



LUND
UNIVERSITY

EDAP15: Program Analysis

POINTER ANALYSIS 1

Christoph Reichenbach



Welcome back!

- ▶ Guest lecture by Patrik Åberg, Ericsson, on the 27th (half of the lecture)
- ▶ Final homework assignment will require *docker* or *podman*
- ▶ More details soon.

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

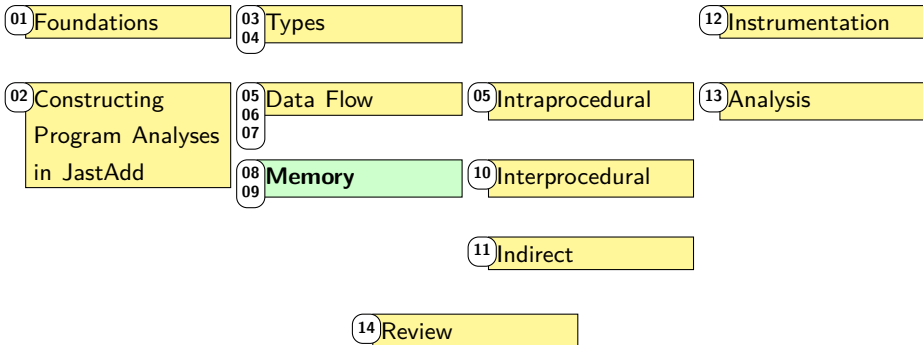
Foundations

Static Analysis

Dynamic
Analysis

Properties

Control Flow



Our Memory Modelling Until Now

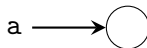
- ▶ Our analyses so far have considered:
 - ▶ Variables with static memory binding
 - ▶ Variables with stack-dynamic memory binding
 - ▶ Local variables
 - ▶ Parameters

Missing: heap variables!

Example Program

Example

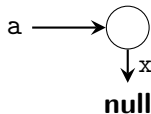
```
a := new();    // ⇐  
a.x := null;  
b := a;  
b.x := new();  
a.x.y := 1;  
c := new();  
c.x := new();  
c.x.x := a;  
c := a.x;  
// A
```



Example Program

Example

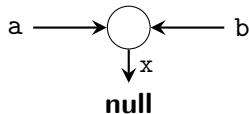
```
a := new();  
a.x := null;  // ⇐  
b := a;  
b.x := new();  
a.x.y := 1;  
c := new();  
c.x := new();  
c.x.x := a;  
c := a.x;  
// A
```



Example Program

Example

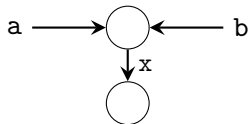
```
a := new();  
a.x := null;  
b := a;      // ⇐  
b.x := new();  
a.x.y := 1;  
c := new();  
c.x := new();  
c.x.x := a;  
c := a.x;  
// A
```



Example Program

Example

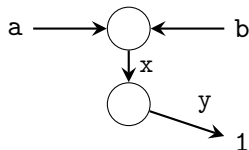
```
a := new();  
a.x := null;  
b := a;  
b.x := new(); // ⇐  
a.x.y := 1;  
c := new();  
c.x := new();  
c.x.x := a;  
c := a.x;  
// A
```



Example Program

Example

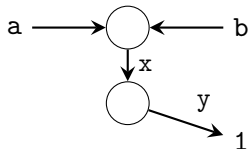
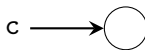
```
a := new();  
a.x := null;  
b := a;  
b.x := new();  
a.x.y := 1;    // ←  
c := new();  
c.x := new();  
c.x.x := a;  
c := a.x;  
// A
```



Example Program

Example

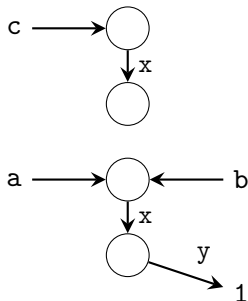
```
a := new();  
a.x := null;  
b := a;  
b.x := new();  
a.x.y := 1;  
c := new();    // ⇐  
c.x := new();  
c.x.x := a;  
c := a.x;  
// A
```



Example Program

Example

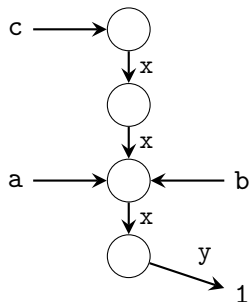
```
a := new();  
a.x := null;  
b := a;  
b.x := new();  
a.x.y := 1;  
c := new();  
c.x := new(); // ←  
c.x.x := a;  
c := a.x;  
// A
```



Example Program

Example

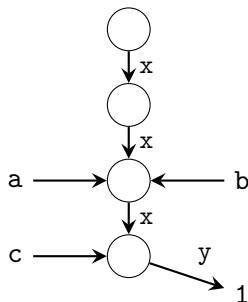
```
a := new();  
a.x := null;  
b := a;  
b.x := new();  
a.x.y := 1;  
c := new();  
c.x := new();  
c.x.x := a;    // ⇐  
c := a.x;  
// A
```



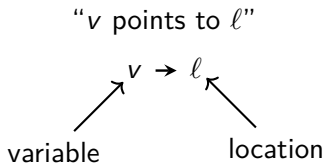
Example Program

Example

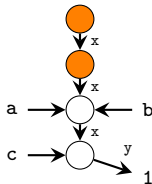
```
a := new();  
a.x := null;  
b := a;  
b.x := new();  
a.x.y := 1;  
c := new();  
c.x := new();  
c.x.x := a;  
c := a.x;      // ⇐  
// A
```



Concrete Heap Graph



- ▶ 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

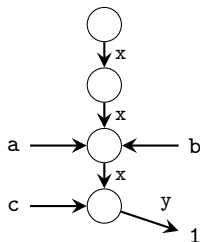
```
a := new();  
a.x := null;  
b := a;  
b.x := new();  
a.x.y := 1;  
c := new();  
c.x := new();  
c.x.x := a;  
c := a.x;  
// A
```

Aliases at // A:

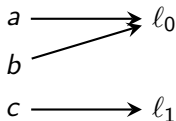
- ▶ a and b represent the same object
- ⇒ a and b are *aliased*

$a \stackrel{\text{alias}}{=} b$

- ⇒ 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* ($\stackrel{alias}{=}$) between variables
- Which *pair/set of variables* may/must point to the same address?

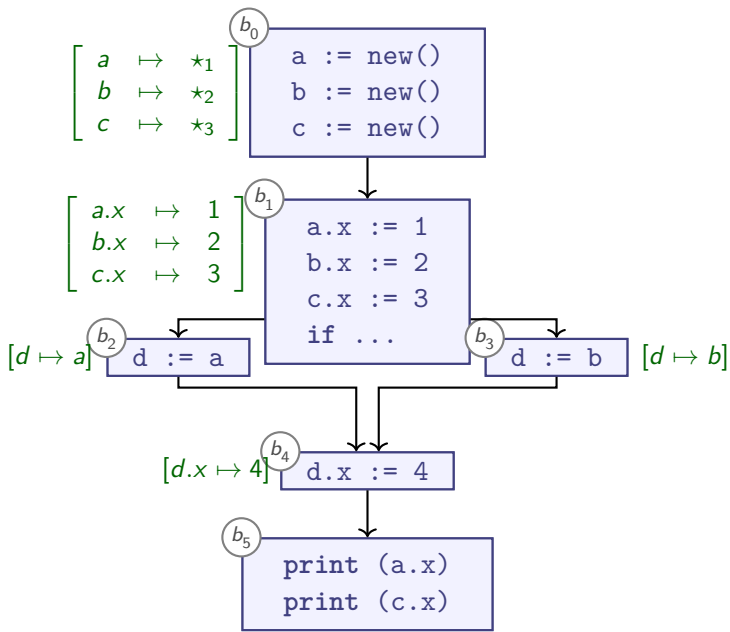
$$a \stackrel{alias}{=} b$$

Summary: Pointer Analysis

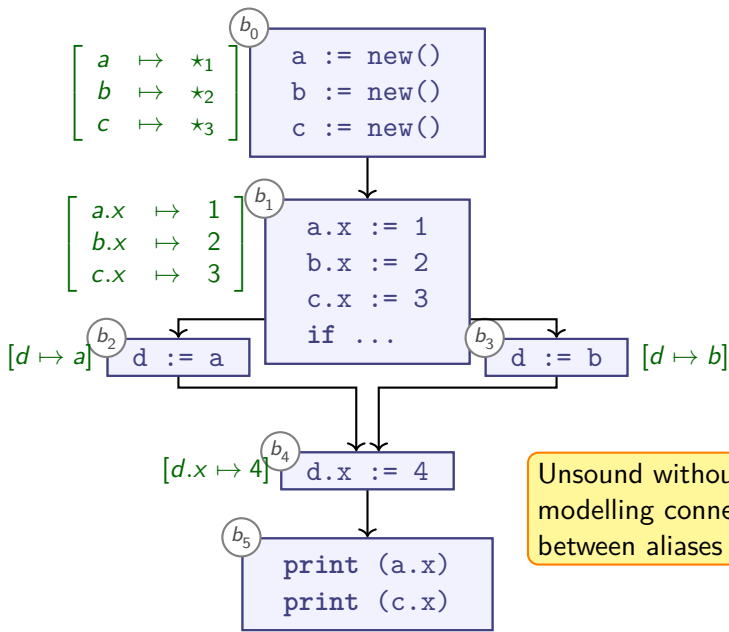
- ▶ Class of analyses to model dynamic heap allocation
- ▶ **Points-To Analysis:** computes mapping
 - ▶ From *variables*
 - ▶ To *pointees* (other variables)
 - ▶ More general than Alias Analysis
- ▶ **Alias Analysis:** computes
 - ▶ *Sharing information* between variables
 - ▶ Implicitly produced by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

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

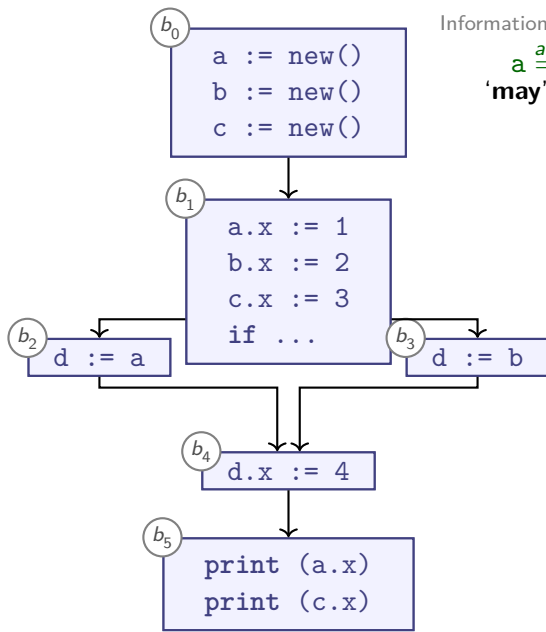
Dataflow with Alias Information



Dataflow with Alias Information



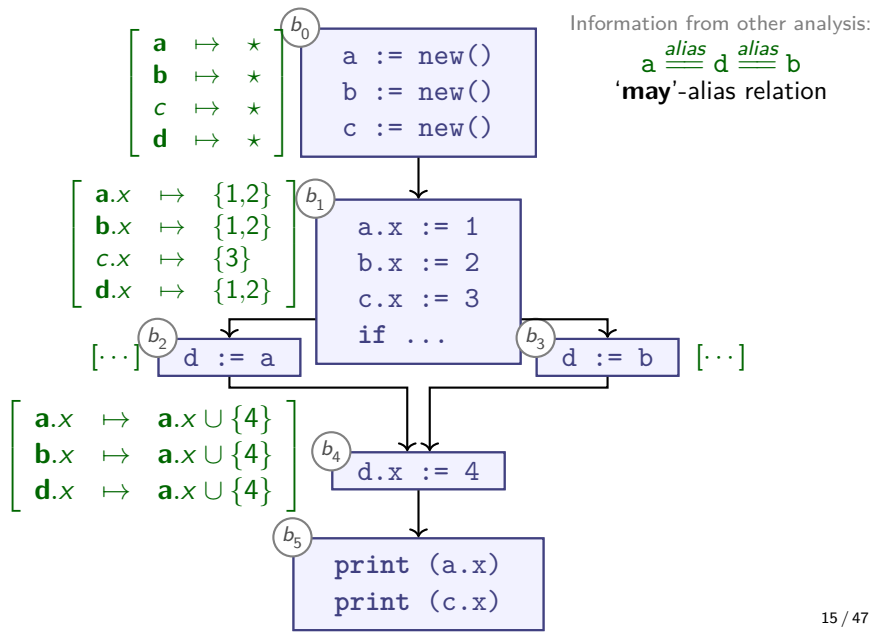
Dataflow with Alias Information



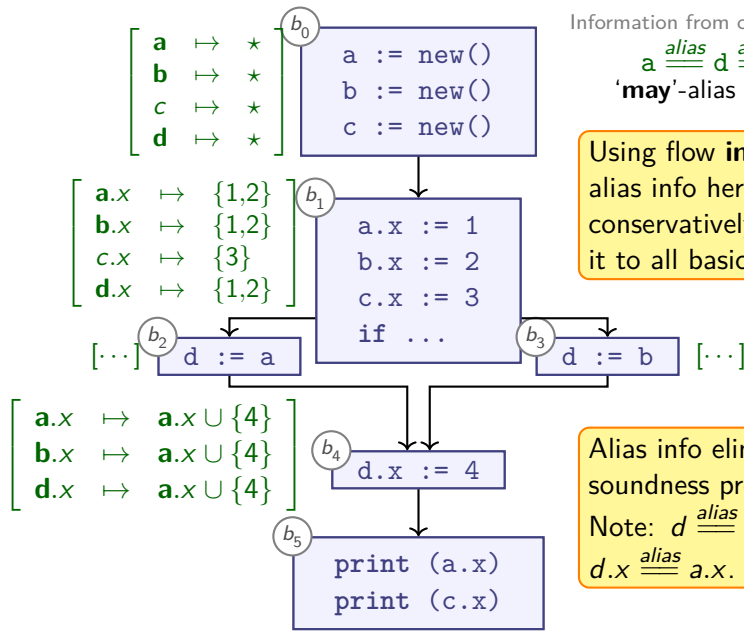
Information from other analysis:

$a \xrightarrow{\text{alias}} d \xrightarrow{\text{alias}} b$
'may'-alias relation

Dataflow with Alias Information



Dataflow with Alias Information



Information from other analysis:

$a \xrightarrow{\text{alias}} d \xrightarrow{\text{alias}} b$
 'may'-alias relation

Using flow **insensitive** alias info here, so we conservatively apply it to all basic blocks.

Alias info eliminates soundness problem.
 Note: $d \xrightarrow{\text{alias}} a$ implies $d.x \xrightarrow{\text{alias}} a.x$.

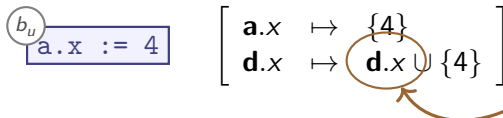
Dataflow + Aliases

- Aliasing affects shared fields:

$$a \stackrel{\text{alias}}{=} d \implies a.x \stackrel{\text{alias}}{=} d.x \text{ for all } x$$

- Use 'may' aliasing knowledge in one of these ways:

- 1 Multiply *updates* for each alias:

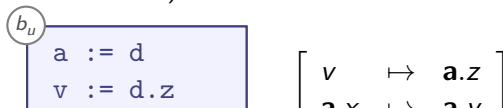


Using MAY alias info means: program *may or may not* not update aliased object.

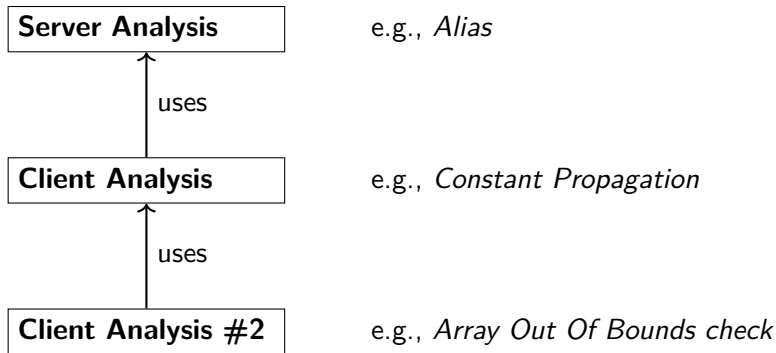
- 2 Multiply *reads* for each alias:



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



Collaboration in Program Analysis



Analyses often form pipeline structures

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:

$$a \stackrel{\text{alias}}{=} b \quad \implies \quad a.x \stackrel{\text{alias}}{=} b.x \text{ for all } x$$

- ▶ So if $a.x \stackrel{\text{alias}}{=} b.y$, then:
 - ▶ $a.x.z \stackrel{\text{alias}}{=} b.y.z$
 - ▶ $a.x.z.z \stackrel{\text{alias}}{=} b.y.z.z$
 - ▶ $a.x.z.z.z \stackrel{\text{alias}}{=} b.y.z.z.z$ etc.

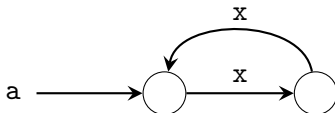
Concrete Heap Graphs (1/3)

Describe heap as a graph:

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

- ▶ G_{CHG} describes *actual* heap contents
- ▶ MemLoc are addressable memory locations
 - ▶ *Named* variables (a)
 - ▶ *Unnamed* variables ($\bigcirc$)
- ▶ 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:

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

- Exactly one ℓ' s.th. $\ell \rightarrow \ell'$:
 - For simplicity, assume **null** $\in \text{MemLoc}$
- Points-to references via fields:

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

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

Concrete Heap Graphs (3/3)

- Direct points-to references:

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

- Language difference:

- **Java/Teal:**

- *Var* is set of global / local variables and parameters
- $Var \cap RefLoc = \emptyset$
- $MemLoc = Var \cup RefLoc$

- **C/C++:** $Var = RefLoc = MemLoc$

- Address-of operator (&) allows translating variable into *MemLoc*
- $Var \subseteq RefLoc$
- $MemLoc = RefLoc$

Example

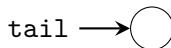
Teal-2

```
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

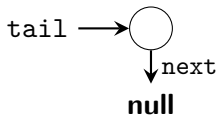
```
fun makeList(len) {  
  var tail := new N(); //⇐  
  tail.next := null;  
  var body := tail;  
  while len > 0 {  
    var t := body;  
    body := new N();  
    body.next := t;  
    len := len - 1;  
  }  
  var list := new N();  
  list.head := body;  
  list.tail := tail;  
  return list; 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



Example

Teal-2

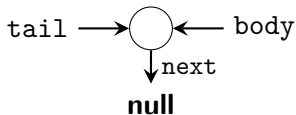
```
fun makeList(len) {  
  var tail := new N();  
  tail.next := null;  //⇐  
  var body := tail;  
  while len > 0 {  
    var t := body;  
    body := new N();  
    body.next := t;  
    len := len - 1;  
  }  
  var list := new N();  
  list.head := body;  
  list.tail := tail;  
  return list;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



Example

Teal-2

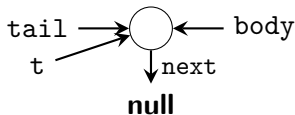
```
fun makeList(len) {  
  var tail := new N();  
  tail.next := null;  
  var body := tail; //⇐  
  while len > 0 {  
    var t := body;  
    body := new N();  
    body.next := t;  
    len := len - 1;  
  }  
  var list := new N();  
  list.head := body;  
  list.tail := tail;  
  return list; 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



Example

Teal-2

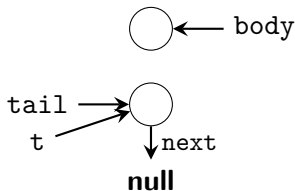
```
fun makeList(len) {  
  var tail := new N();  
  tail.next := null;  
  var body := tail;  
  while len > 0 {  
    var t := body; //⇐  
    body := new N();  
    body.next := t;  
    len := len - 1;  
  }  
  var list := new N();  
  list.head := body;  
  list.tail := tail;  
  return list; 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



Example

Teal-2

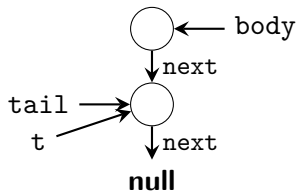
```
fun makeList(len) {  
  var tail := new N();  
  tail.next := null;  
  var body := tail;  
  while len > 0 {  
    var t := body;  
    body := new N(); //⇐  
    body.next := t;  
    len := len - 1;  
  }  
  var list := new N();  
  list.head := body;  
  list.tail := tail;  
  return list; 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



Example

Teal-2

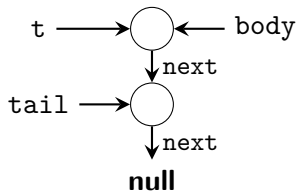
```
fun makeList(len) {  
  var tail := new N();  
  tail.next := null;  
  var body := tail;  
  while len > 0 {  
    var t := body;  
    body := new N();  
    body.next := t; // ←  
    len := len - 1;  
  }  
  var list := new N();  
  list.head := body;  
  list.tail := tail;  
  return list;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



Example

Teal-2

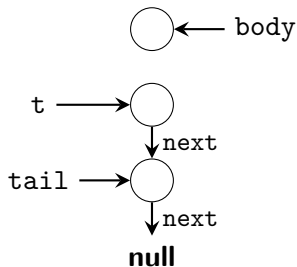
```
fun makeList(len) {  
  var tail := new N();  
  tail.next := null;  
  var body := tail;  
  while len > 0 {  
    var t := body; //⇐  
    body := new N();  
    body.next := t;  
    len := len - 1;  
  }  
  var list := new N();  
  list.head := body;  
  list.tail := tail;  
  return list; 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



Example

Teal-2

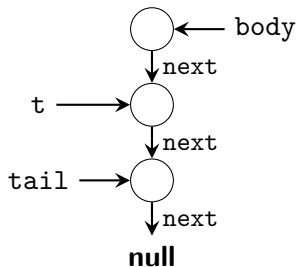
```
fun makeList(len) {  
  var tail := new N();  
  tail.next := null;  
  var body := tail;  
  while len > 0 {  
    var t := body;  
    body := new N(); //⇐  
    body.next := t;  
    len := len - 1;  
  }  
  var list := new N();  
  list.head := body;  
  list.tail := tail;  
  return list; 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



Example

Teal-2

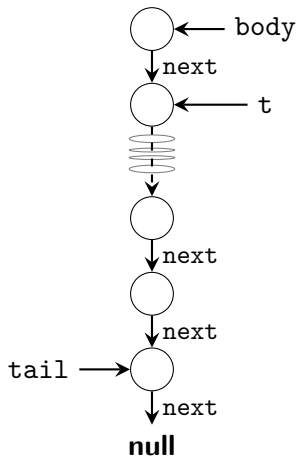
```
fun makeList(len) {  
  var tail := new N();  
  tail.next := null;  
  var body := tail;  
  while len > 0 {  
    var t := body;  
    body := new N();  
    body.next := t; // ←  
    len := len - 1;  
  }  
  var list := new N();  
  list.head := body;  
  list.tail := tail;  
  return list;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



Example

Teal-2

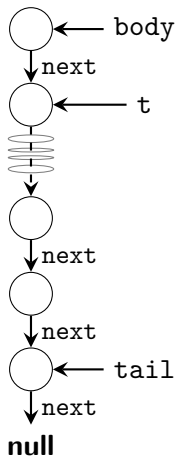
```
fun makeList(len) {  
  var tail := new N();  
  tail.next := null;  
  var body := tail;  
  while len > 0 {  
    var t := body; //⇐  
    body := new N(); //⇐  
    body.next := t; //⇐  
    len := len - 1;  
  }  
  var list := new N();  
  list.head := body;  
  list.tail := tail;  
  return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



Example

Teal-2

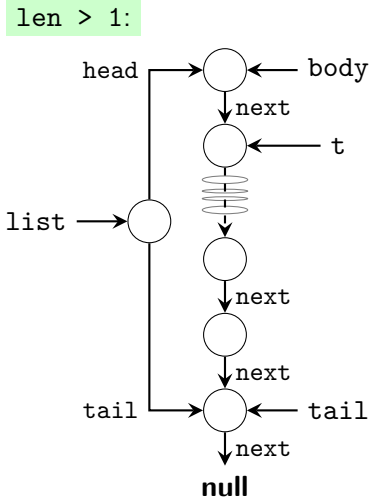
```
fun makeList(len) {  
  var tail := new N();  
  tail.next := null;  
  var body := tail;  
  while len > 0 {  
    var t := body;  
    body := new N();  
    body.next := t;  
    len := len - 1;  
  }  
  var list := new N();  
  list.head := body;  
  list.tail := tail;  
  return list;  
}
```



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;  
}
```

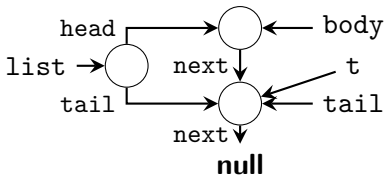


Example

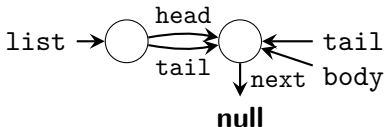
Teal-2

```
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:

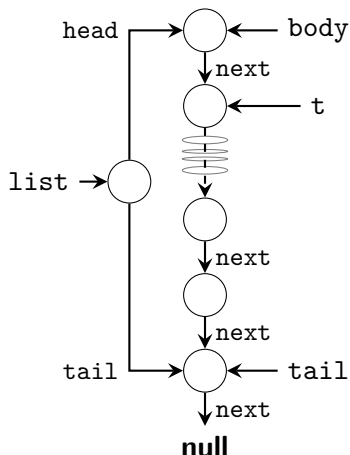


Managing Heap Graphs

- ▶ **Challenge:** 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:** $\text{list.tail} \stackrel{\text{alias}}{=} \text{tail}$

- ▶ **Must:** $\text{list.head} \stackrel{\text{alias}}{=} \text{body}$

- ▶ **Must:** $\text{body.next} \stackrel{\text{alias}}{=} t$

- ▶ **May:** $\text{body.next}^* \stackrel{\text{alias}}{=} \text{tail}$

- ▶ Use *regular expressions* to denote repetition

- ▶ $\text{body.next}^* \stackrel{\text{alias}}{=} \text{tail}$ means:

body	$\stackrel{\text{alias}}{=}$	tail
---------------	------------------------------	---------------

body.next	$\stackrel{\text{alias}}{=}$	tail
--------------------	------------------------------	---------------

body.next.next	$\stackrel{\text{alias}}{=}$	tail
-------------------------	------------------------------	---------------

...

- ▶ 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

- ▶ **Challenge:** 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

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

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

$$G_{\text{AHG}} = \langle \mathcal{P}(\text{MemLoc}), \overset{\color{blue}\blacksquare}{\rightarrow}, \overset{\color{blue}\blacksquare}{\rightarrow} \rangle$$

$$(\overset{\color{blue}\blacksquare}{\rightarrow}) : \mathcal{P}(\text{MemLoc}) \rightarrow \mathcal{P}(\text{MemLoc})$$

$$(\overset{\color{blue}\blacksquare}{\rightarrow}) : \mathcal{P}(\text{MemLoc}) \times \mathcal{P}(\text{Field}) \rightarrow \mathcal{P}(\text{MemLoc})$$

- ▶ Key idea: AHG is *finite* graph that summarises CHG
- ▶ Soundness (**May** analysis) via:

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

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

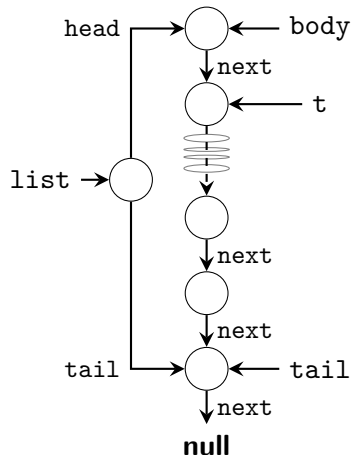
Summary Nodes and Edges

Notation:

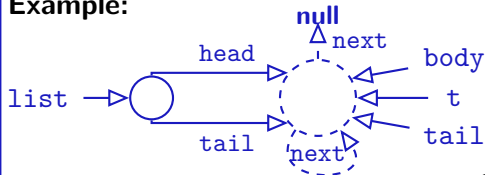
- Abstract node $N \subseteq \text{MemLoc}$:
 - $|N| = 1$: *precise*: 
 - $|N| > 1$ or variable: *summary*: 
- Consider edge $V \rightarrow L$:
 - $|V| = 1$: *precise*:

$$V \rightarrow L$$
 - $|V| > 1$: *summary*:

$$V \dashrightarrow L$$
- Analogous for $(\xrightarrow{f})$



Example:



Summary

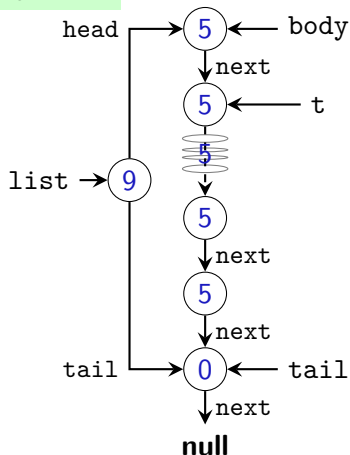
- ▶ **Store-based Models:**
 - ▶ Use **Abstract Heap Graph** to summarise *Concrete Heap Graph*
 - ▶ Common for finding memory bugs
 - ▶ Represents NFA
 - ▶ Equivalent to regular expressions

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;  
}
```

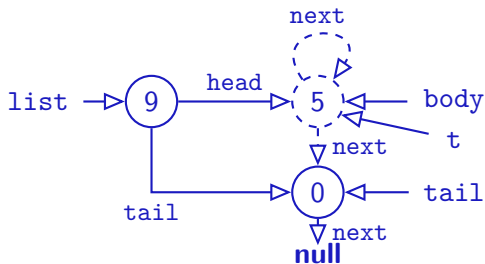
len > 1:



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;  
}
```

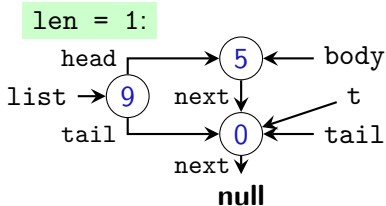
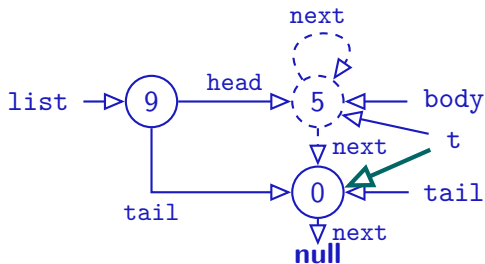


- Summarise *MemLoc* allocated at same program location

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;  
}
```

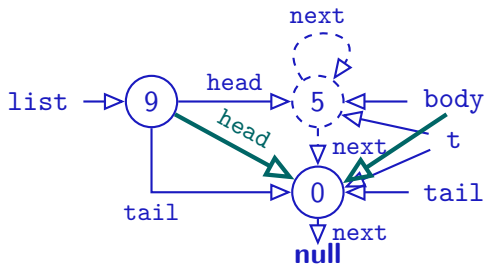


- Summarise *MemLoc* allocated at same program location
- Nodes can have multiple outgoing arrows

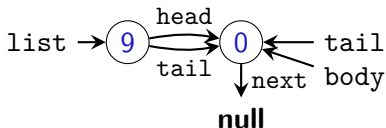
Summaries from Allocation Sites

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:

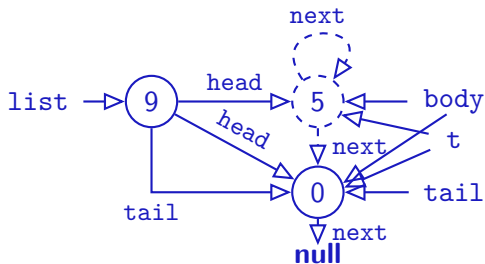


- Summarise *MemLoc* allocated at same program location
- Nodes can have multiple outgoing arrows

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 ($k=3$):

<code>list.head.next</code>	$\Rightarrow$	<code>list.head.next</code>
<code>list.head.next.next</code>	$\Rightarrow$	<code>list.head.next*</code>
<code>list.head.next.next.next</code>	$\Rightarrow$	<code>list.head.next*</code>
<code>list.head.next.next.val</code>	$\Rightarrow$	<code>list.head.(val next)*</code>

- ▶ # 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 P s allowed
 - ▶ Many combinations / extensions conceivable

Pointer Operations

- Points-to analysis must analyse all pointer operations
- All pointer operations fall into one of the following categories:

Referencing

$$a \rightarrow \ell$$

Create, point to location:

Dereferencing

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

Access location:

Aliasing

$$b \rightarrow \ell \iff a \rightarrow \ell$$

Copy pointer:

Teal-2

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

Teal-2

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

Teal-2

```
a := b;
```

Pointer Operations Across Languages

	C	Java	Teal
Referencing	<code>a = &b</code>	<code>a = new A()</code>	<code>a := new A()</code> <code>a := [..., b, ...]</code>
Aliasing	<code>a = b</code>	<code>a = b</code>	<code>a := b</code>
Dereferencing read	<code>a = b->f</code> <code>a = b[i]</code> <code>a = *b</code>	<code>a = b.f</code> <code>a = b[i]</code>	<code>a := b.f</code> <code>a := b[i]</code>
Dereferencing write	<code>a->f = b</code> <code>a[i] = b</code> <code>*a = b</code>	<code>a.f = b</code> <code>a[i] = b</code>	<code>a.f := b</code> <code>a[i] := b</code>

Summary

- Points-to analysis: 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 Proceedings of POPL '96, pages 32–41. ACM Press, 1996.

Distinguishing Field Names?

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

Constraint Collection

- ▶ ‘Points-to-set’: $pt(v)$ approximates $\{\ell \mid v \rightarrow \ell\}$
 - ▶ $pt(v) = \{\ell \mid v \rightarrow \ell\}$
- ▶ For each statement in program:
 - ▶ If **Referencing** ($a := \text{new}_{\ell_b} \dots$) (allocation site ℓ_b):

$$\ell_b \in pt(a)$$

- ▶ If **Aliasing** ($a := b$):

$$pt(a) = pt(b)$$

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

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

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

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

Example

$\Rightarrow x := \text{new}_{\ell_z}$ $\ell_z \in \text{pts}(x)$
 $x := y$ $\text{pts}(x) = \text{pts}(y)$
 $x := y.\square$ for each $\ell \in \text{pt}(y)$
 $\Rightarrow \text{pt}(x) = \text{pt}(\ell)$
 $x.\square := y$ for each $\ell \in \text{pt}(x)$
 $\Rightarrow \text{pt}(y) = \text{pt}(\ell)$

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:

a → (ℓ₁)

p

b ↘ (ℓ₂)

q

r

► Steensgaard:

a →▷ (ℓ₁)

p

b ↘▷ (ℓ₂)

q

r

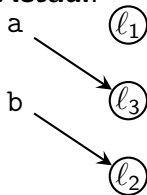
Example

$\Rightarrow x := \text{new}_{\ell_z} \quad \ell_z \in \text{pts}(x)$
 $x := y \quad \text{pts}(x) = \text{pts}(y)$
 $x := y.\square \quad \text{for each } \ell \in \text{pt}(y)$
 $\quad \quad \quad \Rightarrow \text{pt}(x) = \text{pt}(\ell)$
 $x.\square := y \quad \text{for each } \ell \in \text{pt}(x)$
 $\quad \quad \quad \Rightarrow \text{pt}(y) = \text{pt}(\ell)$

Teal

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

► Actual:

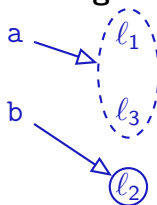


p

q

r

► Steensgaard:



p

q

r

Example

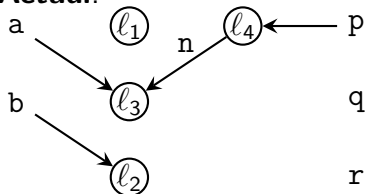
$x := \text{new}_{\ell_z}$ $\ell_z \in \text{pts}(x)$
 $x := y$ $\text{pts}(x) = \text{pts}(y)$
 $x := y.\square$ for each $\ell \in \text{pt}(y)$
 $\implies \text{pt}(x) = \text{pt}(\ell)$
 $\implies x.\square := y$ for each $\ell \in \text{pt}(x)$
 $\implies \text{pt}(y) = \text{pt}(\ell)$

Teal

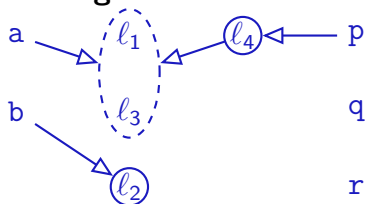
```

var a := new $\ell_1$ ();
var b := new $\ell_2$ ();
a := new $\ell_3$ ();
var p := new $\ell_4$ ();
p.n := a;           // ←
var q := new $\ell_6$ ();
q.n := b;
p := q;
var r := q.n;
    
```

► Actual:



► Steensgaard:



Example

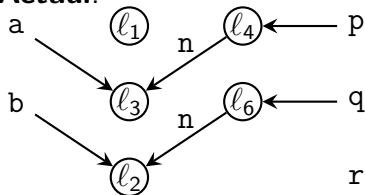
$x := \text{new}_{\ell_z}$ $\ell_z \in \text{pts}(x)$
 $x := y$ $\text{pts}(x) = \text{pts}(y)$
 $x := y.\square$ for each $\ell \in \text{pt}(y)$
 $\implies \text{pt}(x) = \text{pt}(\ell)$
 $\implies x.\square := y$ for each $\ell \in \text{pt}(x)$
 $\implies \text{pt}(y) = \text{pt}(\ell)$

Teal

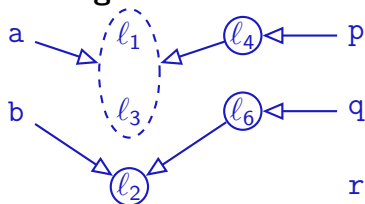
```

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

$x := \text{new}_{\ell_z}$ $\ell_z \in \text{pts}(x)$
 $\Rightarrow x := y$ $\text{pts}(x) = \text{pts}(y)$
 $x := y.\square$ for each $\ell \in \text{pt}(y)$
 $\Rightarrow \text{pt}(x) = \text{pt}(\ell)$
 $x.\square := y$ for each $\ell \in \text{pt}(x)$
 $\Rightarrow \text{pt}(y) = \text{pt}(\ell)$

Teal

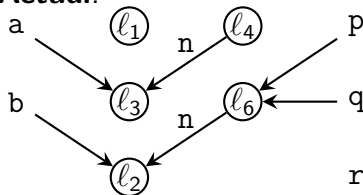
```

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;

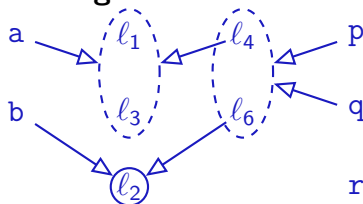
```

// $\Leftarrow$

► Actual:



► Steensgaard:



Example

$x := \text{new}_{\ell_z}$ $\ell_z \in \text{pts}(x)$
 $\Rightarrow x := y$ $\text{pts}(x) = \text{pts}(y)$
 $x := y.\square$ for each $\ell \in \text{pt}(y)$
 $\Rightarrow \text{pt}(x) = \text{pt}(\ell)$
 $x.\square := y$ for each $\ell \in \text{pt}(x)$
 $\Rightarrow \text{pt}(y) = \text{pt}(\ell)$

Teal

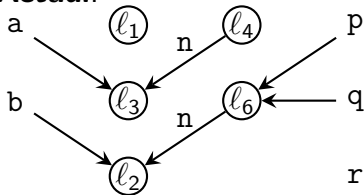
```

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;

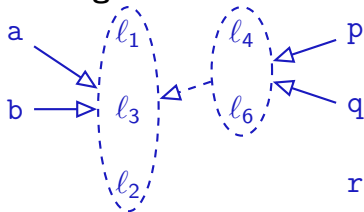
```

// ←

Actual:



Steensgaard:



When merging: 'collapse' children (merge recursively)

Example

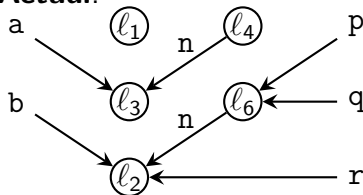
$x := \text{new}_{\ell_z}$ $\ell_z \in \text{pts}(x)$
 $x := y$ $\text{pts}(x) = \text{pts}(y)$
 $\Rightarrow x := y.\square$ for each $\ell \in \text{pt}(y)$
 $\Rightarrow \text{pt}(x) = \text{pt}(\ell)$
 $x.\square := y$ for each $\ell \in \text{pt}(x)$
 $\Rightarrow \text{pt}(y) = \text{pt}(\ell)$

Teal

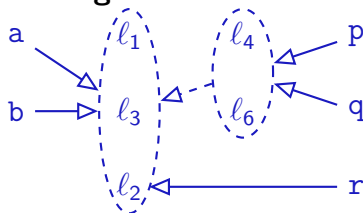
```

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:



When merging: 'collapse' children (merge recursively)

Summary

- ▶ Points-to sets $pt(v) = \{\ell \mid v \rightarrow \ell\}$
 - ▶ Partitions abstract heap locations
 - ▶ pt 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 Tuesday
- ▶ Lab 2 goes out today

<http://cs.lth.se/EDAP15>