LUND UNIVERSITY

EDA045F: Program Analysis

LECTURE 5: POINTER ANALYSIS 1

Christoph Reichenbach

In the last lecture...

- Procedure Summaries
- IFDS algorithm
- IDE algorithm
- Path Sensitivity

Our Memory Modelling Until Now

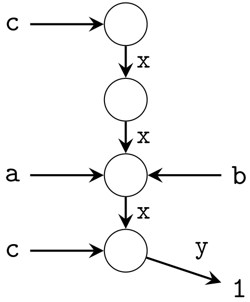
- Our analyses so far have considered:
  - Static Variables
  - Local (stack-dynamic) Variables
  - (Stack-dynamic) parameters

Missing: heap variables!

Example Program

Example

```
a = new();    // ⇐
a.x = null;   // ⇐
b = a;        // ⇐
b.x = new();  // ⇐
a.x.y = 1;    // ⇐
c = new();    // ⇐
c.x = new();  // ⇐
c.x.x = a;    // ⇐
c = a.x;      // ⇐
// A
```



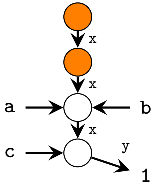
Concrete Heap Graph

“v points to ℓ”

$$v \rightarrow \ell$$

variable location

- Heap graph connects memory locations
- Represents all heap-allocated objects and their points-to relationships
- Edges labelled with field names
- Some objects not reachable from variables



Aliasing

Example

```
a = new();
a.x = null;
b = a;
b.x = new();
a.x.y = 1;
c = new();
c.x = new();
c.x.x = a;
c = a.x;
// A
```

Aliases at `// A`:

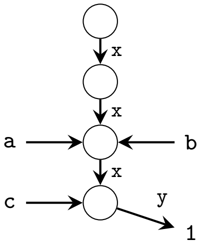
- `a` and `b` represent the same object

$\Longrightarrow$ `a` and `b` are *aliased*

$$\mathtt{a} \overset{\textit{alias}}{=\!=} \mathtt{b}$$

$\Longrightarrow$ `a.x` and `b.x` are *aliased*

- `c` and `a.x` and b.x are *aliased*



Pointer Analysis

$$\begin{tikzcd} a \arrow[r] & \ell_0 \\ b \arrow[ur] & \\ c \arrow[r] & \ell_1 \end{tikzcd}$$

- *Points-To Analysis*:
  - Analyse *heap usage*
  - Which *variables* may/must point to which *heap locations*?

$$a \rightarrow \ell_0$$

- *Alias Analysis*:
  - Analyse *address sharing*
  - Which *pair/set of variables* may/must point to the same address?

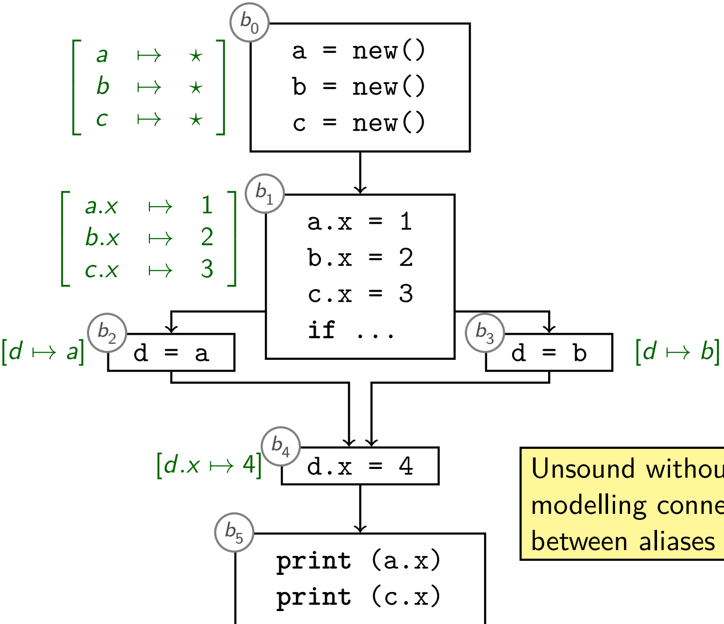
$$a \overset{alias}{=\!=} b$$

Summary: Pointer Analysis

- Class of analyses to model dynamic heap allocation
- **Points-To Analysis**: computes mapping
  - From *variables*
  - To *pointees* (other variables)
  - More general than Alias Analysis
- **Alias Analysis**: computes
  - *Sharing information* between variables
  - Implicitly produced by points-to analysis

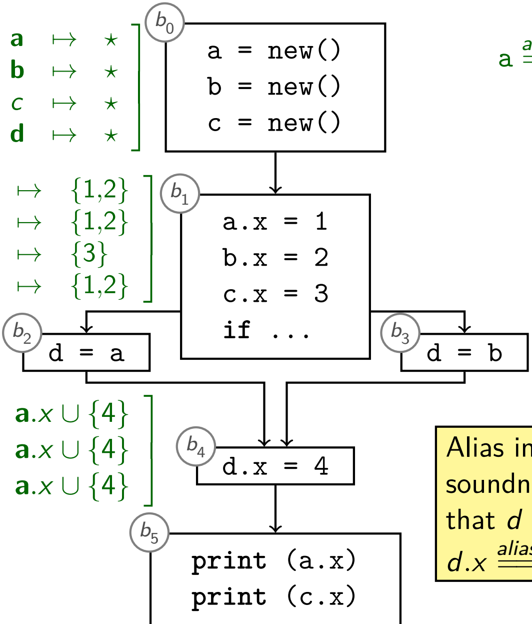
$$a \overset{alias}{=\!=} b \iff a \rightarrow \ell \leftarrow b$$

Dataflow with Alias Information



Unsound without modelling connection between aliases

Dataflow with Alias Information



$$\begin{bmatrix} \mathbf{a} & \mapsto & \star \\ \mathbf{b} & \mapsto & \star \\ c & \mapsto & \star \\ \mathbf{d} & \mapsto & \star \end{bmatrix}$$

$$\mathrm{a} \stackrel{alias}{=\!=} \mathrm{d} \stackrel{alias}{=\!=} \mathrm{b}$$

$$\begin{bmatrix} \mathbf{a}.x & \mapsto & \{1,2\} \\ \mathbf{b}.x & \mapsto & \{1,2\} \\ c.x & \mapsto & \{3\} \\ \mathbf{d}.x & \mapsto & \{1,2\} \end{bmatrix}$$

$[]$ $[]$

$$\begin{bmatrix} \mathbf{a}.x & \mapsto & \mathbf{a}.x \cup \{4\} \\ \mathbf{b}.x & \mapsto & \mathbf{a}.x \cup \{4\} \\ \mathbf{d}.x & \mapsto & \mathbf{a}.x \cup \{4\} \end{bmatrix}$$

Alias info allows soundness. Note that $d \stackrel{alias}{=\!=} a$ implies $d.x \stackrel{alias}{=\!=} a.x$.

Dataflow + Aliases

- Aliasing affects shared fields:

$$\mathtt{a} \overset{\mathit{alias}}{=\!=} \mathtt{b} \quad \Longrightarrow \quad \mathtt{a}.x \overset{\mathit{alias}}{=\!=} \mathtt{b}.x \text{ for all } x$$

- Exploiting aliasing knowledge:
  - Multiply *updates* for each alias:

$$\begin{bmatrix} \mathbf{a}.x & \mapsto & \mathbf{a}.x \cup \{4\} \\ \mathbf{b}.x & \mapsto & \mathbf{a}.x \cup \{4\} \\ \mathbf{d}.x & \mapsto & \mathbf{a}.x \cup \{4\} \end{bmatrix}$$

  - Multiply *reads* for each alias

$$\begin{bmatrix} \mathbf{a}.x & \mapsto & \mathbf{a}.x \cup \mathbf{b}.x \cup \mathbf{c}.x \cup \{4\} \end{bmatrix}$$

  - Replace aliased paths by single representative
    - **Most efficient**

Compute Aliases during Dataflow?

- Previoulsy: Dataflow analysis as *analysis client* of Alias analysis:
- Can use Dataflow Analysis to compute pointer analyses
- Caveat:
  `y.field = z`
  - Transfer function updates y`.field` by z
  - Must extract both y, z from $\mathbf{in}_b$ to compute update
  - *Non-distributive in practice*

Summary

- **Analysis client**: user of analysis, often another analysis
  - E.g., *Type analysis* is client of *name analysis*
- **Alias analysis** helps make dataflow analysis more precise
  - Fields inherit aliasing:

$$\mathtt{a} \overset{alias}{=\!=} \mathtt{b} \quad \Longrightarrow \quad \mathtt{a}.x \overset{alias}{=\!=} \mathtt{b}.x \text{ for all } x$$

  - So if $a.x \overset{alias}{=\!=} b.y$, then:
    - $a.x.z \overset{alias}{=\!=} b.y.z$
    - $a.x.z.z \overset{alias}{=\!=} b.y.z.z$
    - $a.x.z.z.z \overset{alias}{=\!=} b.y.z.z.z$ etc.
- Dataflow analysis can compute pointer analyses
  - Requires non-distributive framework for realistic languages

Concrete Heap Graphs (1/2)

Capturing the heap as a graph:

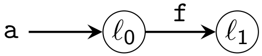
$$G_{\mathsf{CHG}} = \langle \mathit{MemLoc}, \rightarrow, \bar{\rightarrow} \rangle$$

- G_{CHG} describes the *actual* heap contents
- *MemLoc* represents addressable memory locations
  - *Named* variables (`a`)
  - *Unnamed* variables ($\bigcirc$)
- Heap size typically 'unbounded for all practical purposes'
- ($\rightarrow$): Points-to relation from named variables

$$\mathtt{a} \rightarrow \ell_0$$

- ($\bar{\rightarrow}$): Points-to relation from objects/arrays

$$\ell_1 \xrightarrow{f} \ell_1$$



Concrete Heap Graphs (2/2)

- Direct points-to references:

$$(\rightarrow) : \mathit{Var} \rightarrow \mathit{MemLoc}$$

- Language difference:
  - **Java**: *Var* is set of global / local variables and pameters
    - *Disjoint* from *MemLoc*
  - **C/C++**: $\mathit{Var} = \mathit{MemLoc}$
    - Address-of operator (`&`) allows translating variable into *MemLoc*
- Points-to references via fields:

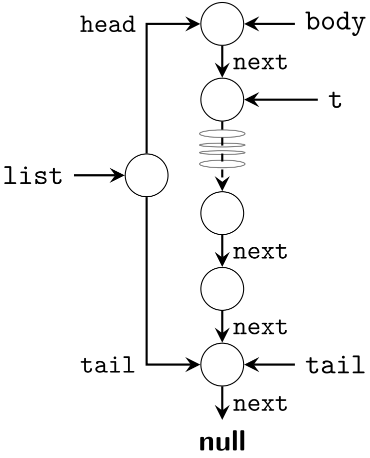
$$(\xrightarrow{-}) : \mathit{MemLoc} \times \mathit{Field} \rightarrow \mathit{MemLoc}$$

- Field labels *Field*:
  - E.g., `x` in '`a.x`' (Java) / '`a->x`' (C/C++)
  - Array indices for '$a[10]$' (i.e., $\mathbb{N} \subseteq \mathit{Field}$)

Example

Example

```
proc makeList(len) {
  tail = new()       //⇐
  tail.next = null   //⇐
  body = tail        //⇐
  while len > 0 {
    t = body         //⇐
    body = new()     //⇐
    body.next = t    //⇐
    len = len - 1
  }
  list = new()       //⇐
  list.head = body   //⇐
  list.tail = tail   //⇐
  return list
}
```



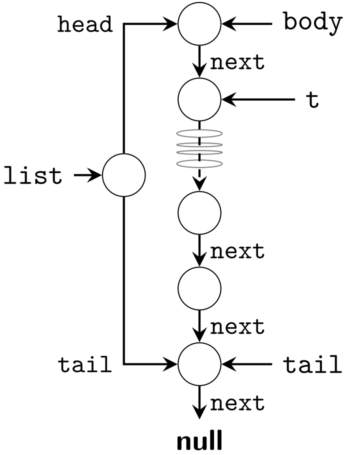
Managing Heap Graphs

- Size of Concrete Heap Graphs is unbounded
- Need summarisation technique to model heap
- **Store-less heap models**:
  - Do away with heap locations
  - Model heap exclusively via *access paths*

```
list.head.next.next
```

- **Store-based heap models**:
  - Keep heap locations explicit
  - Introduce *Summary nodes* that can describe multiple CHG nodes

Store-less Model



- Access path-based equivalences:
  - **Must**: `list.tail` $\overset{\textit{alias}}{=\!=}$ `tail`
  - **Must**: `list.head` $\overset{\textit{alias}}{=\!=}$ `body`
  - **Must**: `body.next` $\overset{\textit{alias}}{=\!=}$ `t`
  - **May**: `body.next`$\star$ $\overset{\textit{alias}}{=\!=}$ `tail`
- Use *regular expressions* to denote repetition
- `body.next`$\star$ $\overset{\textit{alias}}{=\!=}$ `tail` means:

$$
\begin{array}{lcl}
\texttt{body} & \overset{\textit{alias}}{=\!=} & \texttt{tail} \\
\texttt{body.next} & \overset{\textit{alias}}{=\!=} & \texttt{tail} \\
\texttt{body.next.next} & \overset{\textit{alias}}{=\!=} & \texttt{tail} \\
 & \ldots &
\end{array}
$$

- For **May** or **Must** information

Store-based Model

- Concrete Heap Graph (CHG): graph of the program's reality

$$G_{\mathsf{CHG}} = \langle \mathit{MemLoc}, \rightarrow, \bar{\rightarrow} \rangle$$

- Abstract Heap Graph (AHG): approximation of the program's reality

$$G_{\mathsf{AHG}} = \langle \mathcal{P}(\mathit{MemLoc}), \rightharpoonup, \bar{\rightharpoonup} \rangle$$

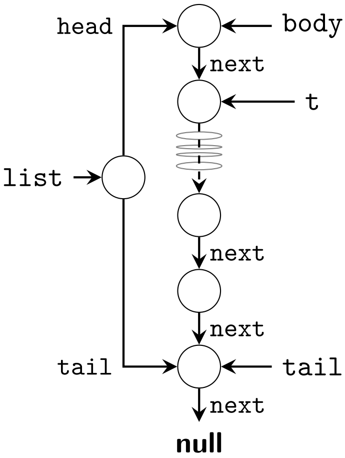
$$\begin{array}{rcl} (\rightharpoonup) & : & \mathcal{P}(\mathit{Var}) \to \mathcal{P}(\mathit{MemLoc}) \\ (\bar{\rightharpoonup}) & : & \mathcal{P}(\mathit{MemLoc}) \times \mathcal{P}(\mathit{Field}) \to \mathcal{P}(\mathit{MemLoc}) \end{array}$$

- Key idea: AHG is *finite* graph that summarises CHG
- Soundness via:

$$\begin{array}{ccccccc} v & \rightarrow & \ell & \text{implies} & \{v\} \cup V' & \rightharpoonup & \{\ell\} \cup L' \\ \ell_0 & \xrightarrow{f} & \ell_1 & \text{implies} & \{\ell_0\} \cup V_0' & \xrightarrow{\{f\} \cup F'} & \{\ell_1\} \cup L_1' \end{array}$$

- Technique: *Summary nodes*

Summary Nodes and Edges



Notation:

- Abstract node $N \subseteq \mathit{MemLoc}$:
  - $|N| = 1$: *precise*:
  - $|N| > 1$: *summary*:
- Consider edge $V \rightarrow\!\!\triangleright\, L$:
  - $|V| = 1$: *precise*:

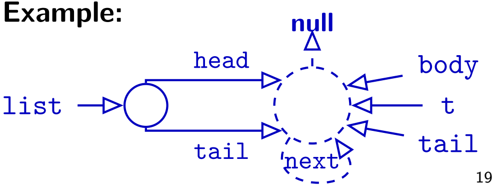
    $V \longrightarrow\!\!\triangleright\, L$

  - $|V| > 1$: *summary*:

    $V \dashrightarrow\!\!\triangleright\, L$

- Analogous for $(\xrightarrow{f}\!\!\triangleright)$

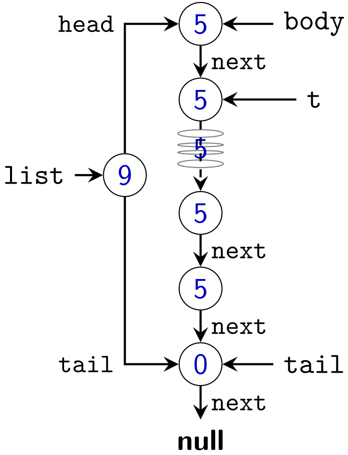
Example:



Summaries from Allocation Sites

Example

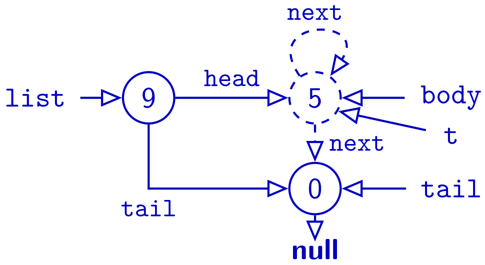
```
proc makeList(len) {
 [0]  tail = new()
 [1]  tail.next = null
 [2]  body = tail
 [3]  while len > 0 {
 [4]    t = body
 [5]    body = new()
 [6]    body.next = t
 [7]    len = len - 1
 [8]  }
 [9]  list = new()
 [10] list.head = body
 [11] list.tail = tail
 [12] return list
}
```



Summaries from Allocation Sites

Example

```
proc makeList(len) {
[0]   tail = new()
[1]   tail.next = null
[2]   body = tail
[3]   while len > 0 {
[4]     t = body
[5]     body = new()
[6]     body.next = t
[7]     len = len - 1
[8]   }
[9]   list = new()
[10]  list.head = body
[11]  list.tail = tail
[12]  return list
}
```



- Summarise *MemLoc* allocated at same program location

Summaries via k-Limiting

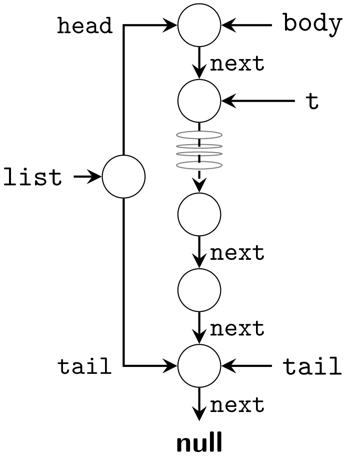
- k-Limiting: bound size
- Examples: Limiting…
  - Access path length
    Example (k=3):

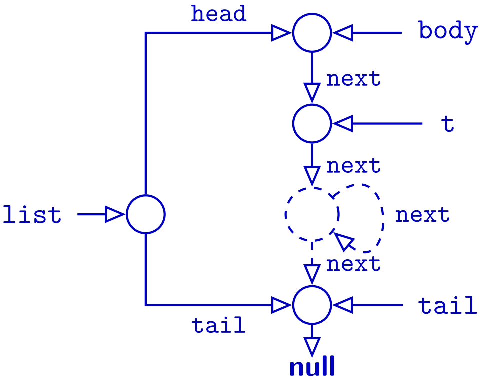
`list.head.next`	$\Rightarrow$	`list.head.next`
`list.head.next.next`	$\Rightarrow$	`list.head.next`$\star$
`list.head.next.next.next`	$\Rightarrow$	`list.head.next`$\star$
`list.head.next.next.val`	$\Rightarrow$	`list.head.(val|next)`$\star$

  - # of ($\multimap$) hops after named variable
  - # of nodes transitively reachable via ($\multimap$) after named variable
  - # of nodes in a loop / function body

  …

Variable-Based Summaries





- Summarise *MemLoc* when not referenced by variables
- For **May** analyses: summarise nodes potentially pointed to by same set of variables

Other Summary Techniques

- General idea: Map $\mathcal{P}(\mathit{MemLoc})$ to finite (manageable!) set
- Can combine different techniques for increased precision
- Other techniques: distinguish heap nodes by:
  - How many edges point to the node?
  - Is the node in a cycle?
  - What is the type of the node? (`ArrayList`, `StringTokenizer`, `File`, ...)

  ...

Design Considerations

- First goal remains: make output finite
- Useful for analysis clients
- Efficient to compute / represent
- When considering flow-sensitive models:
  - Different program locations will have different AHGs
  - Exploit sharing across program locations

Summary of Heap Summaries

- Heap size is unbounded, must summarise
- **Store-less Models**:
  - Use **access paths** to describe memory locations
  - Common in alias analysis
- **Store-based Models**:
  - Use **Abstract Heap Graph** for summarisation
  - Common for finding memory bugs
- Summarisation techniques:
  - **Allocation-Site Based**: summarise nodes allocated at same program point
  - k-**Limiting**: Set bound on some property P: no more than k Ps allowed
  - **Variable-Based**: summarise data not pointed to by variables or pointed to by the same variables (**May** analysis)
  - Many combinations / extensions conceivable

Pointer Operations in C and Java

Referencing

Create location:

C

```c
my_t *p = &var;
p = malloc(8);
```

Java

```java
A a = new A()
```

Dereferencing

Access location:

C

```c
- read -
int x = *ptr;
x = ptr2->fld;
- write -
*ptr = x;
ptr2->fld = x;
```

Java

```java
- read -
int x = a.f;
- write -
a.f = x;
```

Aliasing

Copy pointer:

C

```c
my_t *pa;
pa = pb;
```

Java

```java
A a = b;
```

Pointer Operations

- Three principal pointer operations:
  - **Referencing**:
    - $v := \mathit{address\text{-}of}(\ldots)$
    - Create location ℓ
    - Introduce $v \rightarrow \ell$
  - **Dereferencing**:
    - $x := v.f$
    - Access existing location ℓ
  - **Aliasing**:
    - Pointer/reference variables v_1, v_2
    - $v_2 := v_1$
    - $v_1 \rightarrow \ell \iff v_2 \rightarrow \ell$

Summary

- Points-to anaysis: *approximate* 'v points to location ℓ'

$$v \rightarrow \ell$$

- Analysis must consider:
  - **Referencing**: taking location
  - **Dereferencing**: accessing object at location
  - **Aliasing**: copying location
- Locations ℓ may model different parts of memory:
  - Static variables: uniquely defined
  - Stack-dynamic variables: zero or more copies (recursion!)
  - Heap-dynamic variables:zero or more copies without variable names attached

Steensgaard's Points-To Analysis[1]

- Fast: $O(n\alpha(n,n))$ over variables in program
- Developed to deal with large code bases at AT&T
- Sacrifices Precision
- *Equality-based*
- Intuition:
  Whenever two variables could point to the same memory location, treat them as globally equal

Steensgard: Pointer Operations

- Recall C pointer semantics:
  - `&a`: Address of `a`
  - `*a`: Object pointed to by `a`
  - Converse operators: `*(&a)` $=$ `a`

Steensgard's analysis considers four cases:

	C	**Java**
Referencing	`a = &b`	`a = new A()`
Aliasing	`a = b`	`a = b`
Dereferencing read	`a = *b`	`a = b.f`
Dereferencing write	`*a = b`	`a.f = b`

Constraint Collection

- 'Points-to-set': $pts(v)$ approximates $\{\ell | v \twoheadrightarrow \ell\}$
  - Corresponds to $\{\ell | v \rightharpoonup \ell\}$
- For each statement in program:
  - If **Referencing** (`a = &b`):

$$\ell_{\mathtt{b}} \in pts(\mathtt{a})$$

  - If **Aliasing** (`a = b`):

$$pts(\mathtt{a}) = pts(\mathtt{b})$$

  - If **Dereferencing read** (`a = *b`):

$$\text{for each } \ell \in pts(\mathtt{b}) \implies pts(\mathtt{a}) = pts(\ell)$$

  - If **Dereferencing write** (`*a = b`):

$$\text{for each } \ell \in pts(\mathtt{a}) \implies pts(\mathtt{b}) = pts(\ell)$$

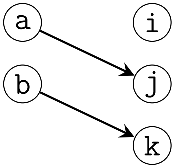
Example

`x = &y`	$\ell_{\mathtt{y}} \in \mathit{pts}(\mathtt{x})$
`x = y`	$\mathit{pts}(\mathtt{x}) = \mathit{pts}(\mathtt{y})$
`x = *y`	for each $\ell \in \mathit{pts}(\mathtt{y})$ $\implies \mathit{pts}(\mathtt{x}) = \mathit{pts}(\ell)$
`*x = y`	for each $\ell \in \mathit{pts}(\mathtt{x})$ $\implies \mathit{pts}(\mathtt{y}) = \mathit{pts}(\ell)$

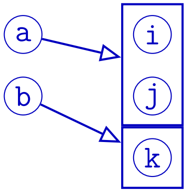
C

```c
int i, j, k;
int* a = &i;
int* b = &k;
a = &j; //⇐
int** p = &a;
int** q = &b;
p = q;
int* c = *q;
```

- **Actual**:



- **Steensgaard**:



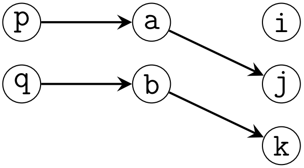
Example

`x = &y`	$\ell_{\mathtt{y}} \in \mathit{pts}(\mathtt{x})$
`x = y`	$\mathit{pts}(\mathtt{x}) = \mathit{pts}(\mathtt{y})$
`x = *y`	for each $\ell \in \mathit{pts}(\mathtt{y}) \implies \mathit{pts}(\mathtt{x}) = \mathit{pts}(\ell)$
`*x = y`	for each $\ell \in \mathit{pts}(\mathtt{x}) \implies \mathit{pts}(\mathtt{y}) = \mathit{pts}(\ell)$

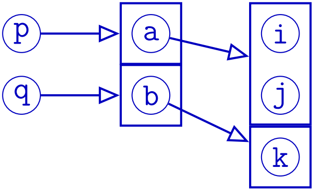
C

```c
int i, j, k;
int* a = &i;
int* b = &k;
a = &j;
int** p = &a;
int** q = &b; //⇐
p = q;
int* c = *q;
```

- **Actual**:



- **Steensgaard**:



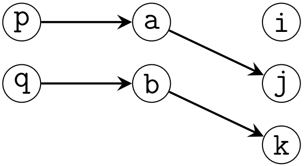
Example

`x = &y`	$\ell_{\mathtt{y}} \in \mathit{pts}(\mathtt{x})$
`x = y`	$\mathit{pts}(\mathtt{x}) = \mathit{pts}(\mathtt{y})$
`x = *y`	for each $\ell \in \mathit{pts}(\mathtt{y})$ $\implies \mathit{pts}(\mathtt{x}) = \mathit{pts}(\ell)$
`*x = y`	for each $\ell \in \mathit{pts}(\mathtt{x})$ $\implies \mathit{pts}(\mathtt{y}) = \mathit{pts}(\ell)$

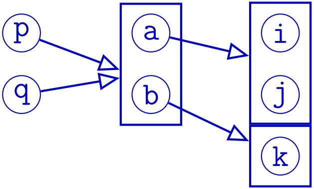
C

```c
int i, j, k;
int* a = &i;
int* b = &k;
a = &j;
int** p = &a;
int** q = &b;
p = q; //⇐
int* c = *q;
```

- **Actual**:



- **Steensgaard**:



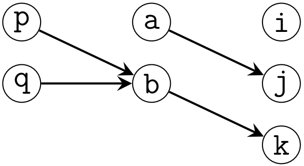
Example

`x = &y`	$\ell_{\texttt{y}} \in \textit{pts}(\texttt{x})$
`x = y`	$\textit{pts}(\texttt{x}) = \textit{pts}(\texttt{y})$
`x = *y`	for each $\ell \in \textit{pts}(\texttt{y})$ $\implies \textit{pts}(\texttt{x}) = \textit{pts}(\ell)$
`*x = y`	for each $\ell \in \textit{pts}(\texttt{x})$ $\implies \textit{pts}(\texttt{y}) = \textit{pts}(\ell)$

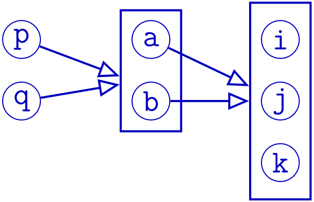
C

```c
int i, j, k;
int* a = &i;
int* b = &k;
a = &j;
int** p = &a;
int** q = &b;
p = q; //⇐
int* c = *q;
```

- **Actual**:



- **Steensgaard**:



When merging: ‘collapse’ children (merge recursively)

Example

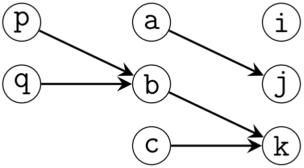
```
x = &y
x = y
x = *y

*x = y
```

$\ell_{\mathtt{y}} \in pts(\mathtt{x})$
$pts(\mathtt{x}) = pts(\mathtt{y})$
for each $\ell \in pts(\mathtt{y})$
$\implies pts(\mathtt{x}) = pts(\ell)$
for each $\ell \in pts(\mathtt{x})$
$\implies pts(\mathtt{y}) = pts(\ell)$

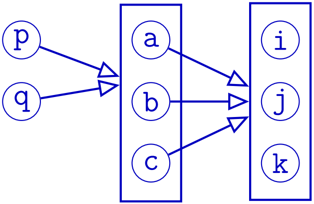
C

```c
int i, j, k;
int* a = &i;
int* b = &k;
a = &j;
int** p = &a;
int** q = &b;
p = q;
int* c = *q; //⇐
```

- **Actual**:



- **Steensgaard**:



When merging: ‘collapse’ children (merge recursively)

Constraint Representation & Solving

- $\widehat{v} \in \mathcal{P}(\mathit{MemLoc})$: set of possible locations of variable v
- Represent with Union-Find data structure (efficient union)
- Collapse child nodes when merging
- Implementing **Referencing** (`a = &b`)
  - $\mathsf{pts}(\widehat{\mathsf{a}})$`.union(`$\widehat{b}$`)`
- Implementing **Aliasing** (`a = b`)
  - $\mathsf{pts}(\widehat{\mathsf{a}})$`.union(`$\mathsf{pts}(\widehat{\mathsf{b}})$`)`
- Implementing **Dereferencing** (`*a = b`)
  - $\mathsf{pts}(\mathsf{pts}(\widehat{\mathsf{a}}))$`.union(`$\mathsf{pts}(\widehat{\mathsf{b}})$`)`

Result is immediate: no further analysis needed

Summary

- Points-to sets $pts(v)$ serve as abstraction over addresses that v can point to
- Steensgaard's points-to analysis:
  - Insensitive to flow, context, fields, ...
- Steensgaard's analysis in practice:
  - Highly efficient
  - Imprecise

Andersen's Points-To Analysis[2]

- Asymptotic performance is $O(n^3)$
- More precise than Steensgaard's analysis
- *Subset-based* (a.k.a. *inclusion-based*)
- Popular as basis for current points-to analyses

Collecting Constraints

- Collect constraints, resolve as needed
- For each statement in program, we record:
  - If **Referencing** (`a = &b`):

$$pts(\mathtt{a}) \supseteq \{\ell_{\mathtt{b}}\}$$

  - If **Aliasing** (`a = b`):

$$pts(\mathtt{a}) \supseteq pts(\mathtt{b})$$

  - If **Dereferencing read** (`a = *b`):

$$pts(\mathtt{a}) \supseteq pts(*\mathtt{b})$$

  - If **Dereferencing write** (`*a = b`):

$$pts(*\mathtt{a}) \supseteq pts(\mathtt{b})$$

Solving Constraints

- We have collected constraints:
  1. $pts(a) \subseteq pts(b)$
  2. $pts(*a) \subseteq pts(b)$
  3. $pts(a) \subseteq pts(*b)$
- Also, we have initial points-to set elements: $\ell \in pts(a)$
- Build directed *inclusion graph* $G_I = \langle MemLoc, E \rangle$
- Edges $a \rightarrow b \in E$ iff one of:
  - $pts(a) \subseteq pts(b)$
  - $a \in pts(v)$ and $pts(*v) \subseteq pts(b)$
  - $pts(a) \subseteq pts(*v)$ and $b \in pts(v)$
- While keeping in mind the following:
  - $(\ell \in pts(a))$ and $(a \rightarrow b \in E) \implies (\ell \in pts(b))$
  - Propagate ℓ along E

Example

C

```c
int i, j, k;
int* a = &i;
int* b = &k;
a = &j;
int** p = &a;
int** q = &b;
p = q;
int* c = *q;
```

- **Actual**:
- **Andersen**:

Example

C

```c
int i, j, k; //⇐
int* a = &i;
int* b = &k;
a = &j;
int** p = &a;
int** q = &b;
p = q;
int* c = *q;
```

- **Actual**:

  i

  j

  k

- **Andersen**:

  i

  j

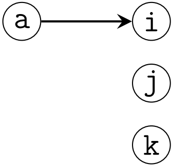
  k

Example

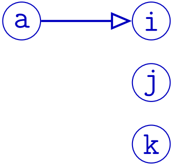
C

```
int i, j, k;
int* a = &i; //⇐
int* b = &k;
a = &j;
int** p = &a;
int** q = &b;
p = q;
int* c = *q;
```

- **Actual**:



- **Andersen**:

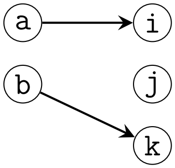


Example

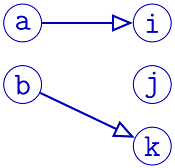
C

```
int i, j, k;
int* a = &i;
int* b = &k; //⇐
a = &j;
int** p = &a;
int** q = &b;
p = q;
int* c = *q;
```

- **Actual**:



- **Andersen**:

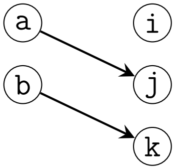


Example

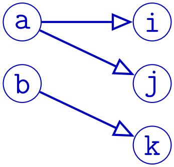
C

```c
int i, j, k;
int* a = &i;
int* b = &k;
a = &j; //⇐
int** p = &a;
int** q = &b;
p = q;
int* c = *q;
```

- **Actual**:



- **Andersen**:

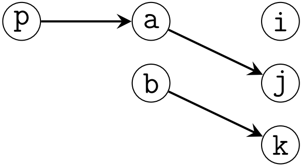


Example

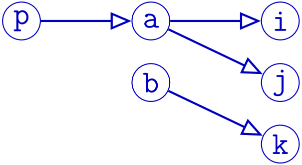
C

```
int i, j, k;
int* a = &i;
int* b = &k;
a = &j;
int** p = &a; //⇐
int** q = &b;
p = q;
int* c = *q;
```

- **Actual**:



- **Andersen**:

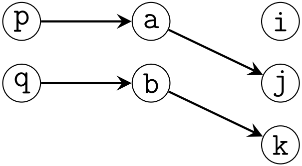


Example

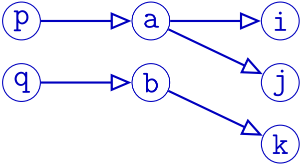
C

```
int i, j, k;
int* a = &i;
int* b = &k;
a = &j;
int** p = &a;
int** q = &b; //⇐
p = q;
int* c = *q;
```

- **Actual**:



- **Andersen**:

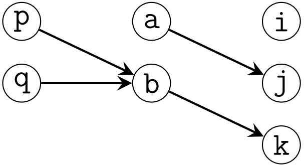


Example

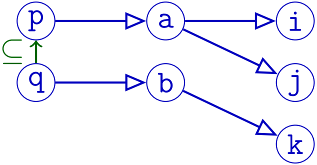
C

```
int i, j, k;
int* a = &i;
int* b = &k;
a = &j;
int** p = &a;
int** q = &b;
p = q; //⇐
int* c = *q;
```

- **Actual**:



- **Andersen**:

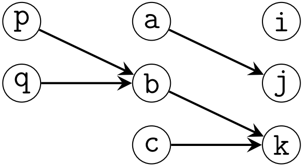


Example

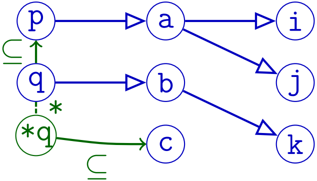
C

```
int i, j, k;
int* a = &i;
int* b = &k;
a = &j;
int** p = &a;
int** q = &b;
p = q;
int* c = *q; //⇐
```

- **Actual**:



- **Andersen**:

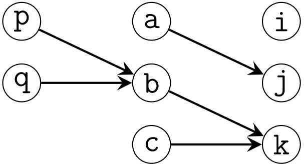


Example

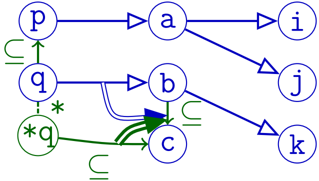
C

```
int i, j, k;
int* a = &i;
int* b = &k;
a = &j;
int** p = &a;
int** q = &b;
p = q;
int* c = *q;
```

- **Actual**:



- **Andersen**:

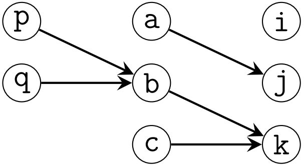


Example

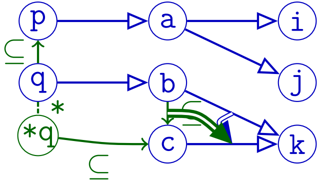
C

```c
int i, j, k;
int* a = &i;
int* b = &k;
a = &j;
int** p = &a;
int** q = &b;
p = q;
int* c = *q;
```

- **Actual**:



- **Andersen**:



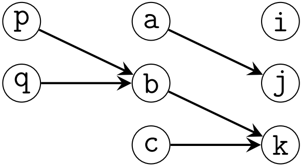
Andersen's algorithm must propagate along inclusion graph

Example

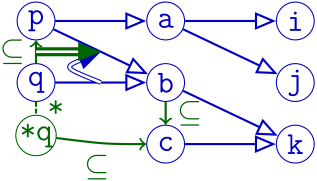
C

```
int i, j, k;
int* a = &i;
int* b = &k;
a = &j;
int** p = &a;
int** q = &b;
p = q;
int* c = *q;
```

- **Actual**:



- **Andersen**:



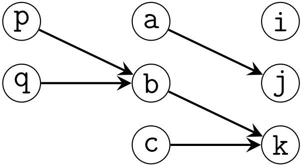
Andersen's algorithm must propagate along inclusion graph

Example

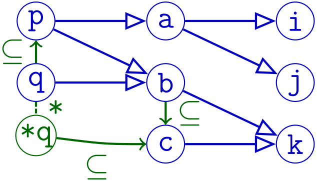
C

```c
int i, j, k;
int* a = &i;
int* b = &k;
a = &j;
int** p = &a;
int** q = &b;
p = q;
int* c = *q;
```

- **Actual**:



- **Andersen**:



Andersen's algorithm must propagate along inclusion graph

Complexity

- Complexity of graph closure: $O(n^3)$
- Traditional assumption about Andersen's analysis
- Recent work observes[3]: Close to $O(n^2)$ if:
  1. Few statements dereference each variable
  2. Control flow graphs not too complex
  - *Both conditions are common in practical programs*

Summary

- Andersen's analysis:
  - Subset-based
  - Builds inclusion graph for propagating memory locations along subset constraints
  - $O(n^3)$ worst-case behaviour
  - Closer to $O(n^2)$ in practice
  - More precise than Steensgaard's analysis
  - Less scalable than Steensgaard's analysis

The Call Graph

```c
int main(int argc,
         char *argv) {
    if (argc > 1) {
        f(argv[0]);
    }
    g();
    return 0;
}
```

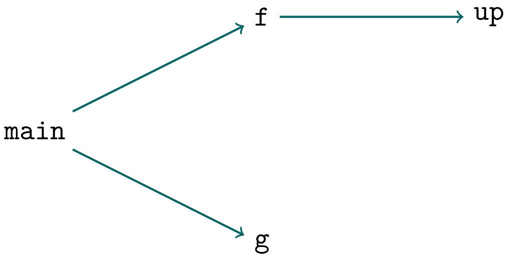
```c
void f(char *s) {
    for (char *p = s; *p; p++) {
        *p = up(*p);
    }
    puts(s);
}
```

```c
char up(char c) {
    if (c >= 'a' && c <= 'z') {
        return c - ('a' - 'A');
    }
    return c;
}
```

```c
void g(void) {
    puts("Hello, World!");
}
```

The Call Graph

- $G_{\text{call}} = \langle P, E_{\text{call}} \rangle$
- Connects procedures from P via call edges from E_{call}
- 'Which procedure can call which other procedure?'
- Often refined to:
  'Which *call site* can call which procedure?'
- Used by program analysis to find procedure call targets



Finding Calls and Targets

```java
class Main {
  public void
  main(String[] args) {
    A[] as = { new A(), new B() };
    for (A a : as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

```java
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

```java
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

```java
class C extends A {
  @Override
  public String
  g() { return "A"; }
}
```

```java
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Finding Calls and Targets

```
class Main {
  public void
  main(String[] args) {
    A[] as = { new A(), new B() };
    for (A a : as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

a.f

a.g

a2.g

```
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

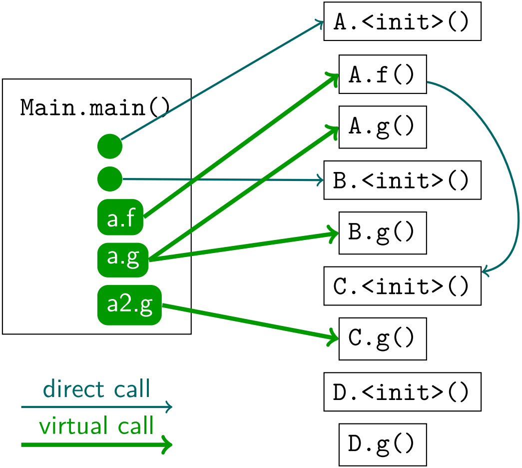
```
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

```
class C extends A {
  @Override
  public String
  g() { return "A"; }
}
```

```
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Dynamic Dispatch: Call Graph

Challenge: Computing the precise call graph:



Summary

- **Call Graphs** capture which procedure calls which other procedure
- For program analysis, further specialised to map:

$$\text{Callsite} \rightarrow \text{Procedure}$$

- Direct calls: straightforward
- Virtual calls (dynamic dispatch):
  - Multiple targets possible for call
  - Not straightforward

Callgraphs with Points-to Data

```
class A {
 public A
 f() {
  return new C();
 }
}
```

```
class B extends A {
 public A
 f() {
  return new A();
 }
}
```

```
class C extends A {
 public A
 f() {
  return new B();
 }
}
```

```
A a = new A();
a = a.f();
a = a.f();
```

- Precision of call graph affects quality of all interprocedural analyses
  - IFDS, IDE
  - Points-to analyses
- Idea: Use points-to analysis to determine *dynamic* type of objects
  - More precise virtual call resolution!
- **Problem**: Mutual dependency between call-graph and points-to analysis!

Review

- Pointer Analysis
  - Points-To Analysis
  - Alias Analysis
- Concrete Heap Graphs
- Abstract Heap Graphs
- Access Paths
- Heap Summarisation
  - Call-site
  - Variable-based
  - k-Limiting
- Steensgard's Analysis
- Andersen's Analysis
- Call graphs

To be continued...

Next week:

- Program Analysis with Datalog