LUND UNIVERSITY

EDAP15: Program Analysis

POINTER ANALYSIS 1

Christoph Reichenbach

Welcome back!

- Final homework assignment will require *docker* or similar.
- More details soon.
- Student representative?

Questions?

Variables and Memory Binding

Teal-2

```
var x := 1;

fun f(y) = {
  var z := y;
  return x + z;
}
```

Java

```
static int x = 1;

static int f(int y) {
  int z = y;
  return x + z;
}
```

- So far: Variables with one of
  - Static memory binding (x)
  - Stack-dynamic memory binding (y, z)
- Key property:
  - Need only track *one* memory binding per **VarDecl**
    - *Static*: Exactly one memory binding per VarDecl
    - *Stack-dynamic*: Zero or more, but only one visible at a time
      Have memory binding iff variable in scope

Heap-Dynamic Memory

- *Heap-Dynamic* memory binding due to:
  - Explicit allocation: **new**, **malloc**
  - Implicit allocation: `[1, 2, 3]` in Teal, Python
- Creates storage locations during execution:
  - Array elements, datastructure fields: also variables!
- Different names can bind to same memory storage location:

Teal

```
var a := [1, 4];
var b := a;
// Different names refer to same memory:
a[1] := 7;
print(b[1]); // prints '7'
// Same name can bind to different memory:
b := [2, 2];
print(b[1]); // prints '2'
```

Teal-2

Teal-2

```
type MyType(name: string, value: int);

var x := new MyType("foo", 0);
x.value := 1;
print(x.field);
```

- Teal-2 adds **User-defined types**:
  - Type definition
  - Allocation
  - Assigning to fields
  - Reading from fields

Lecture Overview

Foundations

Static Analysis

Dynamic Analysis

Properties

Control Flow

01 Foundations

02 Constructing Program Analyses in JastAdd

03 04 Types

05 06 07 Data Flow

08 09 **Memory**

05 Intraprocedural

10 Interprocedural

11 Indirect

12 Instrumentation

13 Analysis

14 Review

Our Memory Modelling Until Now

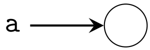
- Our analyses so far have considered:
  - Static Variables
  - Local (stack-dynamic) Variables
  - (Stack-dynamic) parameters

Missing: heap variables!

Example Program

Example

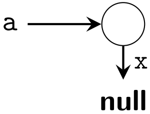
```
a := new();    // ⇐
a.x := null;
b := a;
b.x := new();
a.x.y := 1;
c := new();
c.x := new();
c.x.x := a;
c := a.x;
// A
```



Example Program

Example

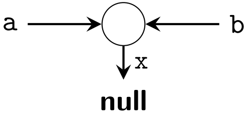
```
a := new();
a.x := null;  // ⇐
b := a;
b.x := new();
a.x.y := 1;
c := new();
c.x := new();
c.x.x := a;
c := a.x;
// A
```



Example Program

Example

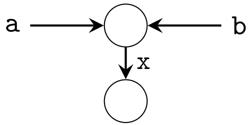
```
a := new();
a.x := null;
b := a;         // ⇐
b.x := new();
a.x.y := 1;
c := new();
c.x := new();
c.x.x := a;
c := a.x;
// A
```



Example Program

Example

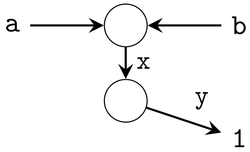
```
a := new();
a.x := null;
b := a;
b.x := new(); // ⇐
a.x.y := 1;
c := new();
c.x := new();
c.x.x := a;
c := a.x;
// A
```



Example Program

Example

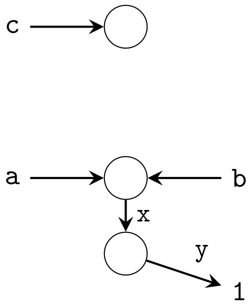
```
a := new();
a.x := null;
b := a;
b.x := new();
a.x.y := 1;    // ⇐
c := new();
c.x := new();
c.x.x := a;
c := a.x;
// A
```



Example Program

Example

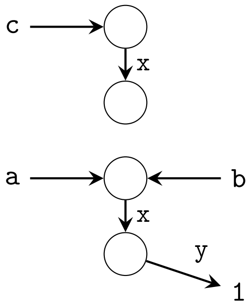
```
a := new();
a.x := null;
b := a;
b.x := new();
a.x.y := 1;
c := new();    // ⇐
c.x := new();
c.x.x := a;
c := a.x;
// A
```



Example Program

Example

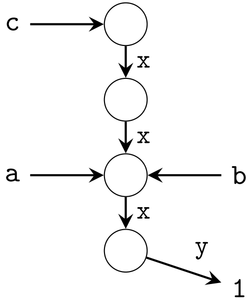
```
a := new();
a.x := null;
b := a;
b.x := new();
a.x.y := 1;
c := new();
c.x := new(); // ⇐
c.x.x := a;
c := a.x;
// A
```



Example Program

Example

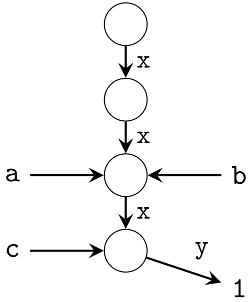
```
a := new();
a.x := null;
b := a;
b.x := new();
a.x.y := 1;
c := new();
c.x := new();
c.x.x := a;    // ⇐
c := a.x;
// A
```



Example Program

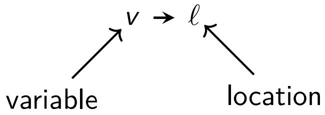
Example

```
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
```

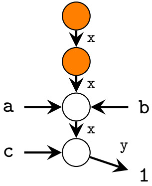


Concrete Heap Graph

"v points to ℓ"



- 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`:

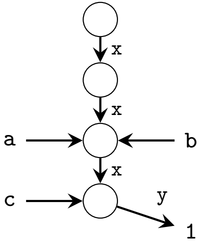
- `a` and `b` represent the same object

⇒ `a` and `b` are *aliased*

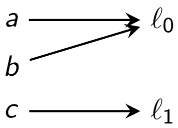
$$a \overset{alias}{=\!=} b$$

⇒ `a.x` and `b.x` are *aliased*

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



Pointer Analysis



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

$$a \rightarrow \ell_0$$

- *Alias Analysis*:
  - Analyse *address sharing*
  - Compute *equivalence relation* ($\overset{alias}{=\!=}$) between variables
  - Which *pair/set of variables* may/must point to the same address?

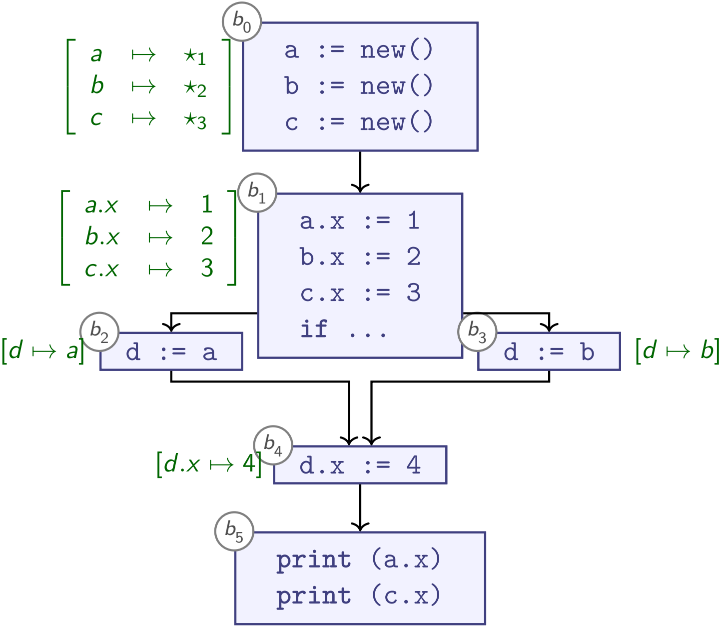
$$a \overset{alias}{=\!=} b$$

Summary: Pointer Analysis

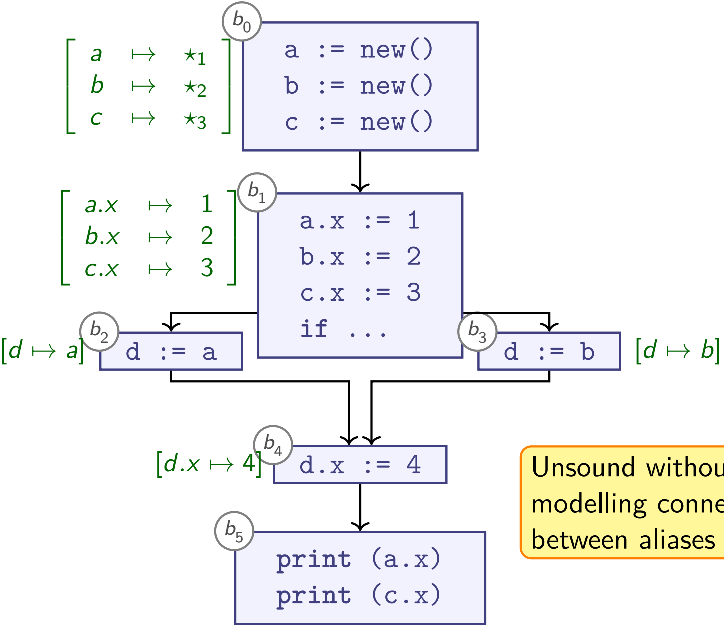
- Class of analyses to model dynamic heap allocation
- **Points-To Analysis**: computes mapping
  - From *variables*
  - To *pointees* (other variables)
  - More general than Alias Analysis
- **Alias Analysis**: computes
  - *Sharing information* between variables
  - Implicitly produced by points-to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

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

Dataflow with Alias Information

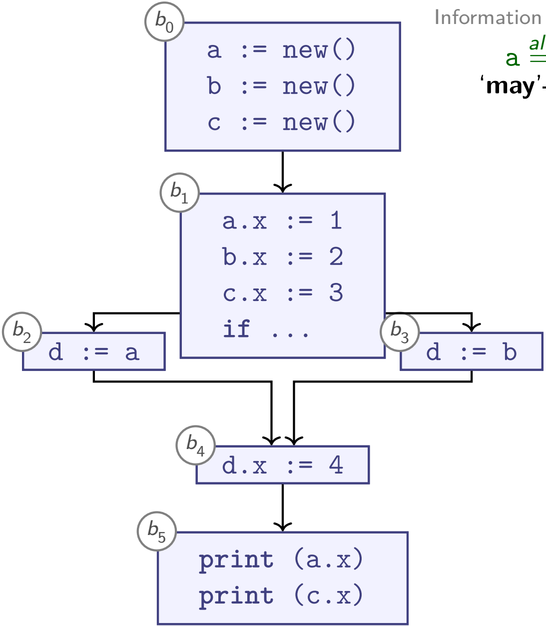


Dataflow with Alias Information



Unsound without modelling connection between aliases

Dataflow with Alias Information

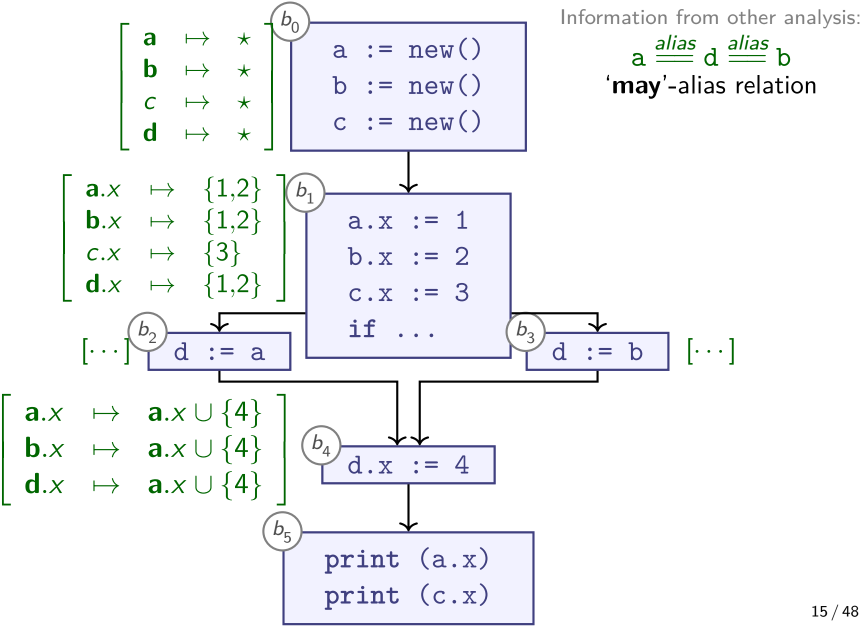


Information from other analysis:

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

‘**may**’-alias relation

Dataflow with Alias Information

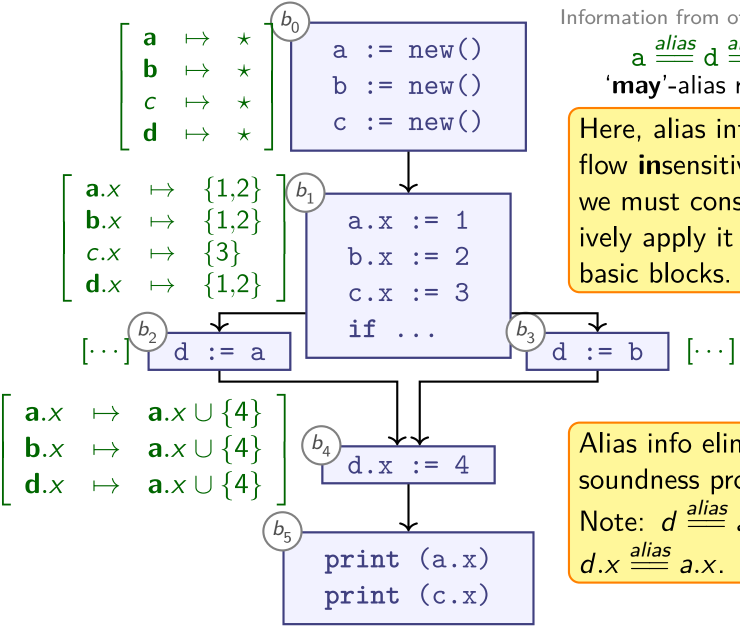


Dataflow with Alias Information

Information from other analysis:

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

‘**may**’-alias relation



Here, alias info is flow **in**sensitive, so we must conservatively apply it to all basic blocks.

Alias info eliminates soundness problem. Note: $d \overset{alias}{=\!=} a$ implies $d.x \overset{alias}{=\!=} a.x$.

Dataflow + Aliases

- Aliasing affects shared fields:

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

- Use aliasing knowledge in one of these ways:

1. Multiply *updates* for each alias:

b_u
```
a.x := 4
```

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

Using MAY alias info means that we might or might not update the aliased object.

2. Multiply *reads* for each alias:

b_u
```
v := a.x
```

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

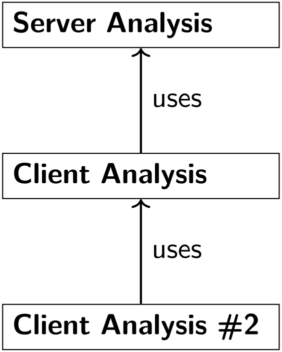
3. Replace aliased paths by single representative (e.g., **a** represents both **d** and **a**):

b_u
```
v := d.z
d.x := a.y
```

$$\begin{bmatrix} v & \mapsto & \mathbf{a}.z \\ \mathbf{a}.x & \mapsto & \mathbf{a}.y \end{bmatrix}$$

Collaboration in Program Analysis

Observation:



e.g., *Alias*

e.g., *Reaching Definitions*

e.g., *Array Out Of Bounds check*

Analyses often form pipeline structures

Compute Aliases during Dataflow?

- Previously: 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` $\mapsto z$
  - Must extract both y, z from $\mathbf{in}_b$ to compute update
    - y, z may have aliases
    - *Non-distributive in general*

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:

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

  - 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

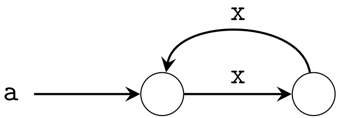
Concrete Heap Graphs (1/3)

Describe heap as a graph:

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

- G_{CHG} describes *actual* heap contents
- *MemLoc* are addressable memory locations
  - *Named* variables (`a`)
  - *Unnamed* variables (◯)
- Heap size typically 'unbounded for all practical purposes'

```
a := new Obj();
a.x := new Obj();
a.x.x := a;
```



Concrete Heap Graphs (2/3)

- Direct points-to references:

$$(\rightarrow) \subseteq \mathit{MemLoc} \times \mathit{MemLoc}$$

- At most one ℓ' s.th. $\ell \rightarrow \ell'$
  - Partial function:

$$(\rightarrow) : \mathit{MemLoc} \to \mathit{MemLoc} \cup \{\mathbf{null}\}$$

- Points-to references via fields:

$$(\overset{\blacksquare}{\rightarrow}) : \mathit{MemLoc} \times \mathit{Field} \to \mathit{MemLoc}$$

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

Concrete Heap Graphs (3/3)

- Direct points-to references:

$$(\,\twoheadrightarrow\,) : \mathit{MemLoc} \to \mathit{RefLoc} \cup \{\mathbf{null}\}$$

- Language difference:
  - **Java/Teal**: *Var* is set of global / local variables and parameters
    - $\mathit{Var} \cap \mathit{RefLoc} = \emptyset$
    - $\mathit{MemLoc} = \mathit{Var} \cup \mathit{RefLoc}$
  - **C/C++**: $\mathit{Var} = \mathit{RefLoc} = \mathit{MemLoc}$
    - Address-of operator (`&`) allows translating variable into *MemLoc*
    - $\mathit{Var} \subseteq \mathit{RefLoc}$
    - $\mathit{MemLoc} = \mathit{RefLoc}$

Example

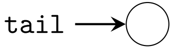
Teal-2

```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```

Example

Teal-2

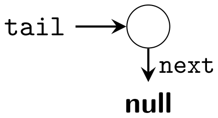
```
fun makeList(len) {
  var tail := new N(); //⇐
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

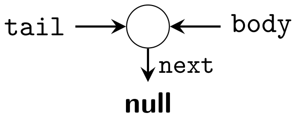
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;   //⇐
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

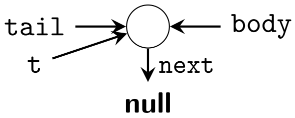
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;        //⇐
  while len > 0 {
    var t := body;
    body := new N();
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

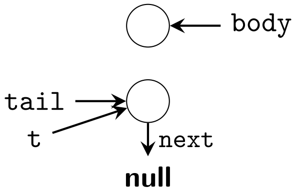
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;       //⇐
    body := new N();
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

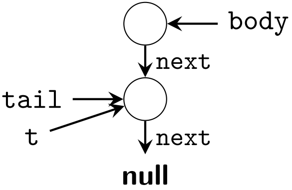
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();    //⇐
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

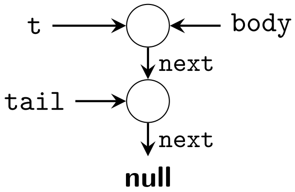
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();
    body.next := t;      //⇐
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

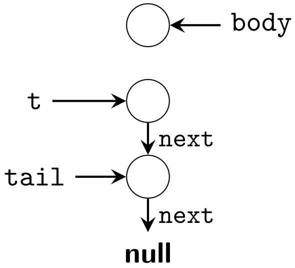
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;        //⇐
    body := new N();
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

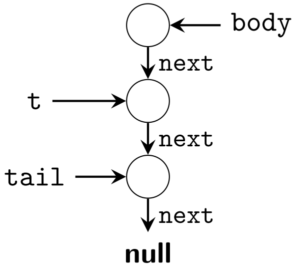
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();    //⇐
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

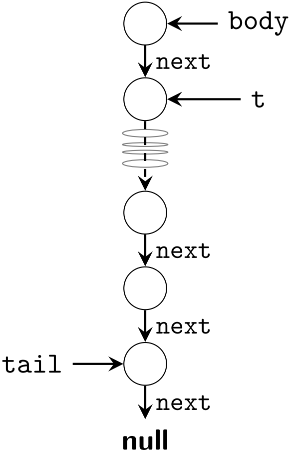
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();
    body.next := t;      //⇐
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

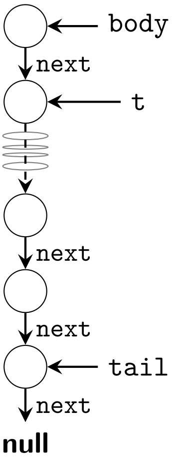
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;        //⇐
    body := new N();      //⇐
    body.next := t;       //⇐
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}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



Example

Teal-2

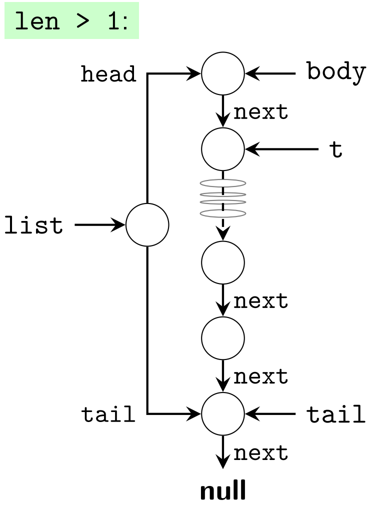
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```

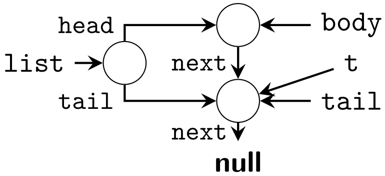


Example

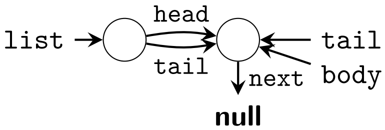
Teal-2

```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```

len = 1:



len = 0:



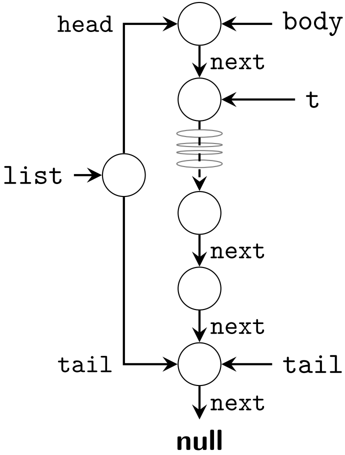
Managing Heap Graphs

- Size of Concrete Heap Graphs is unbounded
- Different parameters $\implies$ different Concrete Heap Graphs

- **Store-less heap models**:
  - Hide heap locations
  - Model heap via *access paths*

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

Store-less Model



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

`body`	$\overset{alias}{=\!=}$	`tail`
`body.next`	$\overset{alias}{=\!=}$	`tail`
`body.next.next`	$\overset{alias}{=\!=}$	`tail`
	$\ldots$	

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

Summary

- **Concrete Heap Graph** (CHG) describes actual heap layout during execution
- CHG is unbounded, must summarise to analyse
- **Store-less Models**:
  - Use **access paths** to describe memory locations
  - Common in alias analysis

Managing Heap Graphs

- Size of Concrete Heap Graphs is unbounded
- Different parameters $\implies$ different Concrete Heap Graphs

- **Store-less heap models**:
  - Hide heap locations
  - Model heap 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-based Model

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

$$G_{\text{CHG}} = \langle \mathit{MemLoc}, \rightarrow, \overset{\blacksquare}{\rightarrow} \rangle$$

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

$$G_{\text{AHG}} = \langle \mathcal{P}(\mathit{MemLoc}), \rightharpoonup, \overset{\blacksquare}{\rightharpoonup} \rangle$$

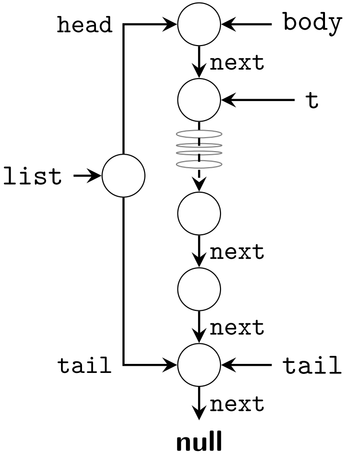
$$\begin{aligned} (\rightharpoonup) &: \mathcal{P}(\mathit{MemLoc}) \to \mathcal{P}(\mathit{MemLoc}) \\ (\overset{\blacksquare}{\rightharpoonup}) &: \mathcal{P}(\mathit{MemLoc}) \times \mathcal{P}(\mathit{Field}) \to \mathcal{P}(\mathit{MemLoc}) \end{aligned}$$

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

$$\begin{array}{ccccccc} v & \rightarrow & \ell & \text{implies} & \{v\} \cup V' & \rightharpoonup & \{\ell\} \cup L' \\ \ell_0 & \overset{f}{\rightarrow} & \ell_1 & \text{implies} & \{\ell_0\} \cup L'_0 & \overset{\{f\}\cup F'}{\rightharpoonup} & \{\ell_1\} \cup L'_1 \end{array}$$

- 'Any CHG edge is represented by (at least) one AHG edge'

Summary Nodes and Edges



Notation:

- Abstract node $N \subseteq MemLoc$:
  - $|N| = 1$: *precise*:
  - $|N| > 1$: *summary*:
- Consider edge $V \rightarrow L$:
  - $|V| = 1$: *precise*:

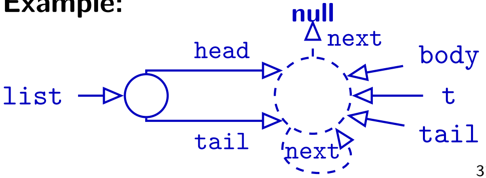
    $V \longrightarrow L$

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

    V - - ▷ L

- Analogous for $(\xrightarrow{\blacksquare} f)$

Example:



Summary

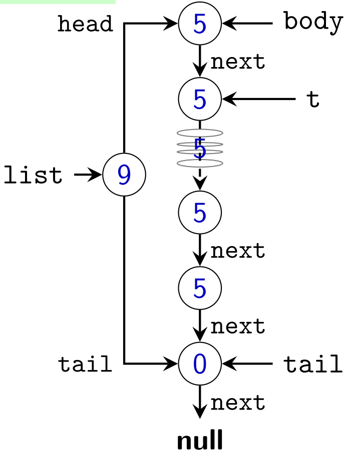
- **Store-based Models**:
  - Use **Abstract Heap Graph** to summarise *Concrete Heap Graph*
  - Common for finding memory bugs
  - Represents NFA
    - Equivalent to regular expressions

Summaries from Allocation Sites

Teal-2

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

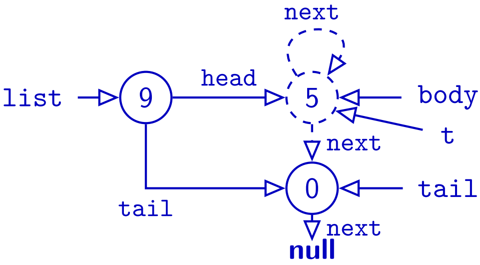
len > 1:



Summaries from Allocation Sites

Teal-2

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

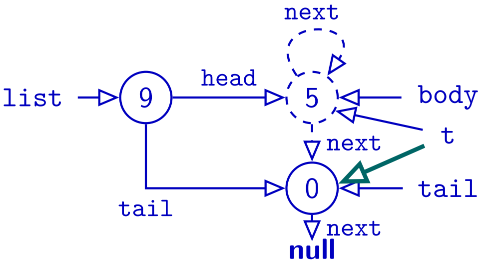


- Summarise *MemLoc* allocated at same program location

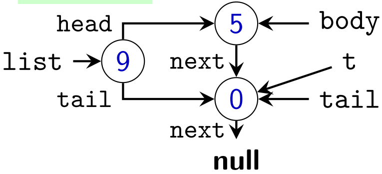
Summaries from Allocation Sites

Teal-2

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



len = 1:

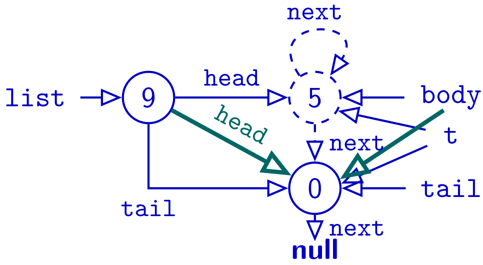


- Summarise *MemLoc* allocated at same program location
- Nodes can have multiple outgoing arrows

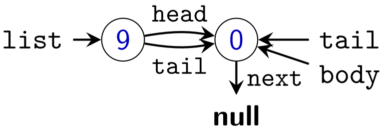
Summaries from Allocation Sites

Teal-2

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



len = 0:

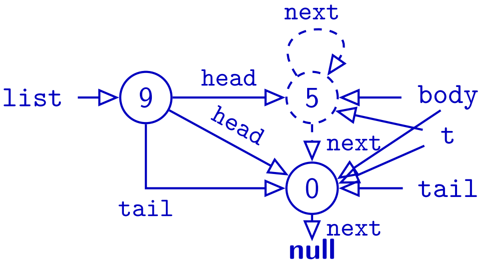


- Summarise *MemLoc* allocated at same program location
- Nodes can have multiple outgoing arrows

Summaries from Allocation Sites

Teal-2

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



- Summarise *MemLoc* allocated at same program location
- Nodes can have multiple outgoing arrows

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

  ...

Other Summary Techniques

- General idea: Partition *MemLoc* into 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

- *Store-less Models*:
  - Common in alias analysis
- **Store-based Models**:
  - Use **Abstract Heap Graph** to summarise *Concrete Heap Graph*
  - Common for finding memory bugs
  - NFA representation $\mapsto$ regular expressions for Access Paths
- Summarisation techniques:
  - **Allocation-Site Based**: summarise nodes allocated at same point in program
  - k-**Limiting**: Set bound on some property P: no more than k Ps allowed
  - Many combinations / extensions conceivable

Pointer Operations

Referencing

$$\texttt{a} \rightarrow \ell$$

Create, point to location:

Teal-2

```
a := new A();
a := [...];
```

Dereferencing

$$\texttt{a} \rightarrow \ell \xrightarrow{f} \ell'$$

Access location:

Teal-2

```
  – read –
...  := a.f;
...  := a[i];
  – write –
a.f  := ...;
a[i]  := ...;
```

Aliasing

$$\texttt{b} \rightarrow \ell \iff \texttt{a} \rightarrow \ell$$

Copy pointer:

Teal-2

```
a := b;
```

Pointer Operations Across Languages

	C	Java	Teal
Referencing	`a = &b`	`a = new A()`	`a := new A()` `a := [..., b, ...]`
Aliasing	`a = b`	`a = b`	`a := b`

Dereferencing read	`a = b->f`
	`a = b[i]`
	`a = *b`
		`a = b.f`
		`a = b[i]`
			`a := b.f`
			`a := b[i]`

Dereferencing write	`a->f = b`
	`a[i] = b`
	`*a = b`
		`a.f = b`
		`a[i] = b`
			`a.f := b`
			`a[i] := b`

Summary

- Points-to anaysis: compute Abstract Heap Graph by *approximating*

$$v \rightarrow \ell$$

- Analysis must consider:
  - **Referencing**: taking (fresh) location
    - In languages like C/C++, code can also reference locations of stack/global variables
  - **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

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

B. Steensgaard. 'Points-to analysis in almost linear time.' In Proceecings of POPL '96, pages 32–41. ACM Press, 1996.

Distinguishing Field Names?

- For simplicity, don't distinguish field names:
- `a.□` instead of `a.f` or `a.g`
- Will discuss consequences of this simplification shortly

Constraint Collection

- 'Points-to-set': $pts(v)$ approximates $\{\ell | v \rightarrow \ell\}$
  - $pts(v) = \{\ell | v \twoheadrightarrow \ell\}$
- For each statement in program:
  - If **Referencing** (`a := new`$_{\ell_b}$ `...`) (allocation site ℓ_b):

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

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

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

  - If **Dereferencing read** (`a := b.`$\square$):

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

  - If **Dereferencing write** (`a.`$\square$ `:= b`):

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

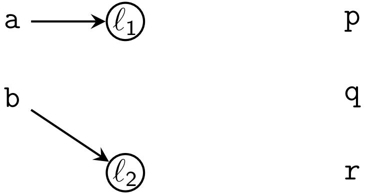
Example

$\Rightarrow$ `x := new`$_{\ell_z}$	$\ell_z \in \mathit{pts}(\mathtt{x})$
`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 $\ell \in \mathit{pts}(\mathtt{x})$ $\implies \mathit{pts}(\mathtt{y}) = \mathit{pts}(\ell)$

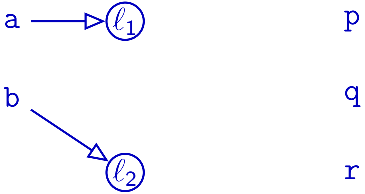
Teal

```
var a := new_ℓ1();
var b := new_ℓ2(); //⇐
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

- **Actual**:



- **Steensgaard**:



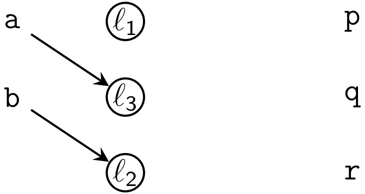
Example

⇒ `x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
`x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
`x := y.□`	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
`x.□ := y`	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

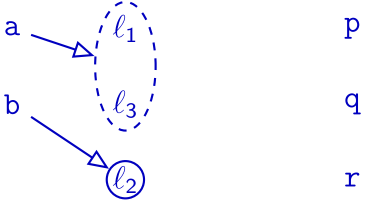
Teal

```
var a := newℓ1();
var b := newℓ2();
a := newℓ3();          //⇐
var p := newℓ4();
p.n := a;
var q := newℓ6();
q.n := b;
p := q;
var r := q.n;
```

- **Actual**:



- **Steensgaard**:



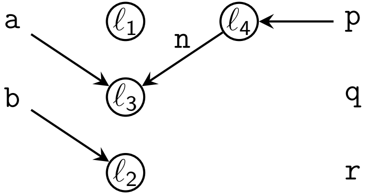
Example

`x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
`x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
`x := y.□`	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
⇒ `x.□ := y`	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

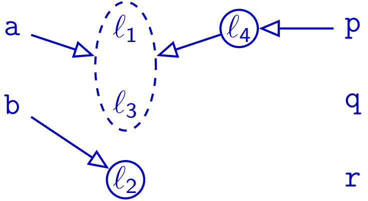
Teal

```
var a := newℓ1();
var b := newℓ2();
a := newℓ3();
var p := newℓ4();
p.n := a;          //⇐
var q := newℓ6();
q.n := b;
p := q;
var r := q.n;
```

- **Actual**:



- **Steensgaard**:



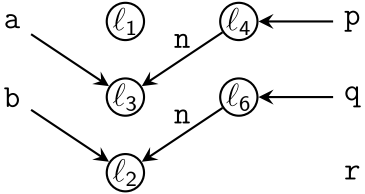
Example

	x := new$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
	x := y	$pts(\mathtt{x}) = pts(\mathtt{y})$
	x := y.□	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
⇒	x.□ := y	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

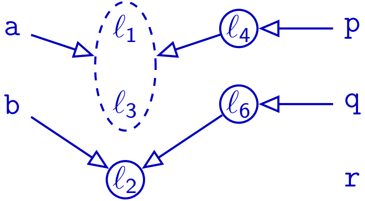
Teal

```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;              //⇐
p := q;
var r := q.n;
```

- **Actual**:



- **Steensgaard**:



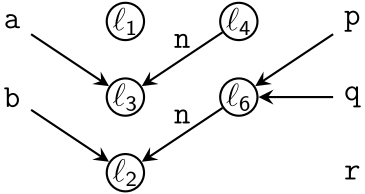
Example

	`x := new`$_{\ell_z}$	$\ell_z \in \mathit{pts}(\mathtt{x})$
$\Rightarrow$	`x := y`	$\mathit{pts}(\mathtt{x}) = \mathit{pts}(\mathtt{y})$
	`x := y.`$\square$	for each $\ell \in \mathit{pts}(\mathtt{y})$ $\implies \mathit{pts}(\mathtt{x}) = \mathit{pts}(\ell)$
	`x.`$\square$ `:= y`	for each $\ell \in \mathit{pts}(\mathtt{x})$ $\implies \mathit{pts}(\mathtt{y}) = \mathit{pts}(\ell)$

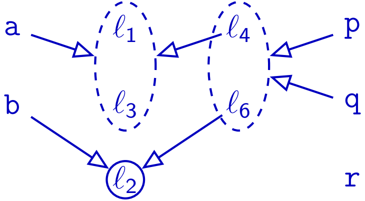
Teal

```
var a := newℓ1();
var b := newℓ2();
a := newℓ3();
var p := newℓ4();
p.n := a;
var q := newℓ6();
q.n := b;
p := q;                  //⇐
var r := q.n;
```

- **Actual**:



- **Steensgaard**:



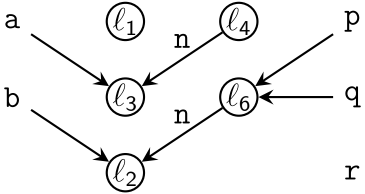
Example

`x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
⇒ `x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
`x := y.`□	for each $\ell \in pts(\mathtt{y})$
	$\implies pts(\mathtt{x}) = pts(\ell)$
`x.`□ `:= y`	for each $\ell \in pts(\mathtt{x})$
	$\implies pts(\mathtt{y}) = pts(\ell)$

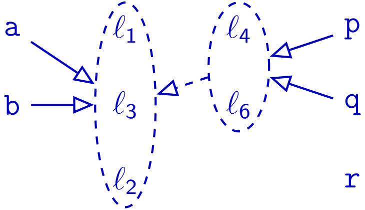
Teal

```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;                //⇐
var r := q.n;
```

- **Actual**:



- **Steensgaard**:



When merging: 'collapse' children (merge recursively)

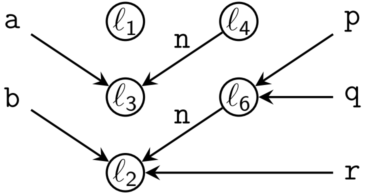
Example

	`x := new`$_{\ell_z}$	$\ell_z \in \mathit{pts}(\mathtt{x})$
	`x := y`	$\mathit{pts}(\mathtt{x}) = \mathit{pts}(\mathtt{y})$
$\Rightarrow$	`x := y.□`	for each $\ell \in \mathit{pts}(\mathtt{y})$ $\implies \mathit{pts}(\mathtt{x}) = \mathit{pts}(\ell)$
	`x.□ := y`	for each $\ell \in \mathit{pts}(\mathtt{x})$ $\implies \mathit{pts}(\mathtt{y}) = \mathit{pts}(\ell)$

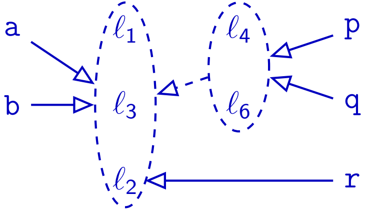
Teal

```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;         //⇐
```

- **Actual**:



- **Steensgaard**:



When merging: ‘collapse’ children (merge recursively)

Summary

- Points-to sets $pts(v) = \{\ell | v \rightarrow\!\!\triangleright \ell\}$
  - Partitions abstract heap locations
  - *pts* thus equivalent to Abstract Heap Graph
- Steensgaard's points-to analysis:
  - special case of *type analysis*
- Steensgaard's analysis in practice:
  - Highly efficient when implemented with UNION-FIND
  - Less precise than other commonly-used analyses

Outlook

- More pointers on Monday
- Class representative needed: please e-mail me

`http://cs.lth.se/EDAP15`