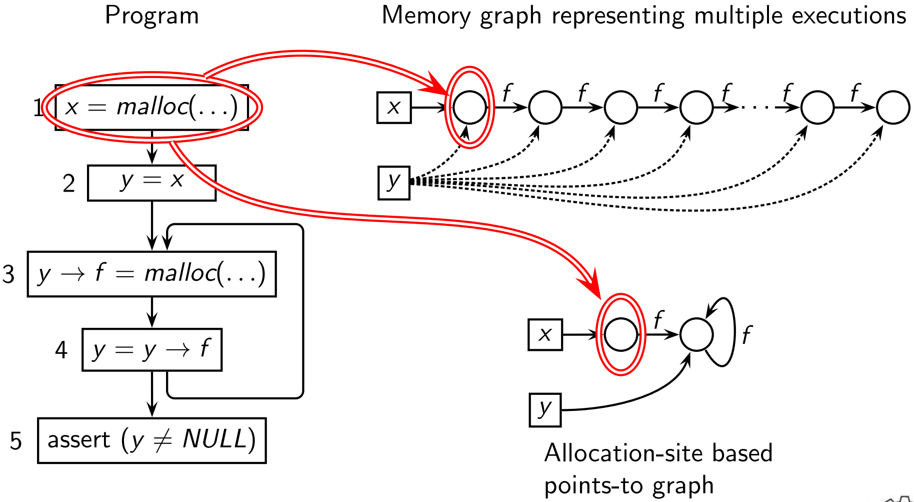
Memory graph representing multiple executions



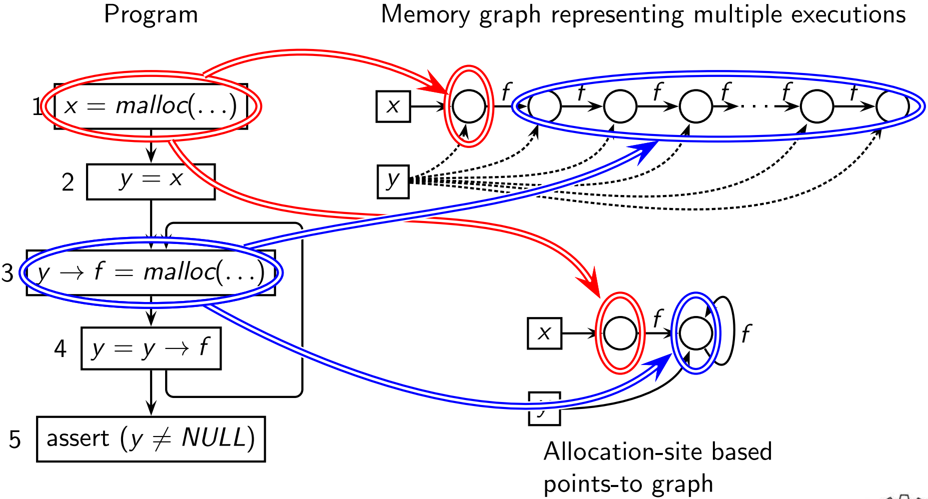


Allocation-site based points-to graph

Allocation Site Based Abstraction of Points-to Graph



Allocation Site Based Abstraction of Points-to Graph

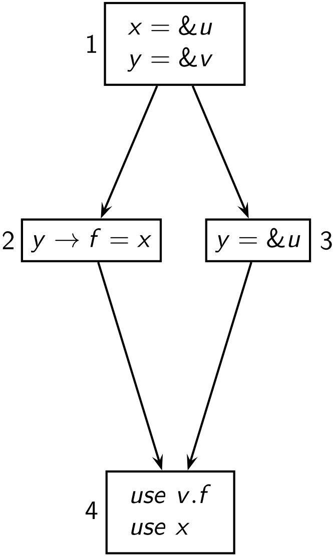


Extractor Functions in the Presence of Structures (2)

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



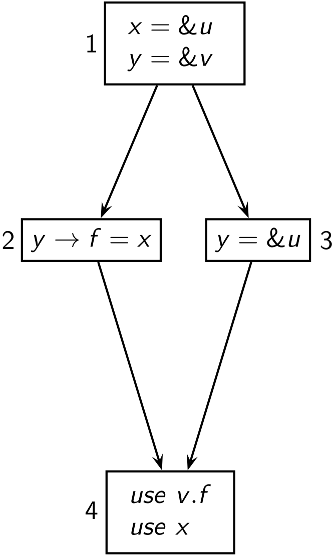
Type Information

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

Andersen's Points-to Graph

Steensgaard's Points-to Graph

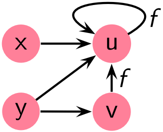
An Example of Flow-Sensitive May Points-to Analysis



Type Information

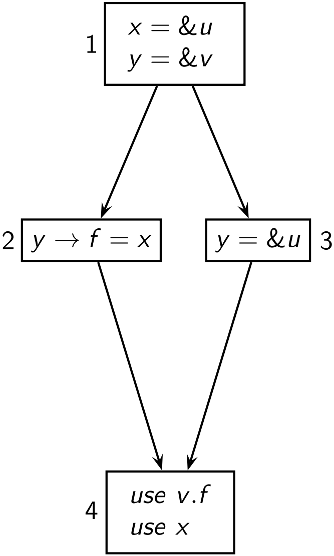
```
struct s {
    struct s *f;
    int n;
}   *x, *y, u, v;
```

Andersen's Points-to Graph



Steensgaard's Points-to Graph

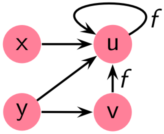
An Example of Flow-Sensitive May Points-to Analysis



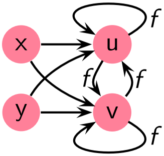
Type Information

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

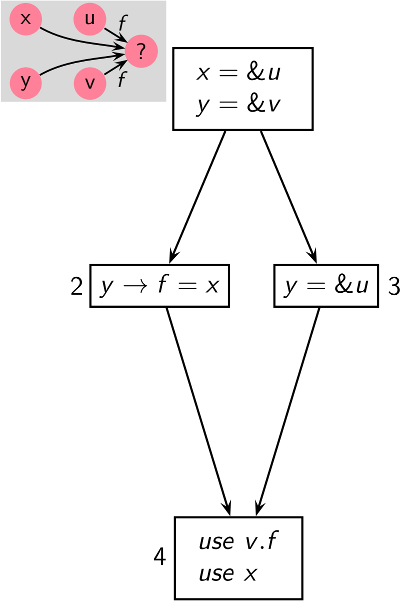
Andersen's Points-to Graph



Steensgaard's Points-to Graph



An Example of Flow-Sensitive May Points-to Analysis



$x = \&u$
$y = \&v$

$y \rightarrow f = x$

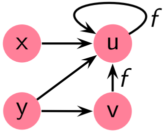
$y = \&u$

use $v.f$
use x

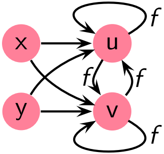
Type Information

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

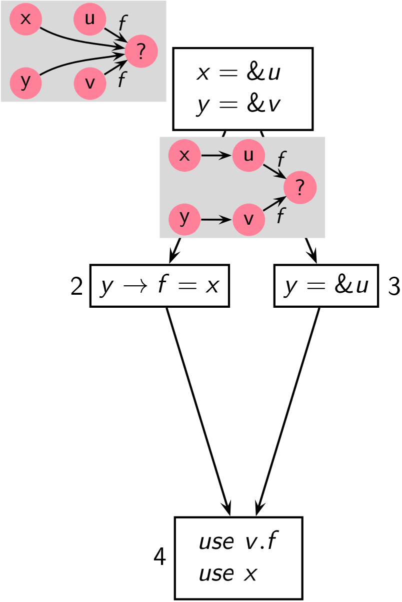
Andersen's Points-to Graph



Steensgaard's Points-to Graph



An Example of Flow-Sensitive May Points-to Analysis



$x = \&u$
$y = \&v$

2 $y \rightarrow f = x$

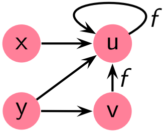
$y = \&u$ 3

4 *use* $v.f$
use x

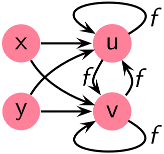
Type Information

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

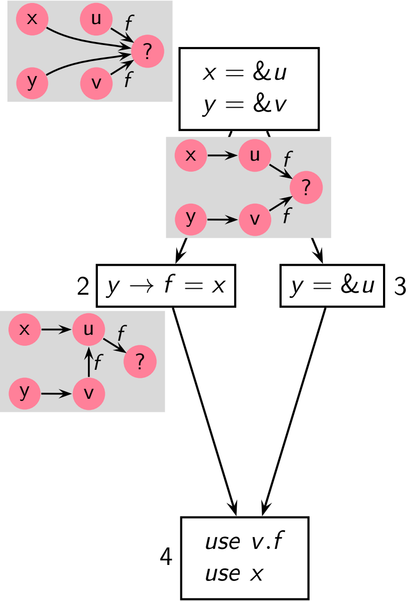
Andersen's Points-to Graph



Steensgaard's Points-to Graph



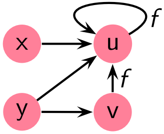
An Example of Flow-Sensitive May Points-to Analysis



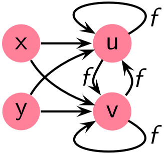
Type Information

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

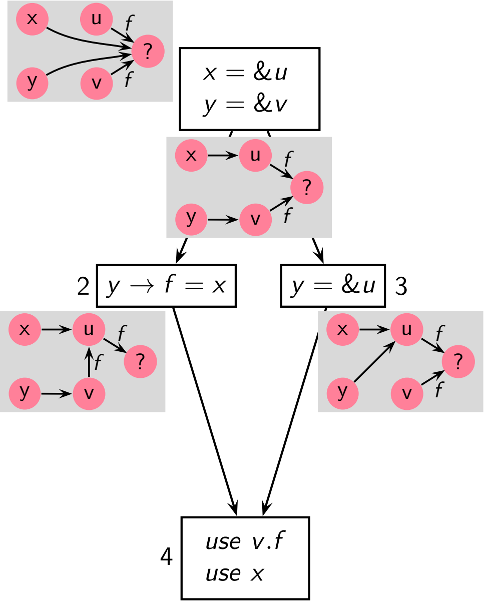
Andersen's Points-to Graph



Steensgaard's Points-to Graph



An Example of Flow-Sensitive May Points-to Analysis



$x = \&u$
$y = \&v$

2 $y \rightarrow f = x$

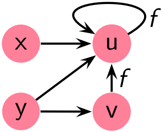
$y = \&u$ 3

4 *use* $v.f$
use x

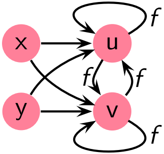
Type Information

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

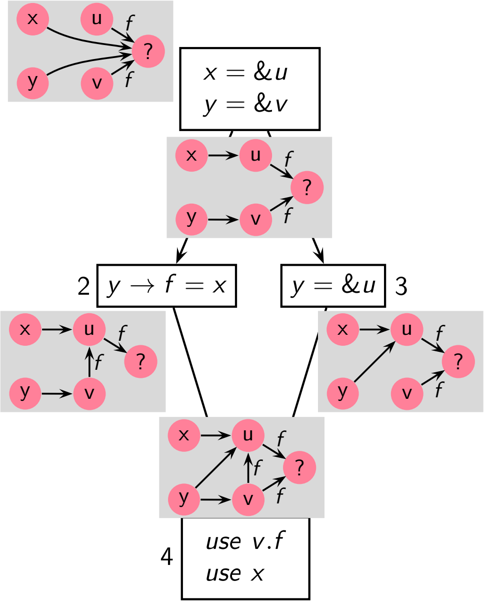
Andersen's Points-to Graph



Steensgaard's Points-to Graph



An Example of Flow-Sensitive May Points-to Analysis



$x = \&u$
$y = \&v$

2 $y \rightarrow f = x$

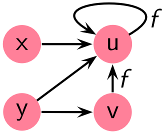
$y = \&u$ 3

4 use $v.f$
use x

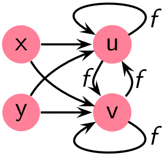
Type Information

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

Andersen's Points-to Graph

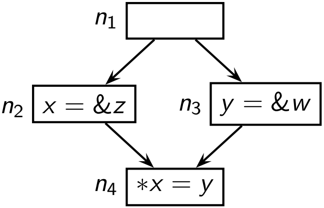


Steensgaard's Points-to Graph



Non-Distributivity of Points-to Analysis

May Points-to



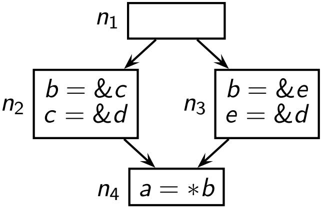
n_1

n_2 $x = \&z$

n_3 $y = \&w$

n_4 $*x = y$

Must Points-to



n_1

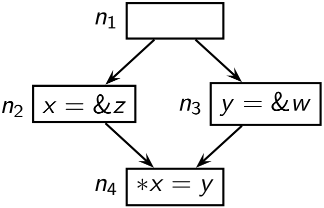
n_2 $b = \&c$ $c = \&d$

n_3 $b = \&e$ $e = \&d$

n_4 $a = *b$

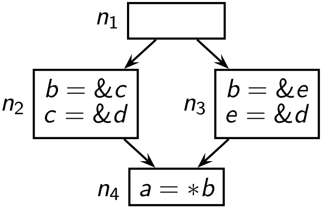
Non-Distributivity of Points-to Analysis

May Points-to



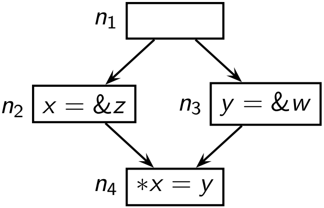
$z \rightarrowtail w$ is spurious

Must Points-to



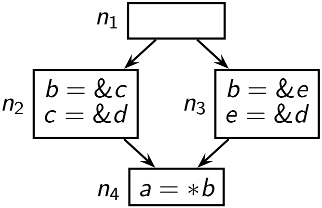
Non-Distributivity of Points-to Analysis

May Points-to



$z \rightarrowtail w$ is spurious

Must Points-to



$a \rightarrowtail d$ is missing