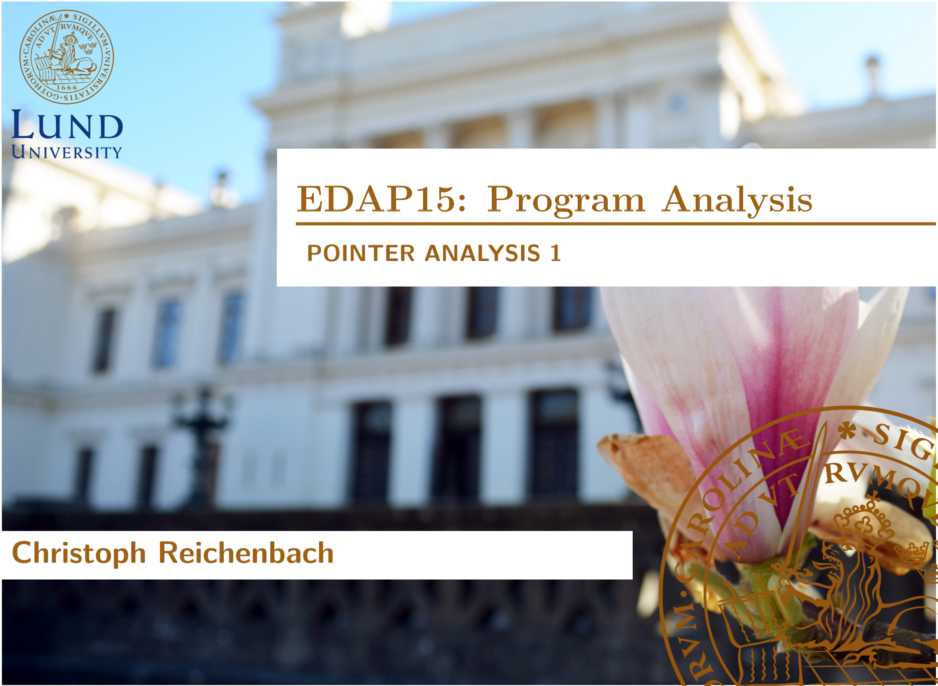


EDAP15: Program Analysis

POINTER ANALYSIS 1

Christoph Reichenbach

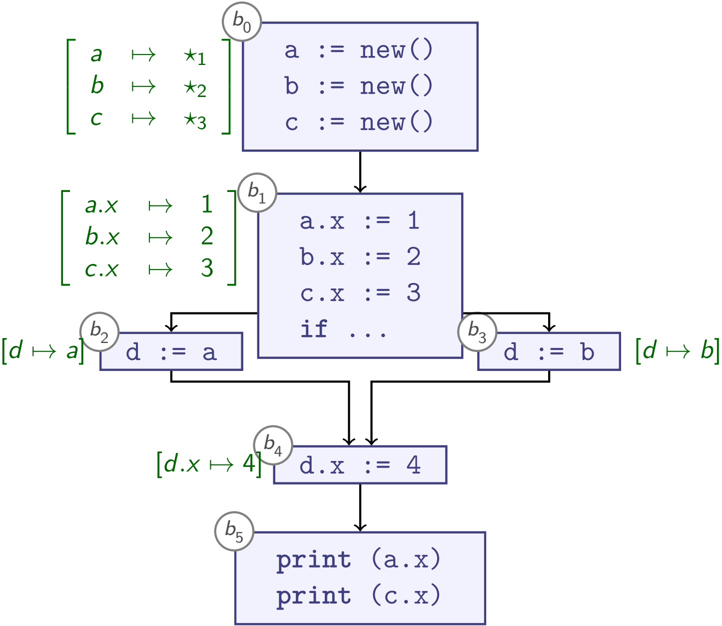
Welcome back!

- Questions?

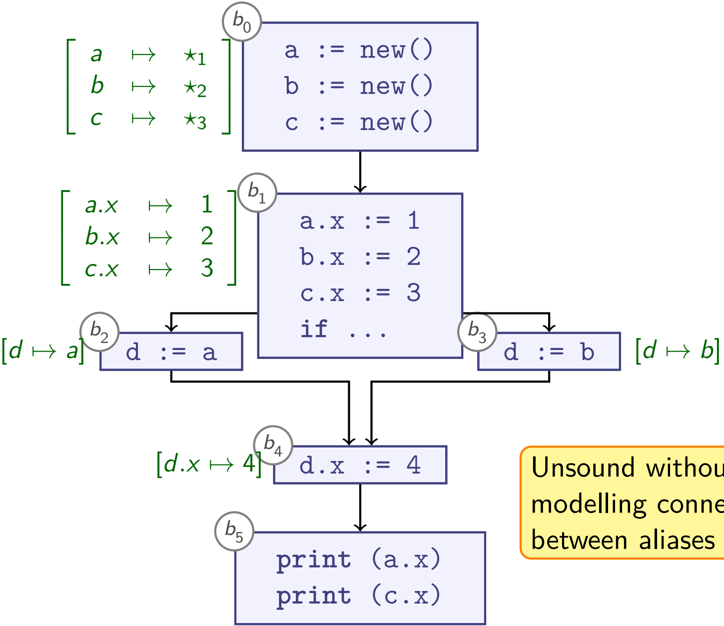
Teal-2

```
var x := new MyType();
x.field = value;
print(x.field);
```

Dataflow with Alias Information



Dataflow with Alias Information

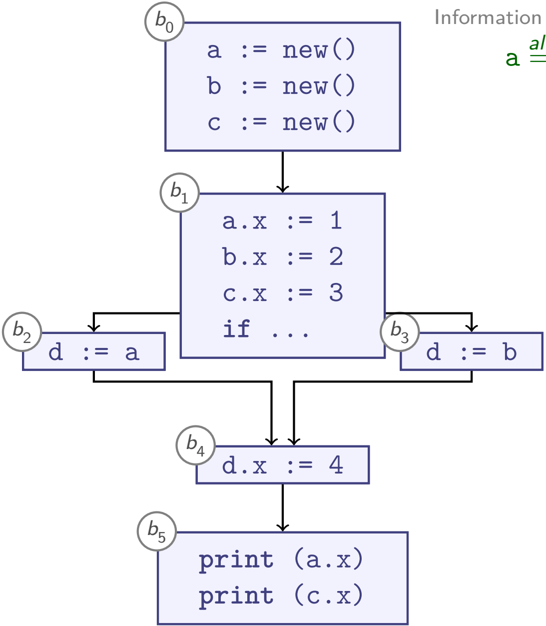


Unsound without modelling connection between aliases

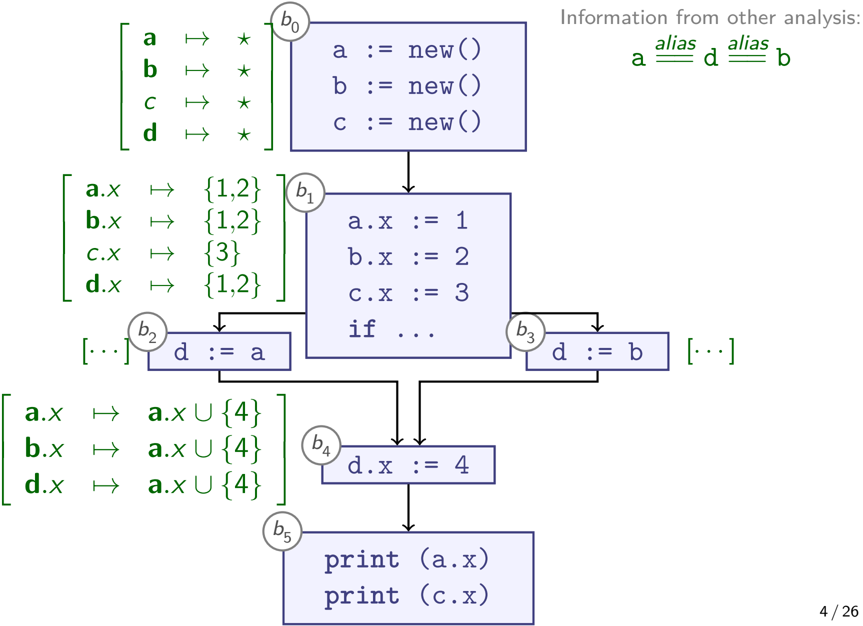
Dataflow with Alias Information

Information from other analysis:

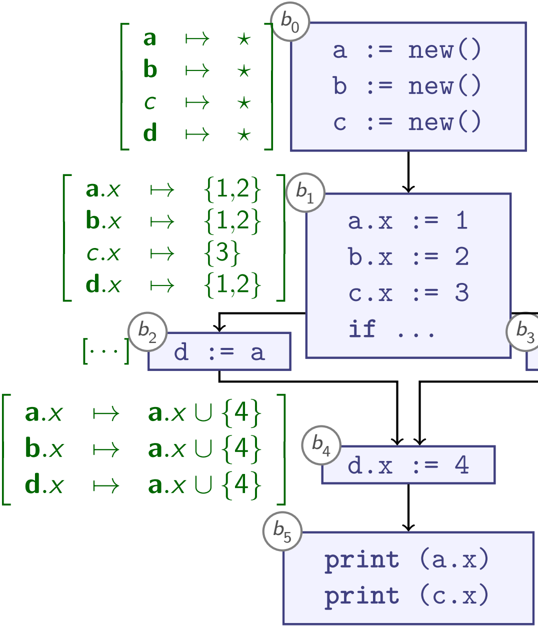
$\texttt{a} \overset{alias}{=\!=} \texttt{d} \overset{alias}{=\!=} \texttt{b}$



Dataflow with Alias Information



Dataflow with Alias Information



Information from other analysis:

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

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
```

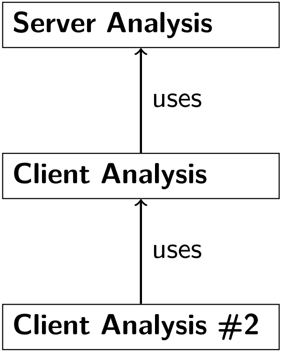
$$\begin{bmatrix} v & \mapsto & \mathbf{a}.x \cup \mathbf{d}.x \end{bmatrix}$$

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



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:

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

  - 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

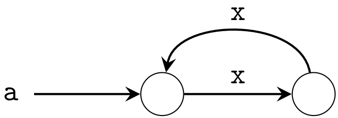
Concrete Heap Graphs (1/2)

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 ($\bigcirc$)
- Heap size typically ‘unbounded for all practical purposes’

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



Concrete Heap Graphs (2/2)

- Direct points-to references:

$$(\rightarrow) : (\mathit{Var} \cup \mathit{DynLoc}) \to \mathit{DynLoc}$$

- Language difference:
  - **Java/Teal**: *Var* is set of global / local variables and pameters
    - *Disjoint* from *DynLoc*
    - $\mathit{HeapLoc} = \mathit{Var} \cup \mathit{DynLoc}$
  - **C/C++**: $\mathit{Var} = \mathit{DynLoc} = \mathit{HeapLoc}$
    - Address-of operator (`&`) allows translating variable into *MemLoc*
- 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., $\mathbb{N} \subseteq \mathit{Field}$)

Example

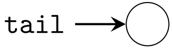
Teal-2

```
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

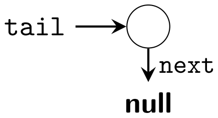
```
fun makeList(len) {
  var tail := new N(); //⇐
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

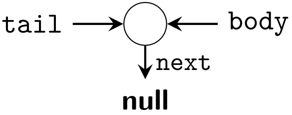
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;    //⇐
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

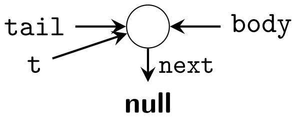
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;          //⇐
  while len > 0 {
    var t := body;
    body := new N();
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

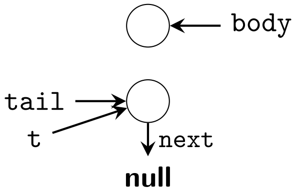
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;      //⇐
    body := new N();
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

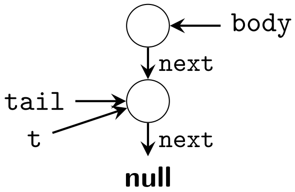
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();    //⇐
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

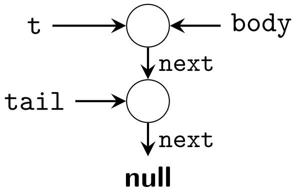
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();
    body.next := t;      //⇐
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

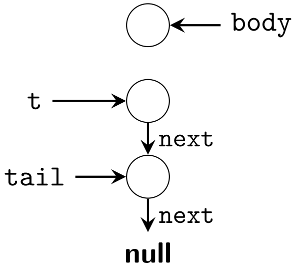
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;        //⇐
    body := new N();
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

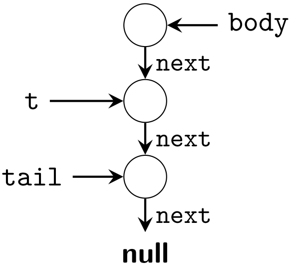
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();    //⇐
    body.next := t;
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

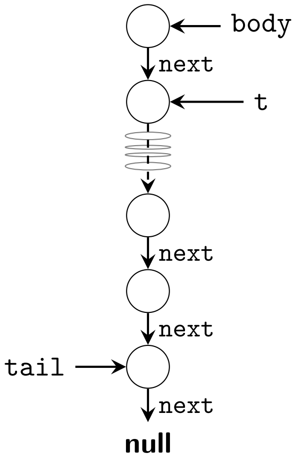
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;
    body := new N();
    body.next := t;     //⇐
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

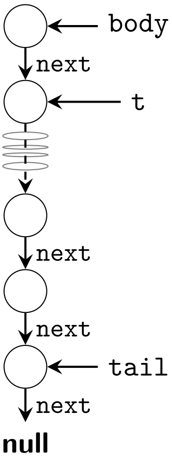
```
fun makeList(len) {
  var tail := new N();
  tail.next := null;
  var body := tail;
  while len > 0 {
    var t := body;          //⇐
    body := new N();        //⇐
    body.next := t;         //⇐
    len := len - 1;
  }
  var list := new N();
  list.head := body;
  list.tail := tail;
  return list;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



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;
}
```

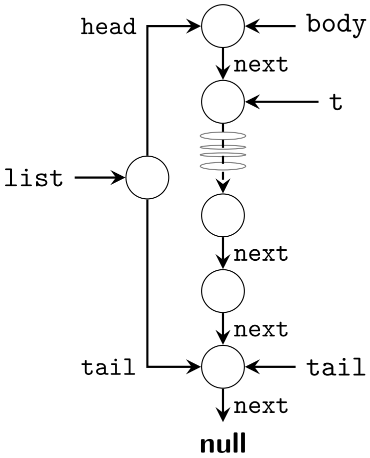


Example

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:

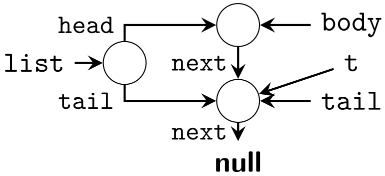


Example

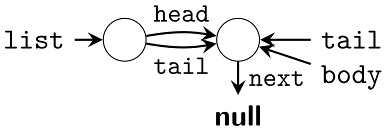
Teal-2

```
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:



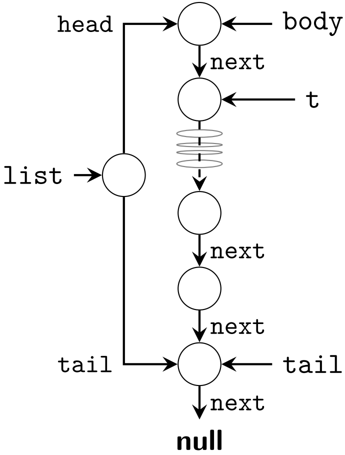
Managing Heap Graphs

- 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`
	...	

- 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

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

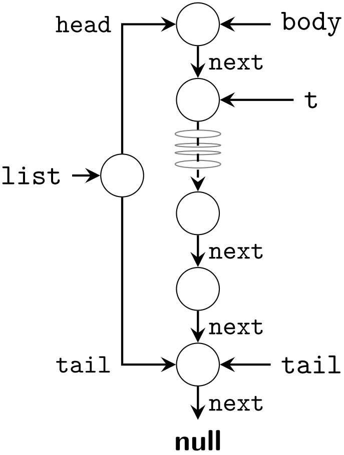
$$\begin{array}{rcl} (\rightarrowtriangle) & : & \mathcal{P}(\mathit{MemLoc}) \to \mathcal{P}(\mathit{MemLoc}) \\ (\overset{\blacksquare}{\rightarrowtriangle}) & : & \mathcal{P}(\mathit{MemLoc}) \times \mathcal{P}(\mathit{Field}) \to \mathcal{P}(\mathit{MemLoc}) \end{array}$$

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

$$\begin{array}{ccccccc} v & \rightarrow & \ell & \text{implies} & \{v\} \cup V' & \rightarrowtriangle & \{\ell\} \cup L' \\ \ell_0 & \overset{f}{\rightarrow} & \ell_1 & \text{implies} & \{\ell_0\} \cup L_0' & \overset{\{f\} \cup F'}{\rightarrowtriangle} & \{\ell_1\} \cup L_1' \end{array}$$

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

Summary Nodes and Edges



Notation:

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

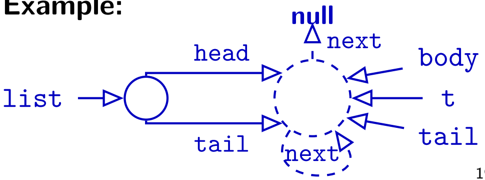
    $V \longrightarrow L$

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

    $V \dashrightarrow L$

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

Example:



Summary

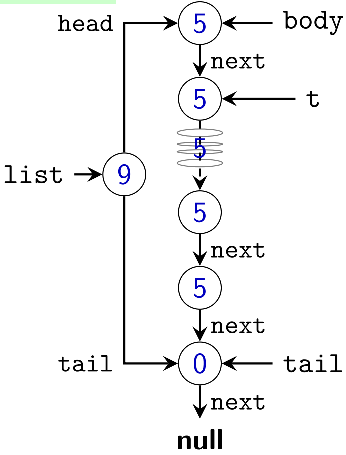
- **Store-based Models**:
  - Use **Abstract Heap Graph** to summarise *Concrete Heap Graph*
  - Common for finding memory bugs
  - Represents NFA
    - Equivalent to regular expressions

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;
}
```

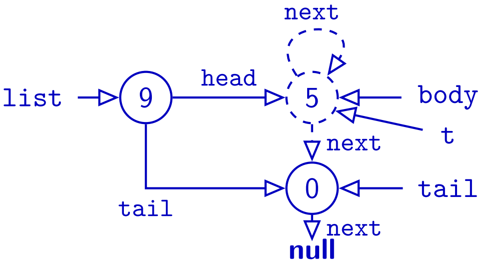
len > 1:



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;
}
```

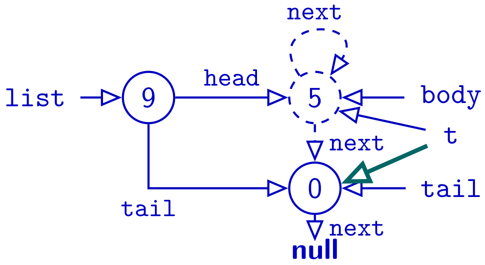


- Summarise *MemLoc* allocated at same program location

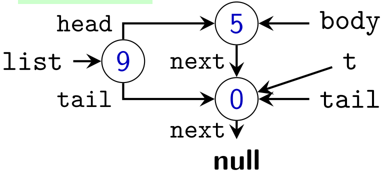
Summaries from Allocation Sites

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:

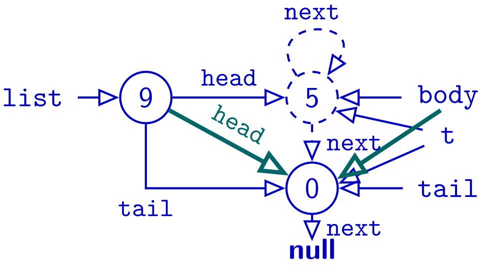


- Summarise *MemLoc* allocated at same program location
- Nodes can have multiple outgoing arrows

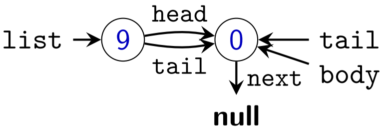
Summaries from Allocation Sites

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:

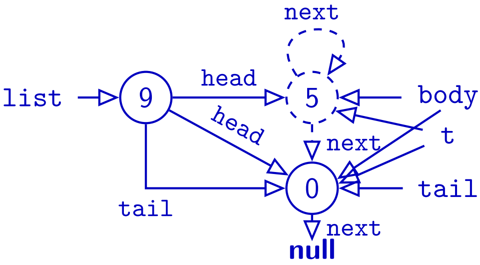


- Summarise *MemLoc* allocated at same program location
- Nodes can have multiple outgoing arrows

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                ⇒  list.head.next
list.head.next.next           ⇒  list.head.next⋆
list.head.next.next.next      ⇒  list.head.next⋆
list.head.next.next.val       ⇒  list.head.(val|next)⋆
```

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

  ...

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

- *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