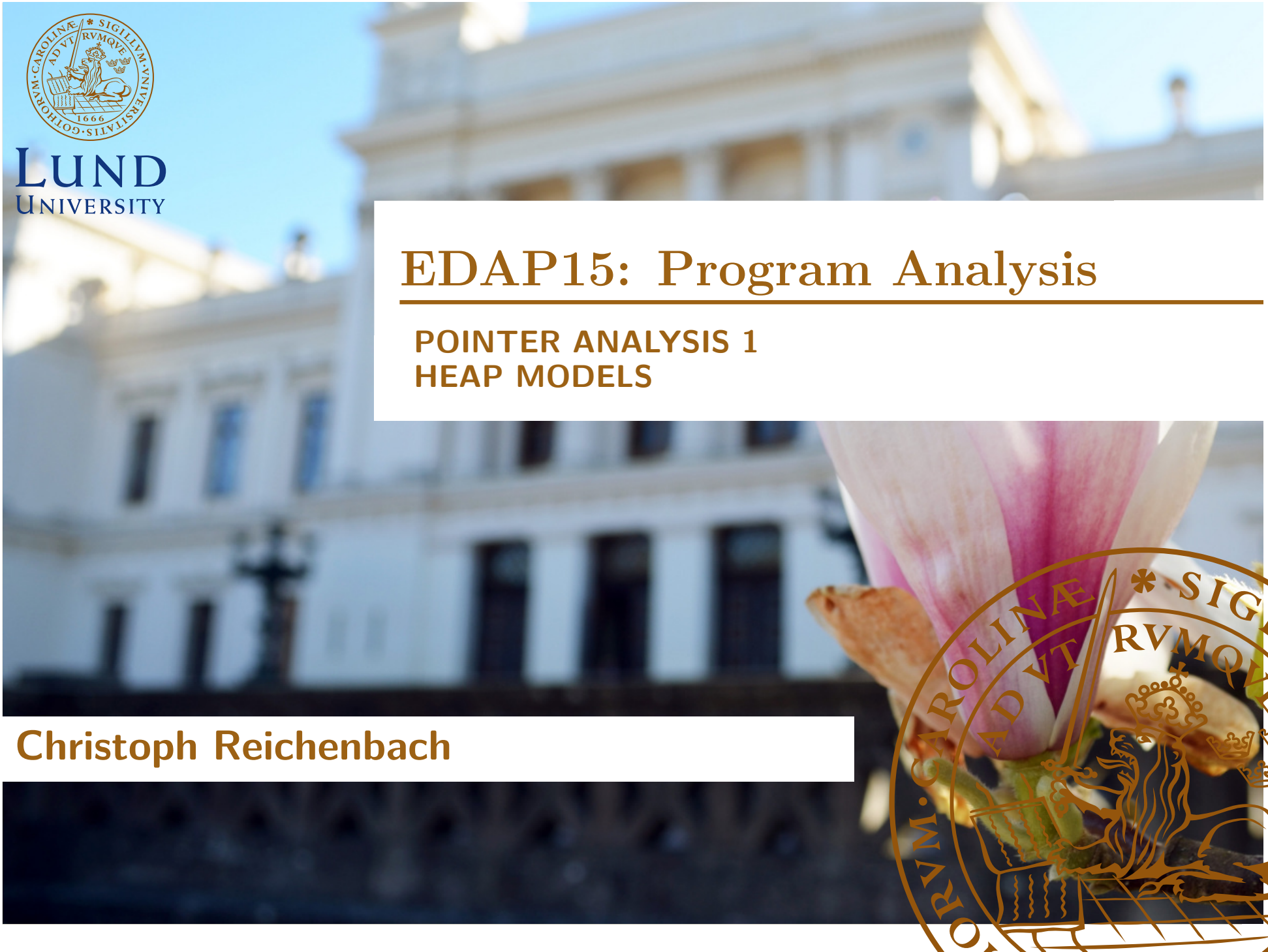


LUND UNIVERSITY

EDAP15: Program Analysis

POINTER ANALYSIS 1
HEAP MODELS

Christoph Reichenbach

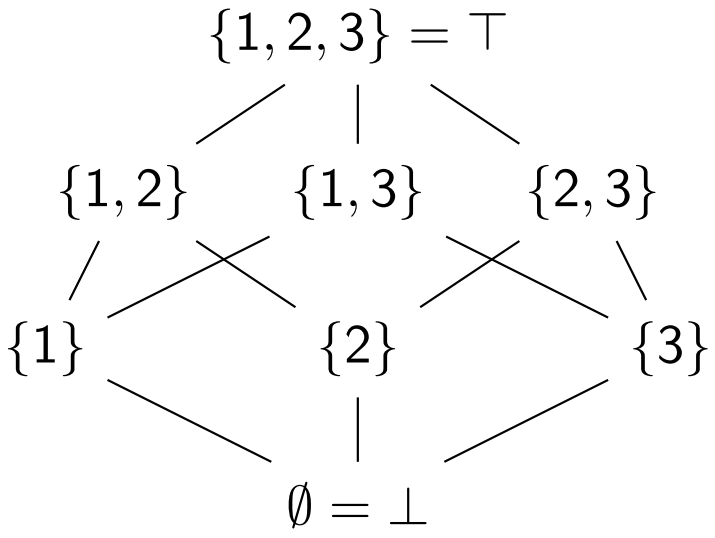
Dataflow Analysis

Analyse properties of variables or basic blocks

Examples in practice:

- *Live Variables*
  Is this variable ever read?
- *Reaching Values*
  What are the possible values for this variable?

Analyses on Powerset Lattices (1/2)

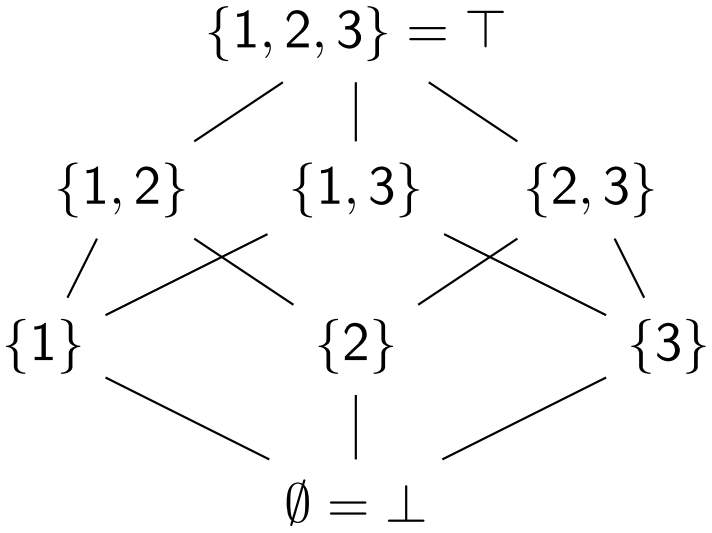


$join_b = \sqcup = \cup$

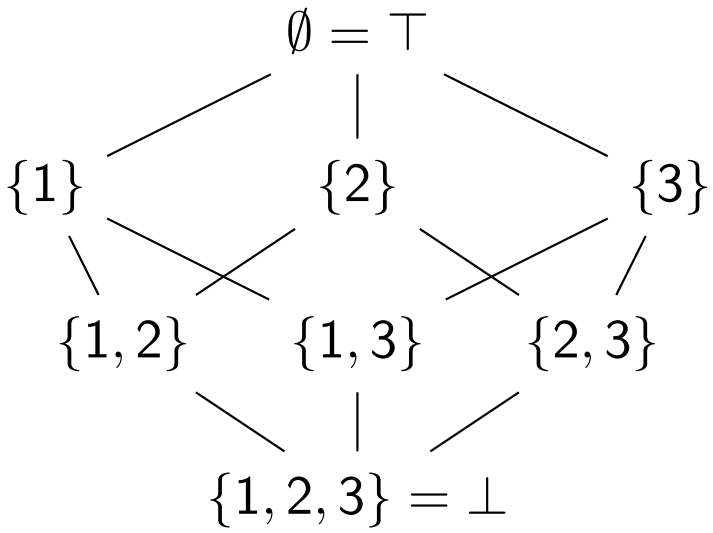
$S = \{1,2,3\}$

- Examples:
  - $S \subseteq \mathbb{Z}$ (*Reaching Definitions*)
    - $S =$ Numeric Constants in code $\cup \{0,1\}$
  - $S =$ Variables (*Live Variables*)
  - $S =$ Program Locations (*alt. Reaching Definitions*)
  - $S =$ Types
- Abstract Domain: Powerset $\mathcal{P}(S)$
  - Finite iff S is finite

Analyses on Powerset Lattices (2/2)



$join_b = \sqcup = \cup$



$join_b = \sqcup = \cap$

- $join_b$ can be $\cup$ or $\cap$
- $\cup$:
  - Property that is true on *some* path
  - **May**-analysis
- $\cap$:
  - Property that is true over *all* paths
  - **Must**-analysis

Available Expressions

"Which expressions do we currently have evaluated and stored?"

Teal

```
var y := 2 + z;
var x := 3 * z;
if z > 0 {
  x := 4;
}
f(2 + z); // Can re-use y here!
f(3 * z); // Can NOT re-use x here!
```

- Forward analysis
- $join_b = \sqcup = \cap$

Very Busy Expressions

"Which expression do we definitely need to evaluate at least once?"

Teal

```
// (x / 42) is very busy:  (A),(B).
// Can eval early!
if z > 0 {
  x := 4 + x / 42; // (A)
  y := 1;
} else {
  x := x / 42; // (B)
}
g(x);
```

- Backwards analysis
- $join_b = \sqcup = \cap$

Summary

- Data Flow Analysis in Monotone Frameworks:
  - **Forward** or **Backward**?
  - **May** or **Must**?
  - Which Lattice, $\sqcup$, $\top$, $\bot$?
  - Which transfer functions?

	May	**Must**
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions

Our Memory Modelling Until Now

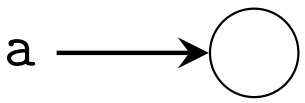
- Our analyses so far have considered:
  - Static Variables
  - Local (stack-dynamic) Variables
  - (Stack-dynamic) parameters

Missing: heap variables!

Example Program

Example

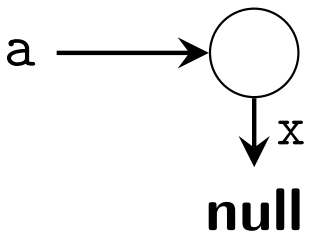
```
a := new();    // ⇐
a.x := null;
b := a;
b.x := new();
a.x.y := 1;
c := new();
c.x := new();
c.x.x := a;
c := a.x;
// A
```



Example Program

Example

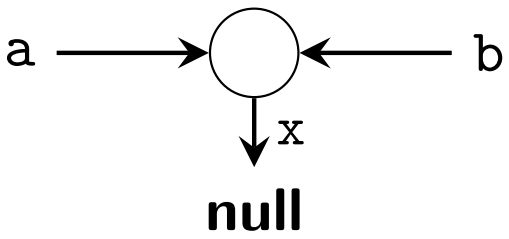
```
a := new();
a.x := null;  // ⇐
b := a;
b.x := new();
a.x.y := 1;
c := new();
c.x := new();
c.x.x := a;
c := a.x;
// A
```



Example Program

Example

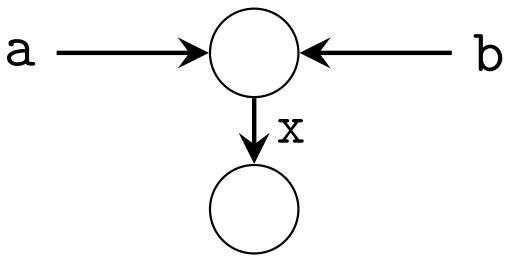
```
a := new();
a.x := null;
b := a;          // ⇐
b.x := new();
a.x.y := 1;
c := new();
c.x := new();
c.x.x := a;
c := a.x;
// A
```



Example Program

Example

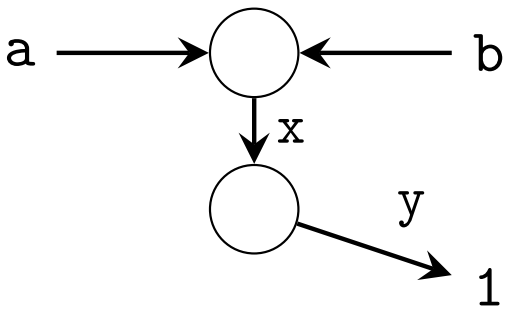
```
a := new();
a.x := null;
b := a;
b.x := new(); // ⇐
a.x.y := 1;
c := new();
c.x := new();
c.x.x := a;
c := a.x;
// A
```



Example Program

Example

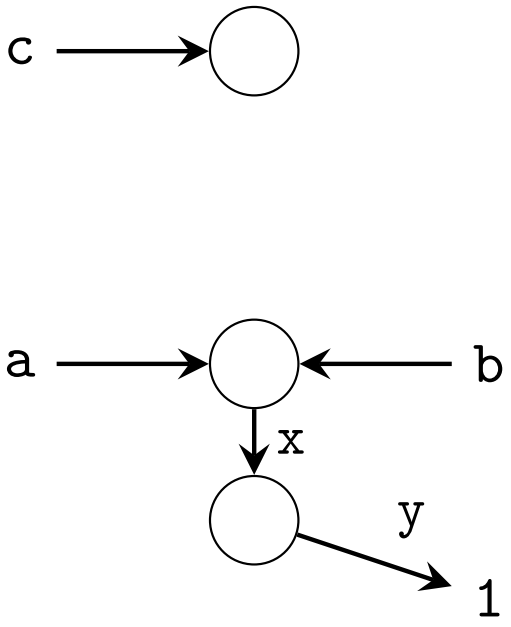
```
a := new();
a.x := null;
b := a;
b.x := new();
a.x.y := 1;    // ⇐
c := new();
c.x := new();
c.x.x := a;
c := a.x;
// A
```



Example Program

Example

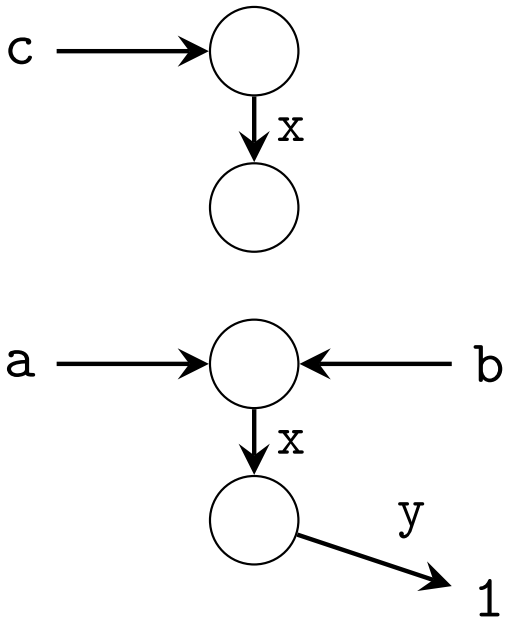
```
a := new();
a.x := null;
b := a;
b.x := new();
a.x.y := 1;
c := new();    // ⇐
c.x := new();
c.x.x := a;
c := a.x;
// A
```



Example Program

Example

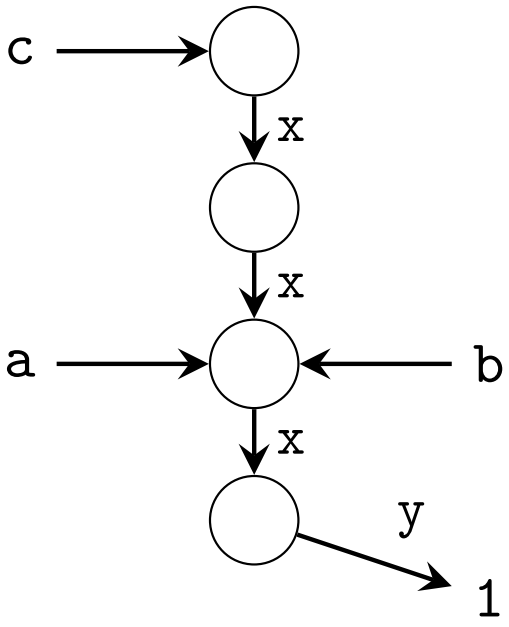
```
a := new();
a.x := null;
b := a;
b.x := new();
a.x.y := 1;
c := new();
c.x := new(); // ⇐
c.x.x := a;
c := a.x;
// A
```



Example Program

Example

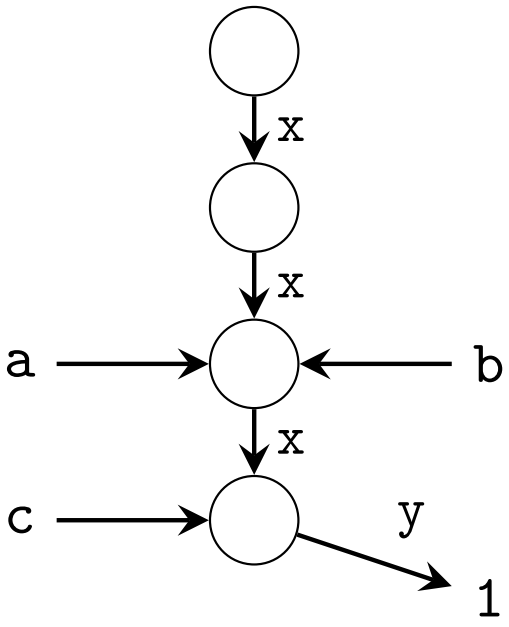
```
a := new();
a.x := null;
b := a;
b.x := new();
a.x.y := 1;
c := new();
c.x := new();
c.x.x := a;    // ⇐
c := a.x;
// A
```



Example Program

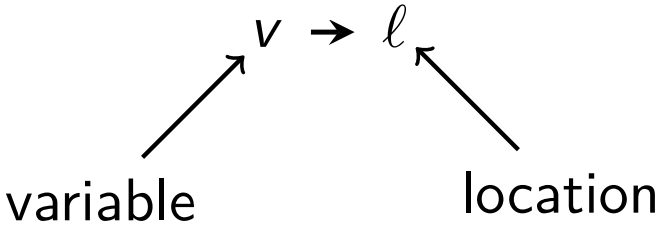
Example

```
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
```

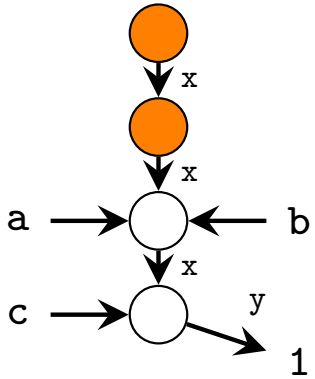


Concrete Heap Graph

"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`:

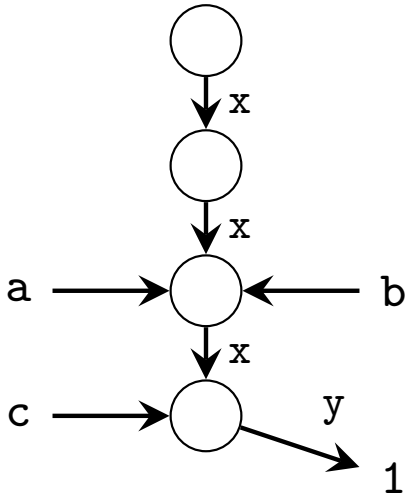
- a and b represent the same object

$\Rightarrow$ a and b are *aliased*

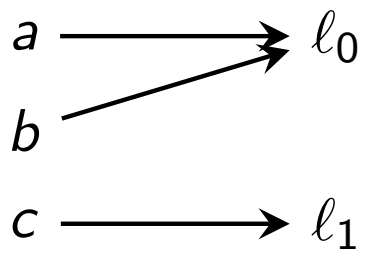
$$a \overset{alias}{=\!=} b$$

$\Rightarrow$ 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*
  - Which *pair/set of variables* may/must point to the same address?

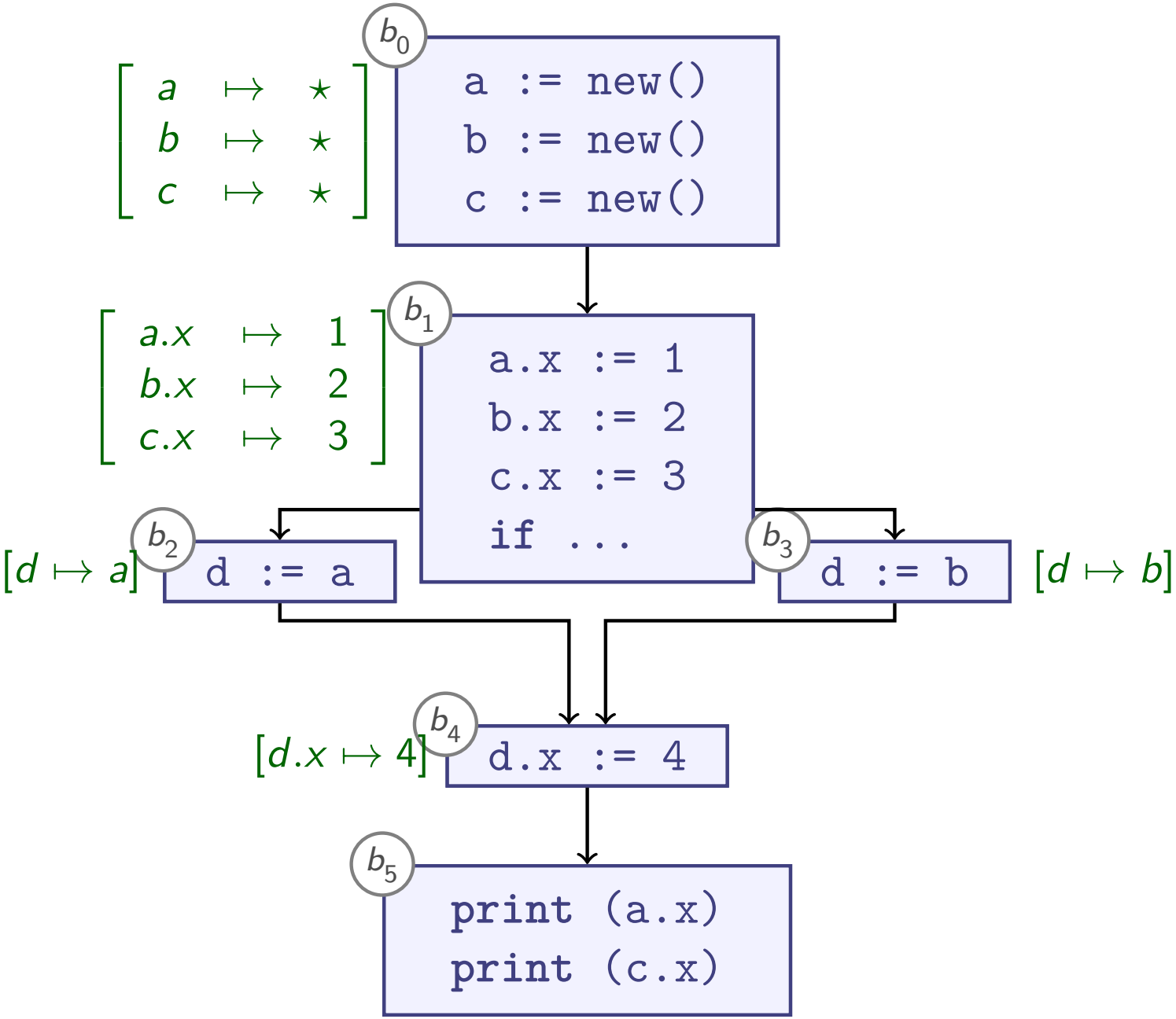
$$a \overset{alias}{=\!=} b$$

Summary: Pointer Analysis

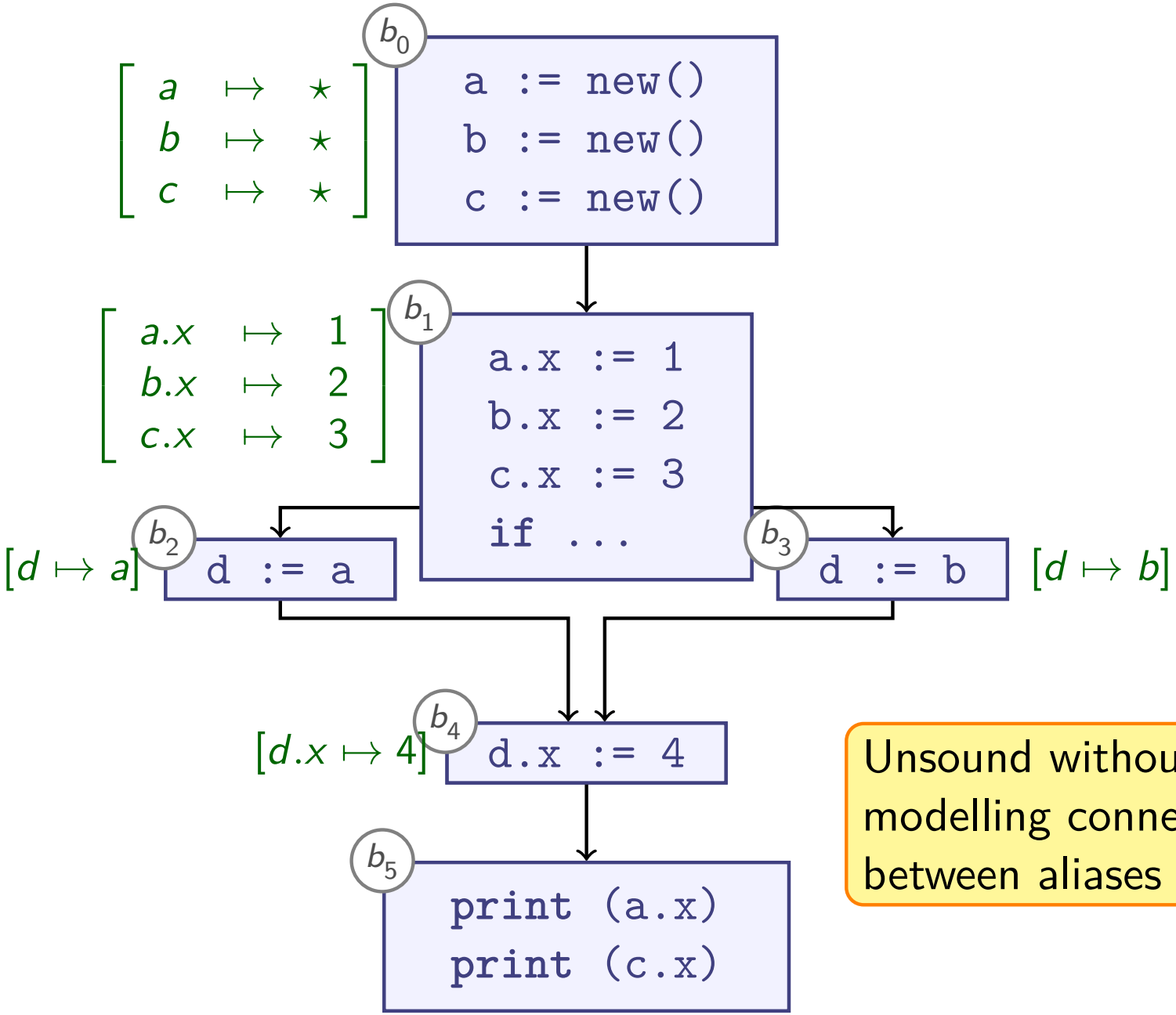
- Class of analyses to model dynamic heap allocation
- **Points-To Analysis**: computes mapping
  - From *variables*
  - To *pointees* (other variables)
  - More general than Alias Analysis
- **Alias Analysis**: computes
  - *Sharing information* between variables
  - Implicitly produced by points-to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

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

Dataflow with Alias Information

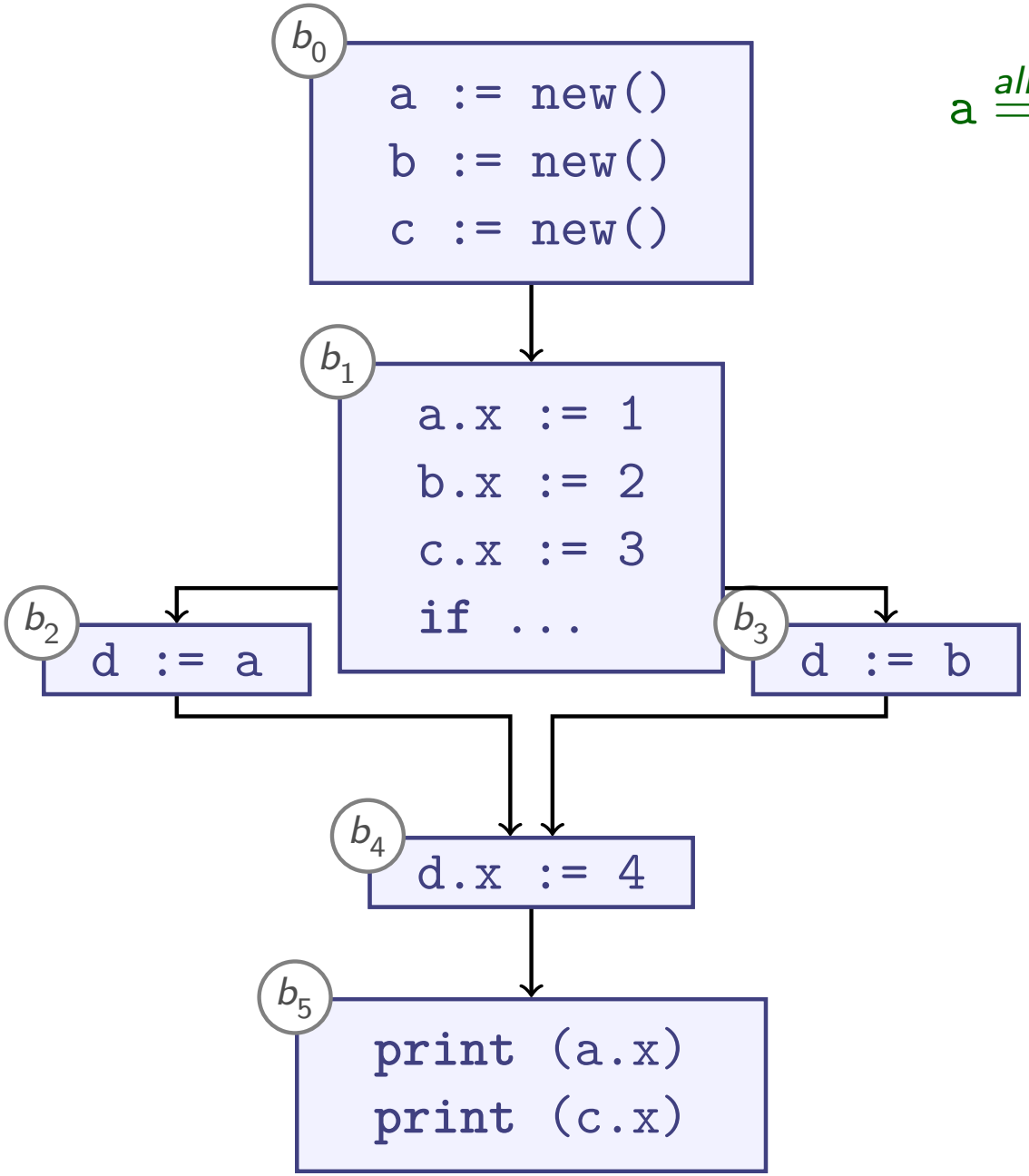


Dataflow with Alias Information



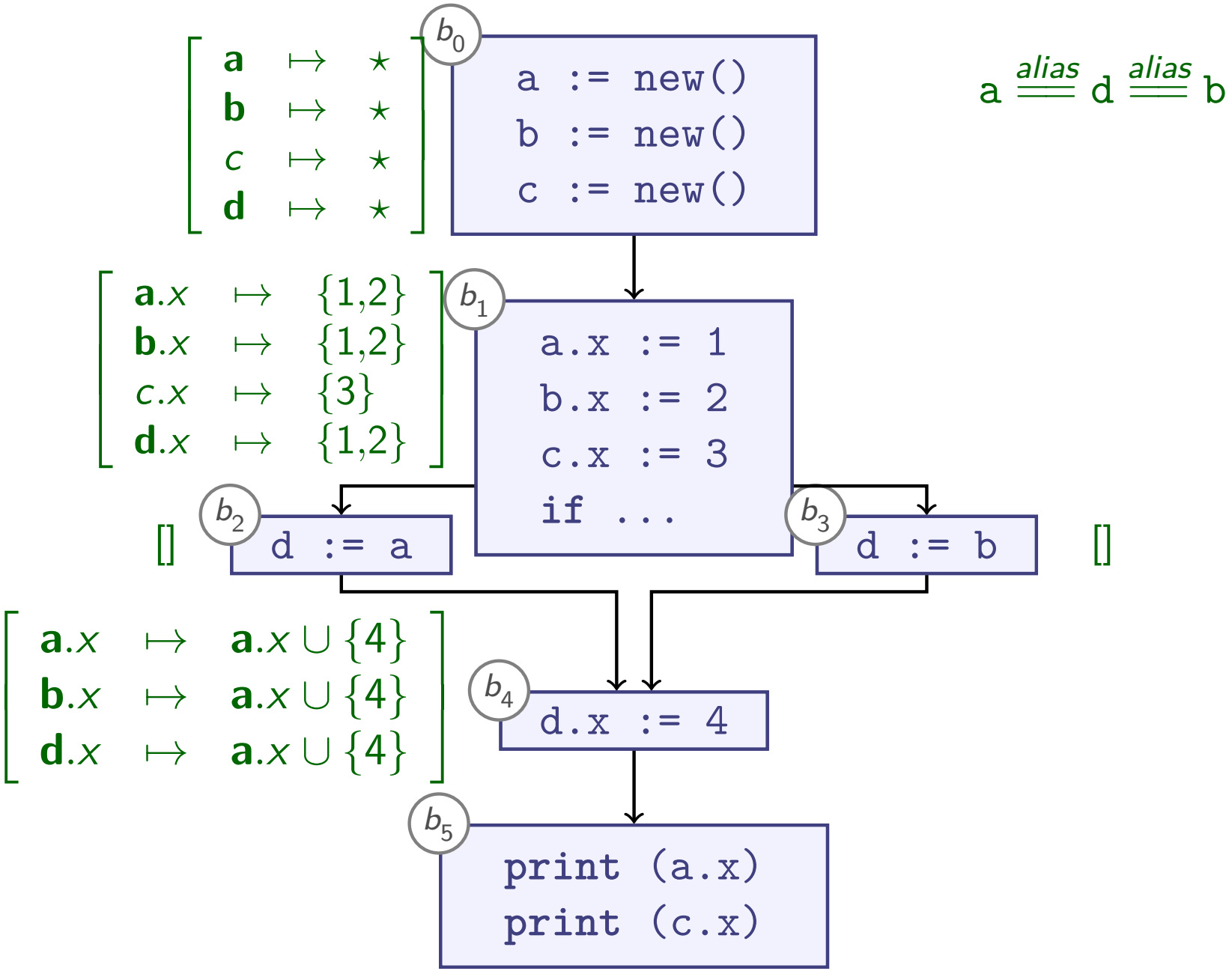
Unsound without modelling connection between aliases

Dataflow with Alias Information

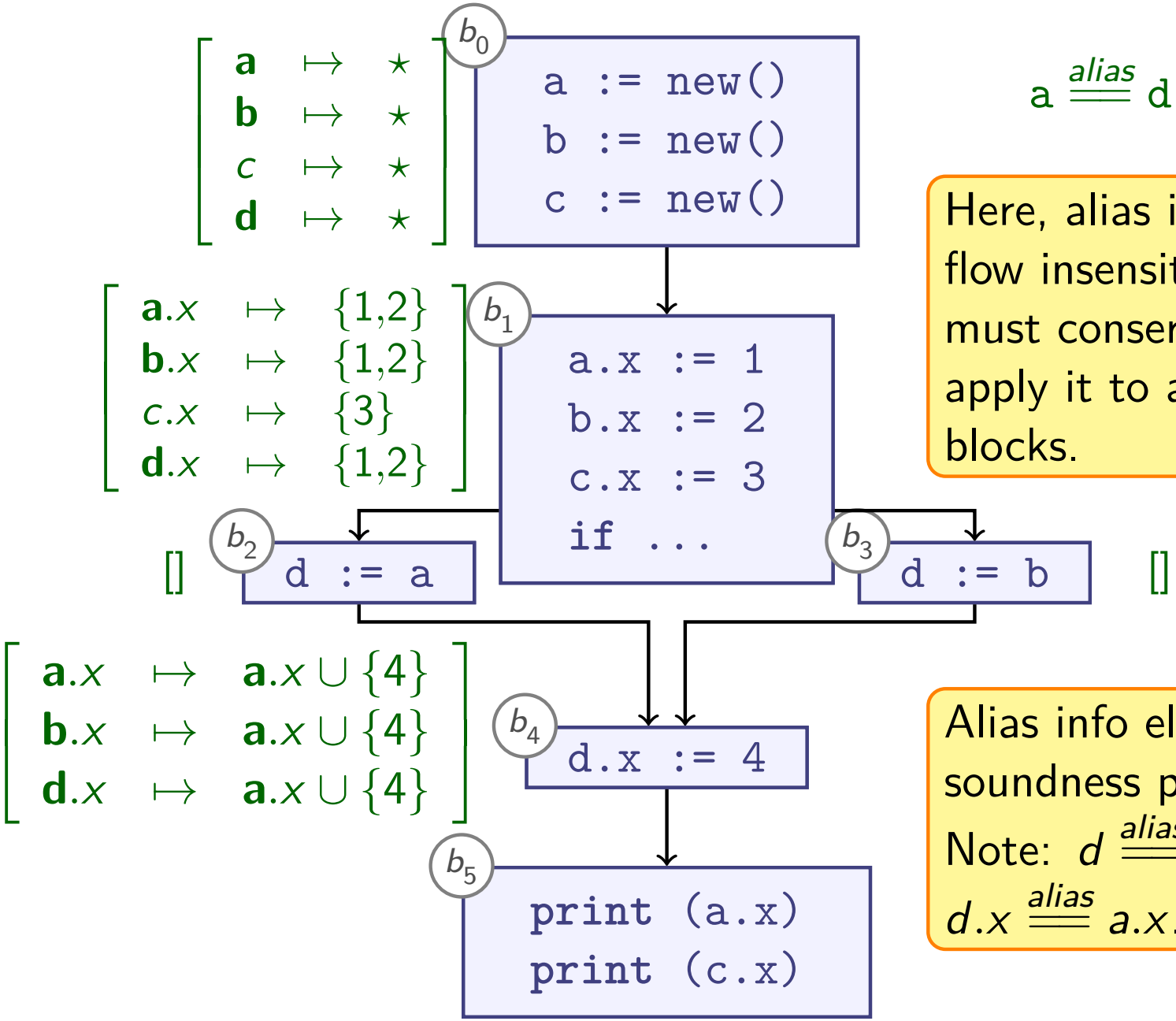


$a \overset{alias}{=\!=} d \overset{alias}{=\!=} b$

Dataflow with Alias Information



Dataflow with Alias Information



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

Here, alias info is flow insensitive 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:

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

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` by z
  - Must extract both y, z from $\mathbf{in}_b$ to compute update
  - *Non-distributive in practice*

Summary

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

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

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

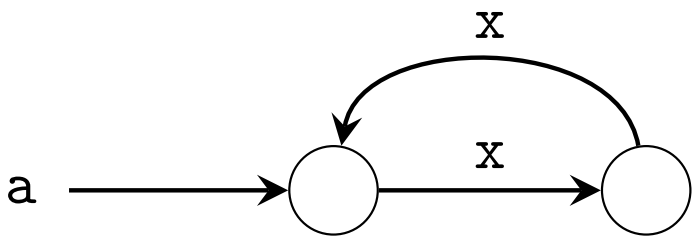
Concrete Heap Graphs (1/2)

Describe heap as a graph:

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

- 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 (2/2)

- Direct points-to references:

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

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

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

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

Example

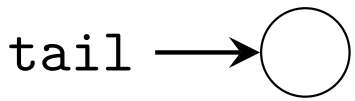
Teal-2

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

Example

Teal-2

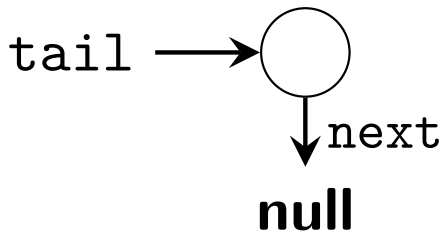
```
fun makeList(len) {
  tail := new N();        //⇐
  tail.next := null;
  body := tail;
  while len > 0 {
    t := body;
    body := new N();
    body.next := t;
    len := len - 1;
  }
  list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

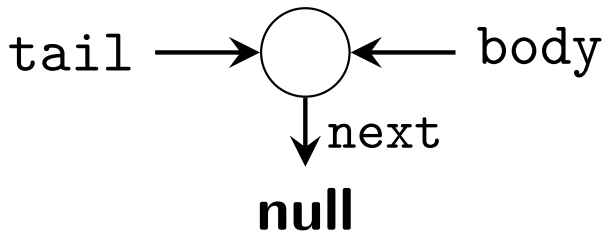
```
fun makeList(len) {
  tail := new N();
  tail.next := null;    //⇐
  body := tail;
  while len > 0 {
    t := body;
    body := new N();
    body.next := t;
    len := len - 1;
  }
  list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

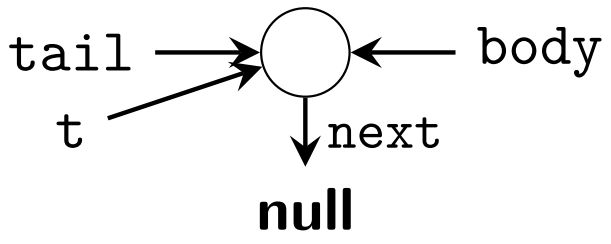
```
fun makeList(len) {
  tail := new N();
  tail.next := null;
  body := tail;          //⇐
  while len > 0 {
    t := body;
    body := new N();
    body.next := t;
    len := len - 1;
  }
  list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

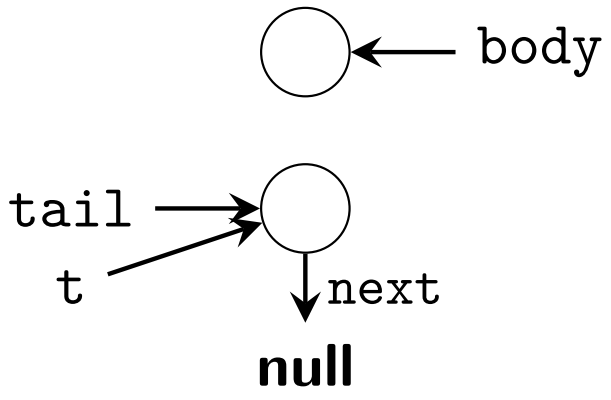
```
fun makeList(len) {
  tail := new N();
  tail.next := null;
  body := tail;
  while len > 0 {
    t := body;          //⇐
    body := new N();
    body.next := t;
    len := len - 1;
  }
  list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

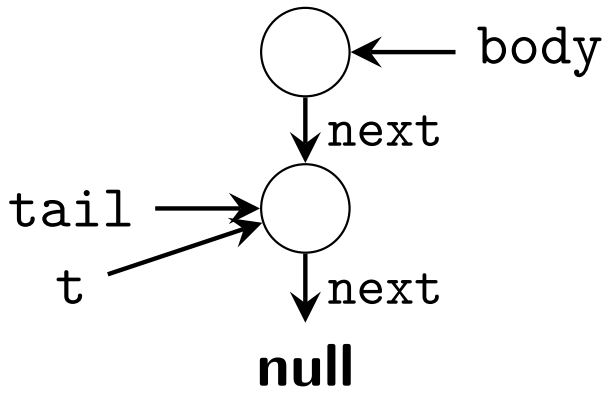
```
fun makeList(len) {
  tail := new N();
  tail.next := null;
  body := tail;
  while len > 0 {
    t := body;
    body := new N();    //⇐
    body.next := t;
    len := len - 1;
  }
  list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

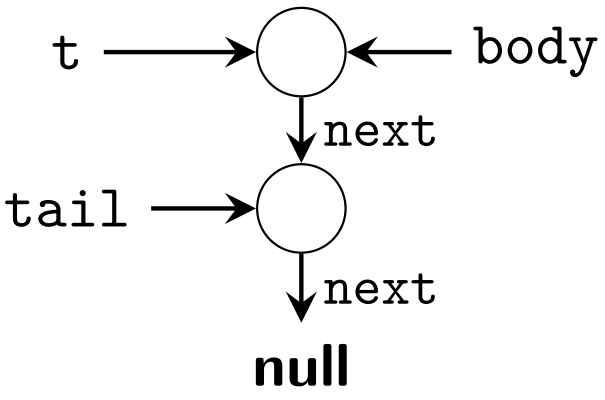
```
fun makeList(len) {
  tail := new N();
  tail.next := null;
  body := tail;
  while len > 0 {
    t := body;
    body := new N();
    body.next := t;     //⇐
    len := len - 1;
  }
  list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

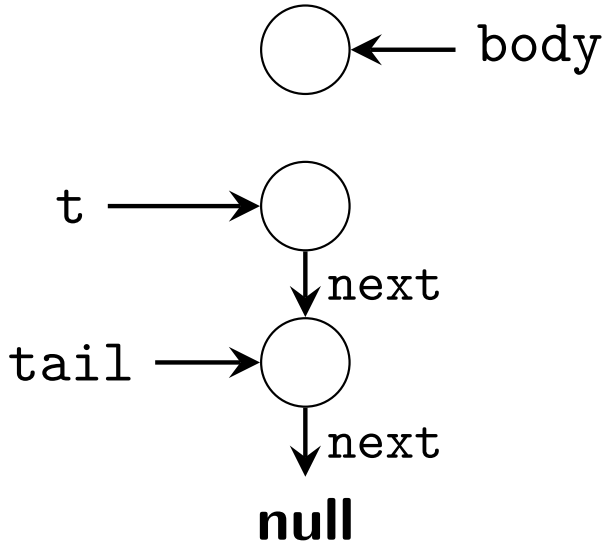
```
fun makeList(len) {
  tail := new N();
  tail.next := null;
  body := tail;
  while len > 0 {
    t := body;          //⇐
    body := new N();
    body.next := t;
    len := len - 1;
  }
  list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

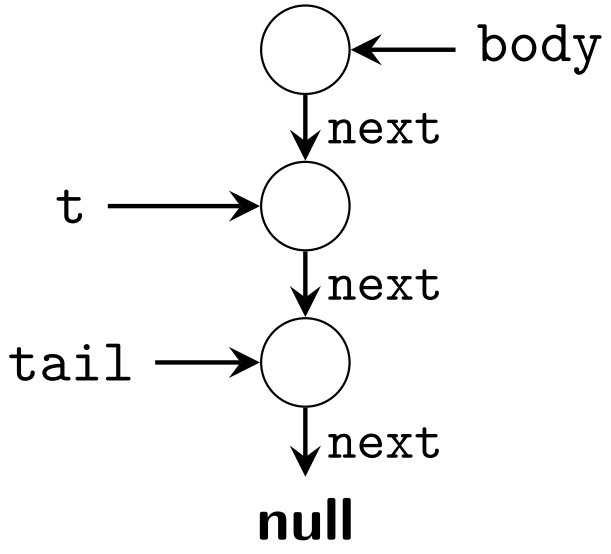
```
fun makeList(len) {
  tail := new N();
  tail.next := null;
  body := tail;
  while len > 0 {
    t := body;
    body := new N();    //⇐
    body.next := t;
    len := len - 1;
  }
  list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

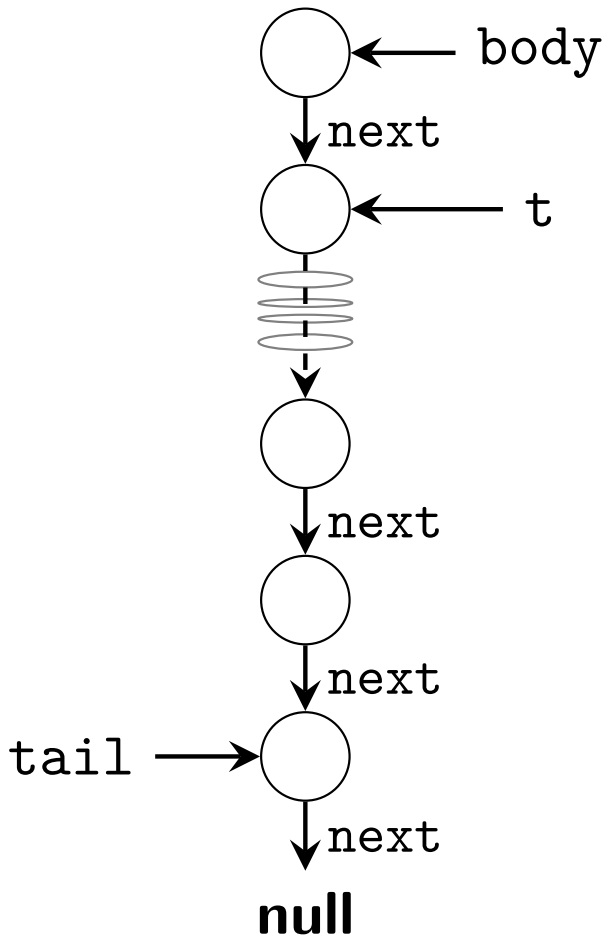
```
fun makeList(len) {
  tail := new N();
  tail.next := null;
  body := tail;
  while len > 0 {
    t := body;
    body := new N();
    body.next := t;     //⇐
    len := len - 1;
  }
  list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

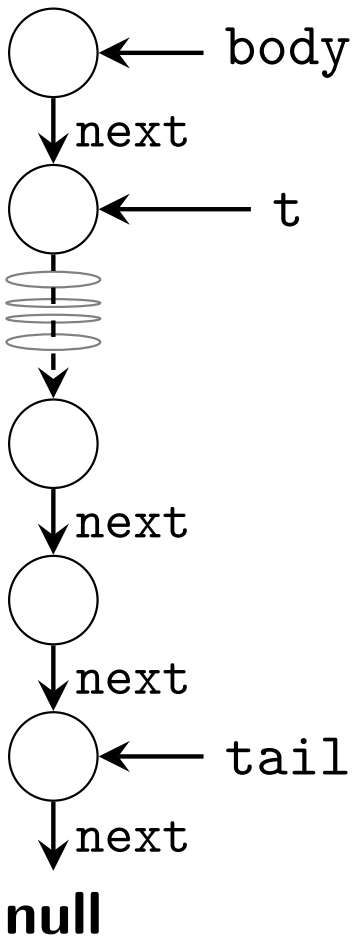
```
fun makeList(len) {
  tail := new N();
  tail.next := null;
  body := tail;
  while len > 0 {
    t := body;            //⇐
    body := new N();      //⇐
    body.next := t;       //⇐
    len := len - 1;
  }
  list := new N();
  list.head := body;
  list.tail := tail;
  return list;
}
```



Example

Teal-2

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

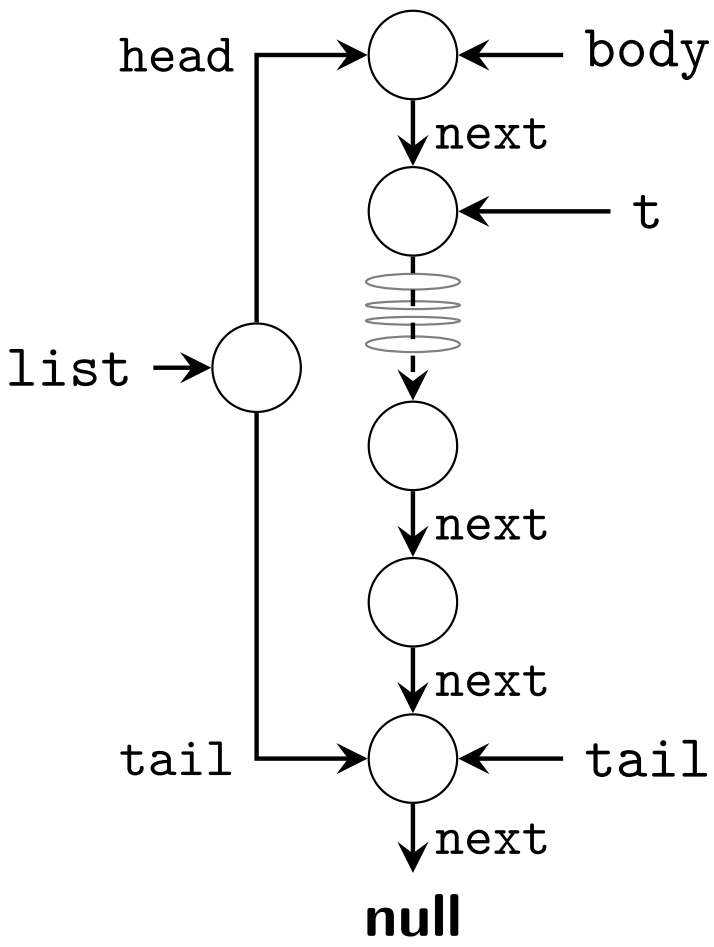


Managing Heap Graphs

- Size of Concrete Heap Graphs is unbounded
- **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:

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

- 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
- **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

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

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

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

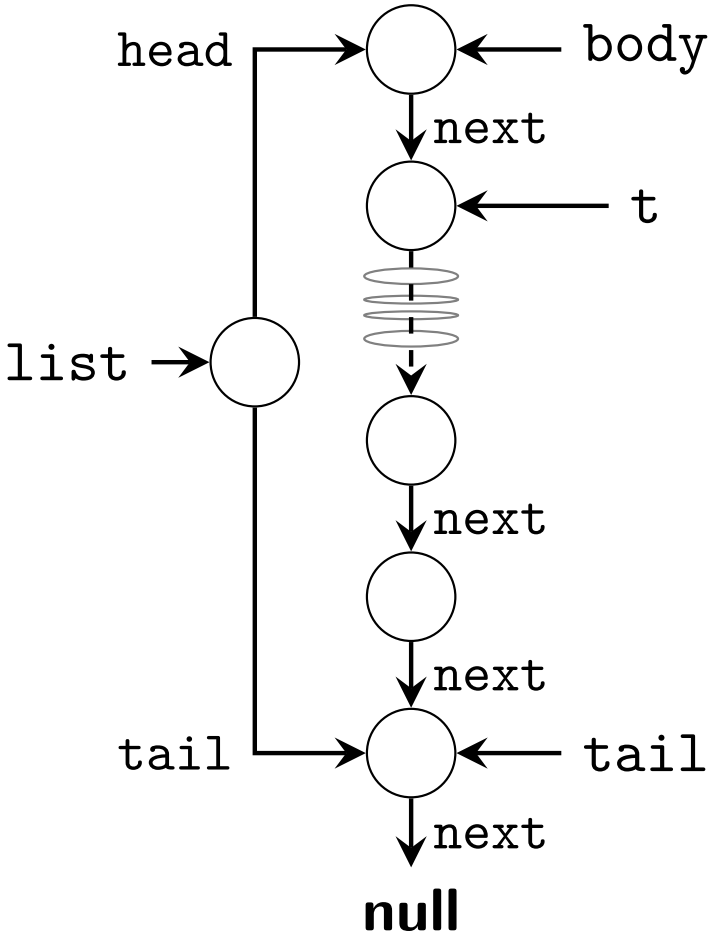
$$\begin{array}{lcl} (\rightarrowtriangle) & : & \mathcal{P}(Var) \rightarrow \mathcal{P}(MemLoc) \\ (\bar{\rightarrowtriangle}) & : & \mathcal{P}(MemLoc) \times \mathcal{P}(Field) \rightarrow \mathcal{P}(MemLoc) \end{array}$$

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

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

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

Summary Nodes and Edges



Notation:

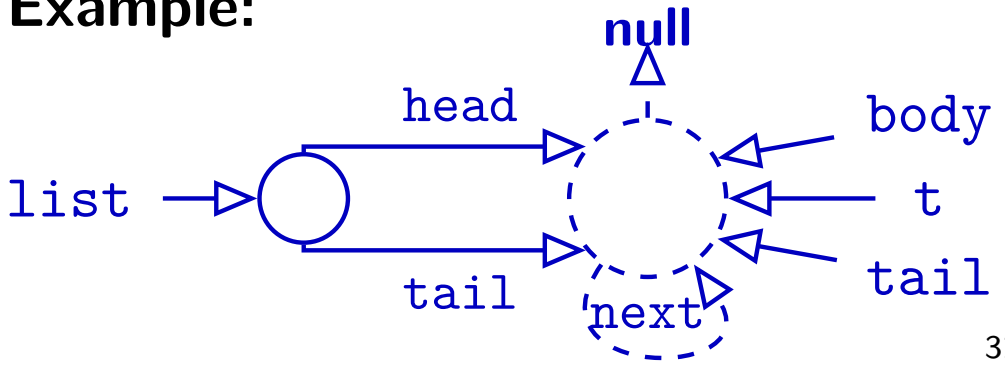
- Abstract node $N \subseteq MemLoc$:
  - $|N| = 1$: *precise*:
  - $|N| > 1$: *summary*:
- Consider edge $V \rightarrow L$:
  - $|V| = 1$: *precise*:

    $V \longrightarrow L$

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

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

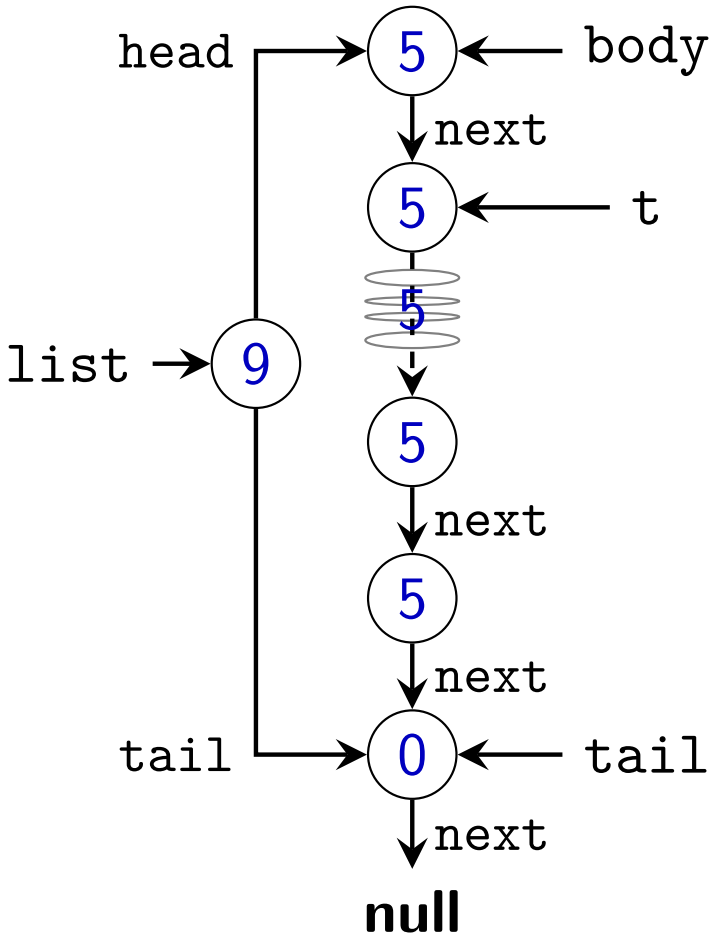
Example:



Summaries from Allocation Sites

Teal-2

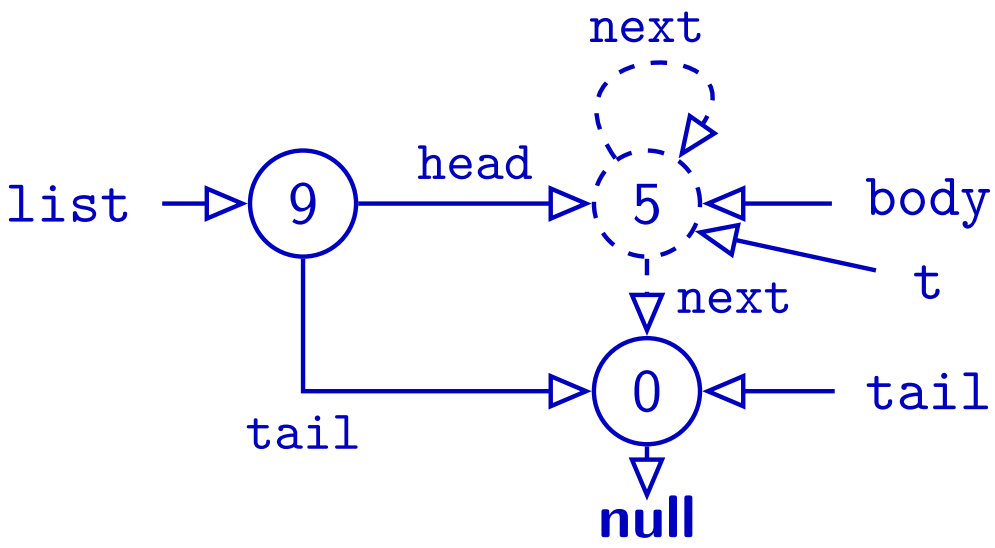
```
fun makeList(len) {
[0]  tail := new N();
[1]  tail.next := null;
[2]  body := tail;
[3]  while len > 0 {
[4]    t := body;
[5]    body := new N();
[6]    body.next := t;
[7]    len := len - 1;
[8]  }
[9]  list := new N();
[10] list.head := body;
[11] list.tail := tail;
[12] return list;
}
```



Summaries from Allocation Sites

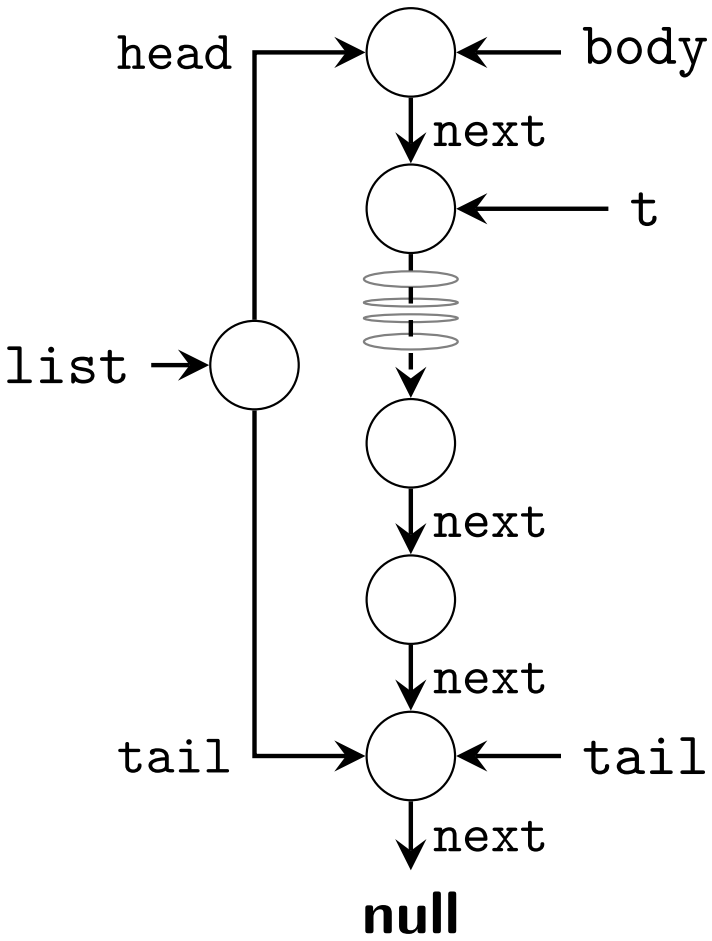
Teal-2

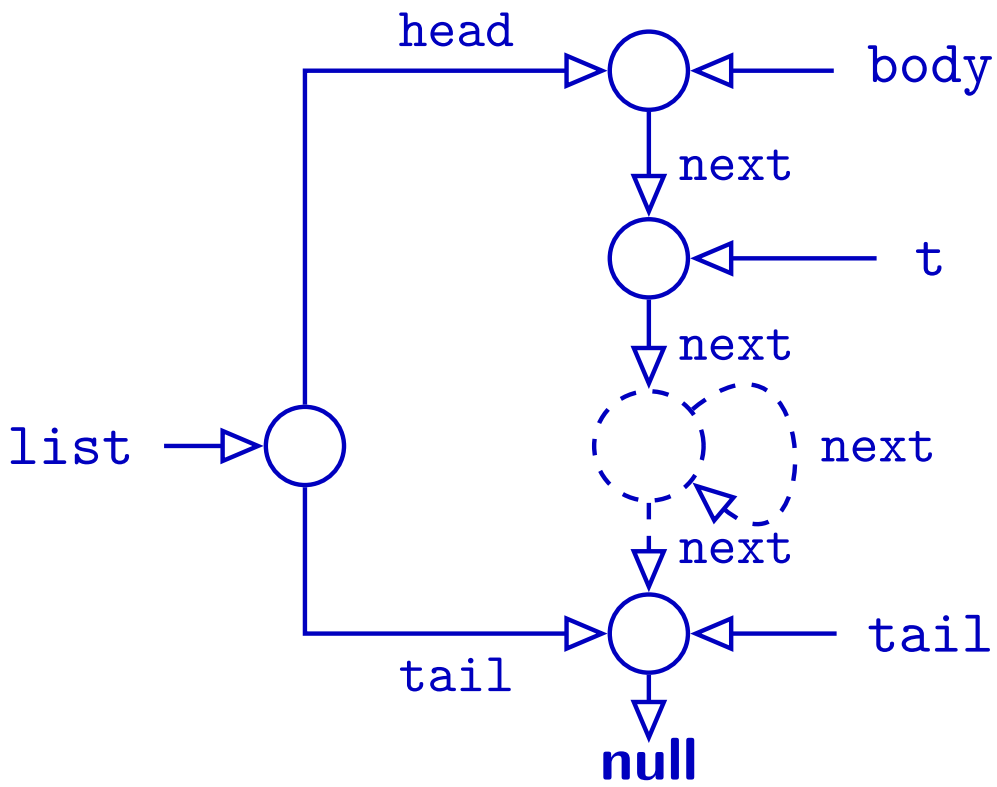
```
fun makeList(len) {
[0]  tail := new N();
[1]  tail.next := null;
[2]  body := tail;
[3]  while len > 0 {
[4]    t := body;
[5]    body := new N();
[6]    body.next := t;
[7]    len := len - 1;
[8]  }
[9]  list := new N();
[10] list.head := body;
[11] list.tail := tail;
[12] return list;
}
```



- Summarise *MemLoc* allocated at same program location

Variable-Based Summaries





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

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

  ...

Other Summary Techniques

- General idea: Map $\mathcal{P}(MemLoc)$ 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
- Summarisation techniques:
  - **Allocation-Site Based**: summarise nodes allocated at same pointin program
  - k-**Limiting**: Set bound on some property P: no more than k Ps allowed
  - **Variable-Based**: summarise data not pointed to by variables or pointed to by the same variables (**May** analysis)
  - Many combinations / extensions conceivable