Shape Analysis

Rupesh Nasre.

CS6843 Program Analysis
IIT Madras
Jan 2019

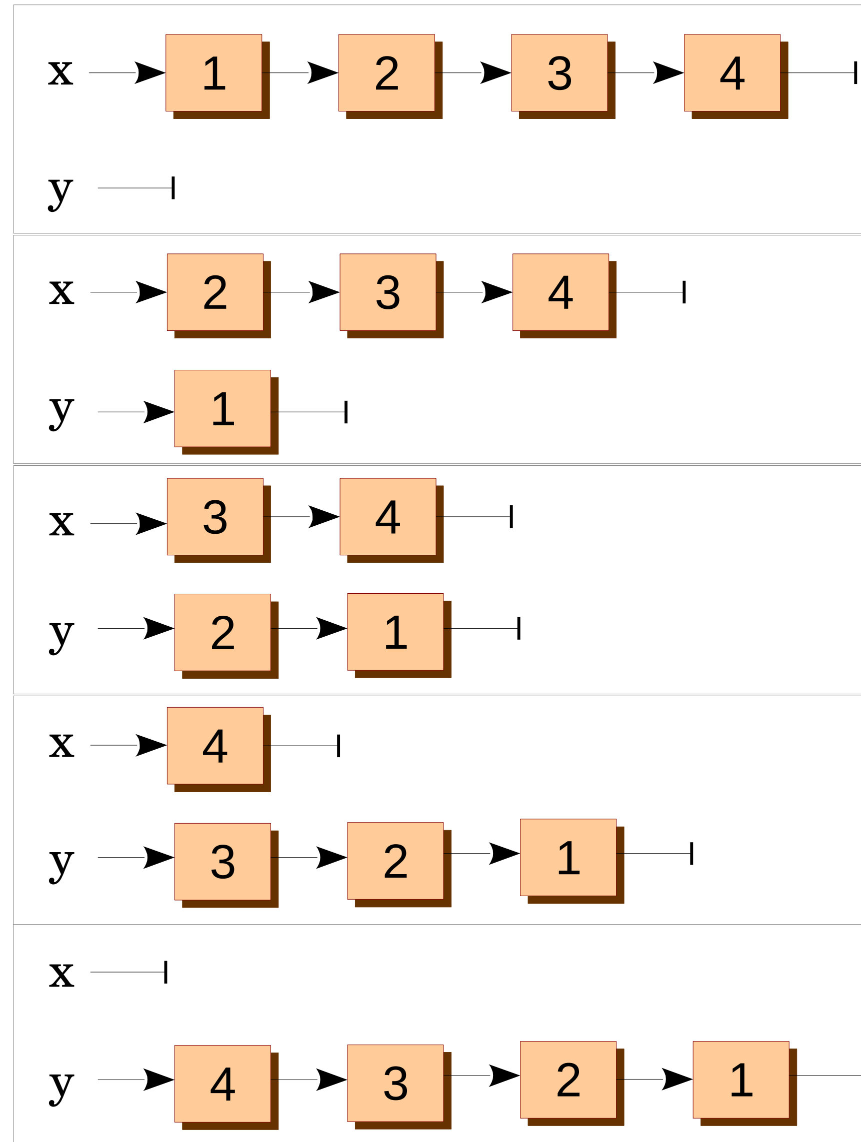
Learning Outcomes

- Limitations of pointer analysis
- Identify lists
- Identify trees, DAGs, cyclic graphs
- Identifying rotations
- List reversal and other transformations

Limitations of Pointer Analysis

```
listReverse(List x) {
  assert("x is an acyclic singly linked list");

  for (y = null; x;) {
      t = y;
      y = x;
      x = x→next;
      y→next = t;
  }
  x = y;
  t = null;
  y = null;
}
```

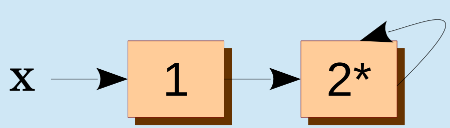
We want to check if x points to a singly linked list at the end of listReverse.
That is,
x → {4}, 4.next → {3}, …, 1.next → {null}

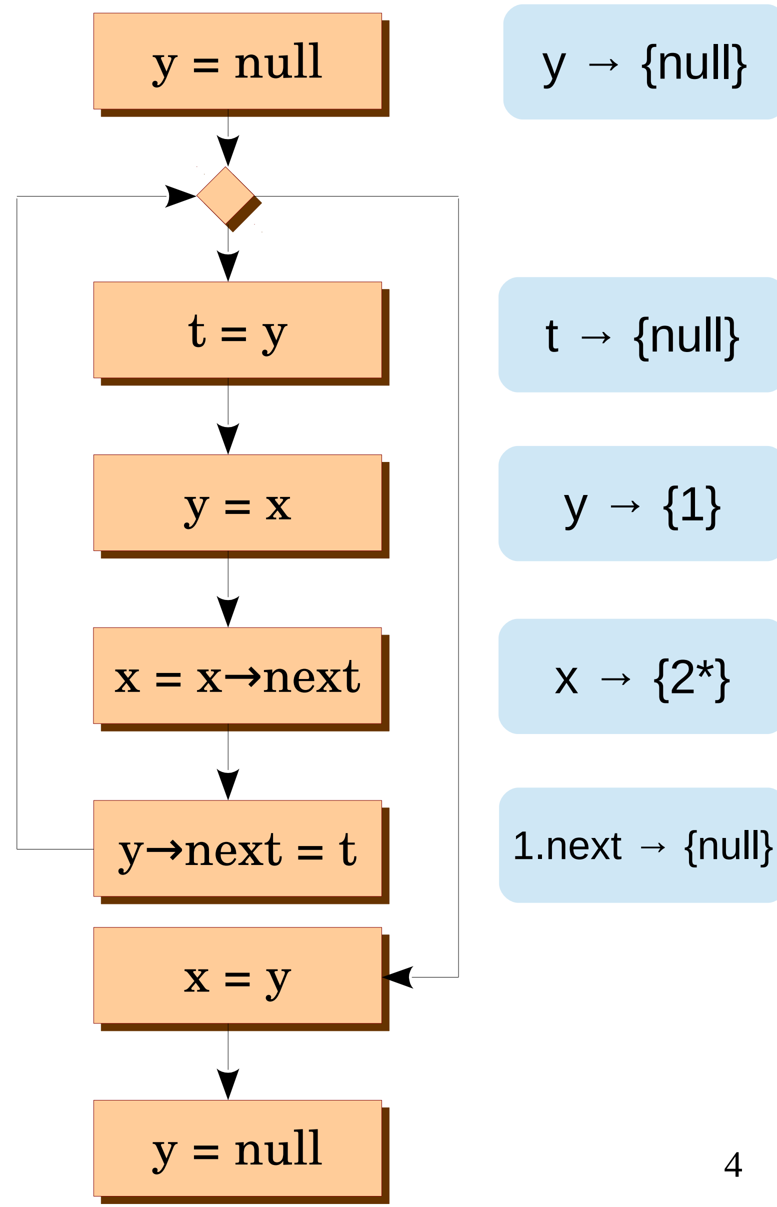


Limitations of Pointer Analysis

```
listReverse(List x) {
  assert("x is an acyclic singly linked list");

  for (y = null; x;) {
      t = y;
      y = x;
      x = x→next;
      y→next = t;
  }
  x = y;
  t = null;
  y = null;
}
```

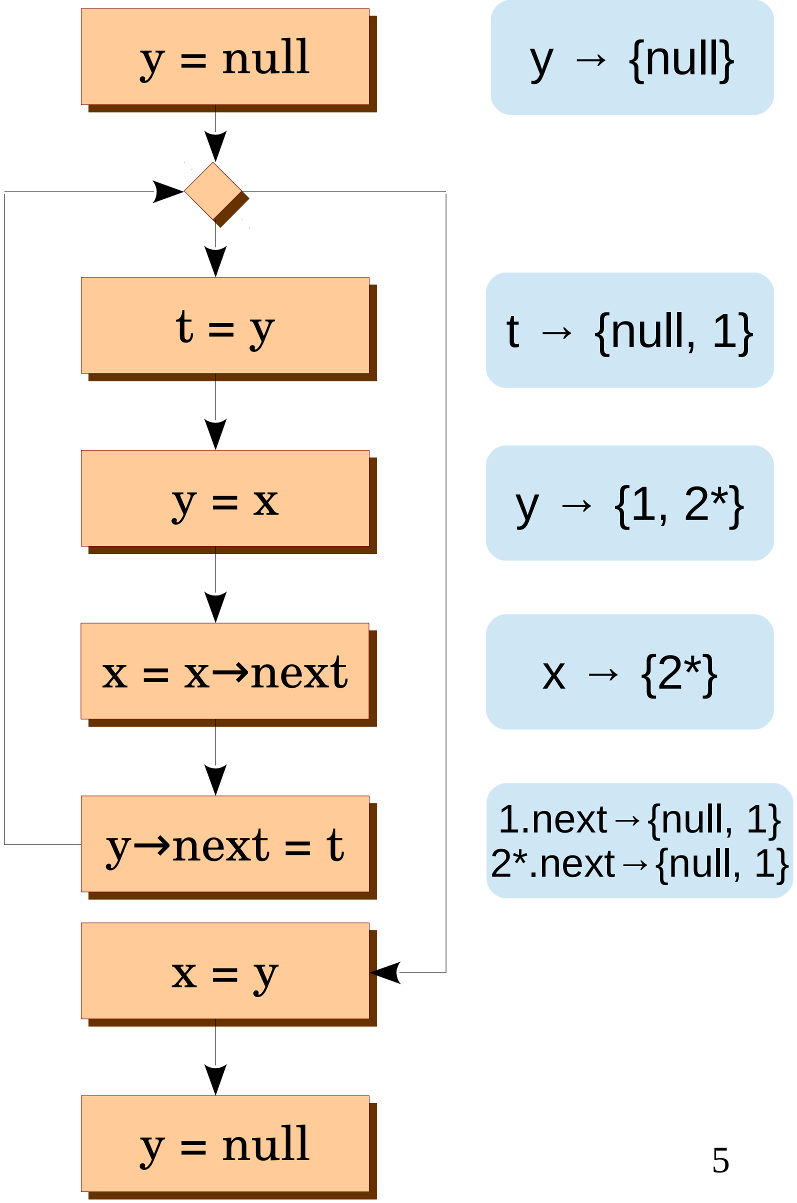
We model the list as a two node structure.





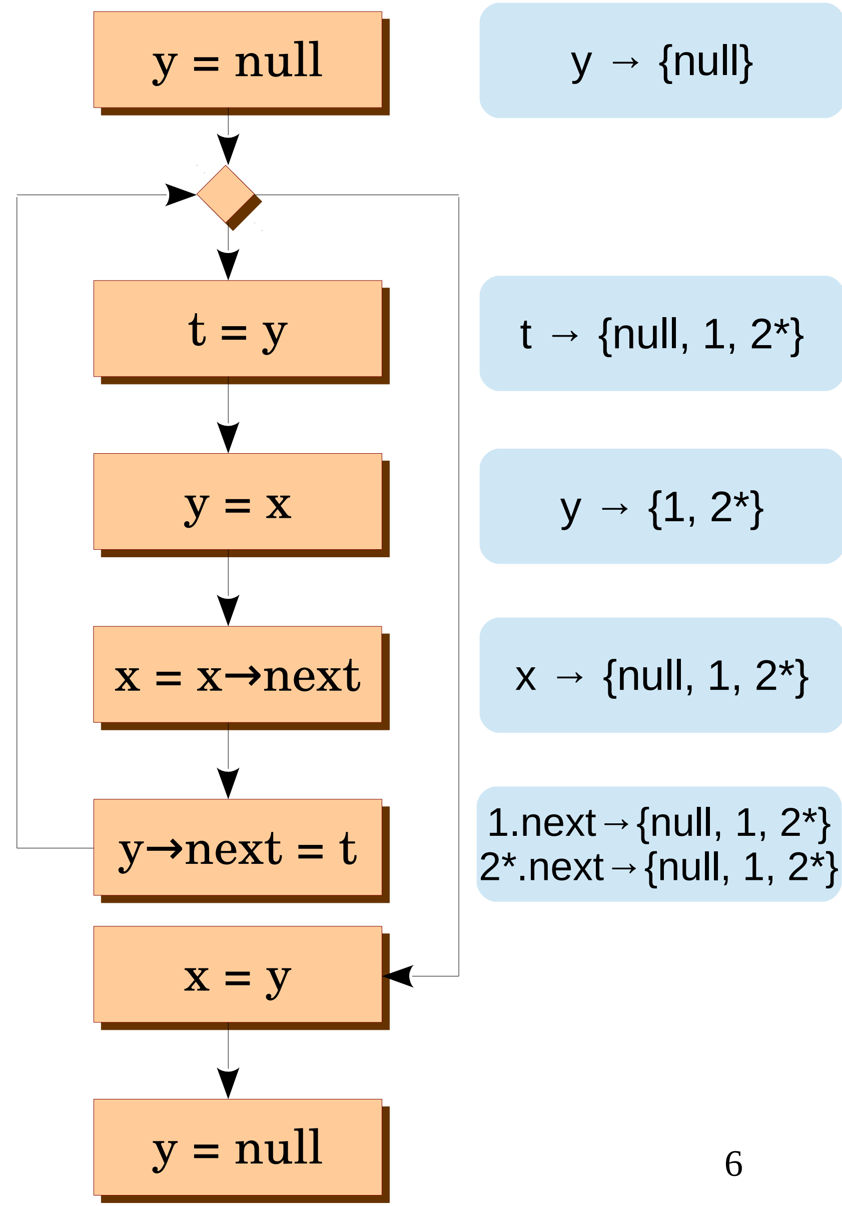
Limitations of Pointer Analysis

```
listReverse(List x) {
  assert("x is an acyclic singly linked list");

  for (y = null; x;) {
        t = y;
        y = x;
        x = x→next;
        y→next = t;
  }
  x = y;
  t = null;
  y = null;
}
```



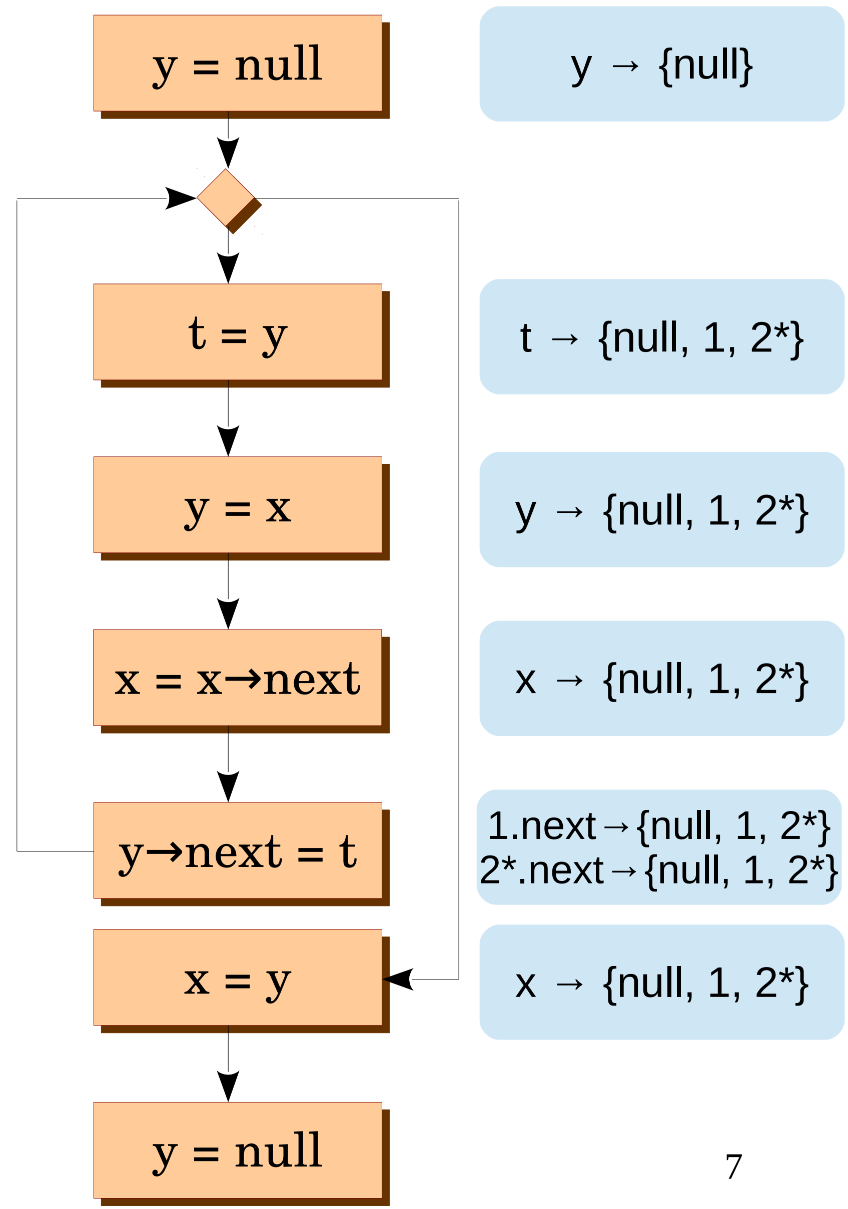
Limitations of Pointer Analysis

```
listReverse(List x) {
  assert("x is an acyclic singly linked list");

  for (y = null; x;) {
        t = y;
        y = x;
        x = x→next;
        y→next = t;
  }
  x = y;
  t = null;
  y = null;
}
```



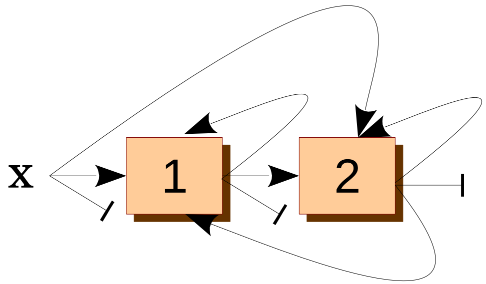
Limitations of Pointer Analysis

```
listReverse(List x) {
  assert("x is an acyclic singly linked list");

  for (y = null; x;) {
        t = y;
        y = x;
        x = x→next;
        y→next = t;
  }
  x = y;
  t = null;
  y = null;
}
```



Limitations of Pointer Analysis

```
listReverse(List x) {
  assert("x is an acyclic singly linked list");

  for (y = null; x;) {
        t = y;
        y = x;
        x = x→next;
        y→next = t;
  }
  x = y;
  t = null;
  y = null;
}
```



y → {null}

t → {null, 1, 2*}

y → {null, 1, 2*}

x → {null, 1, 2*}

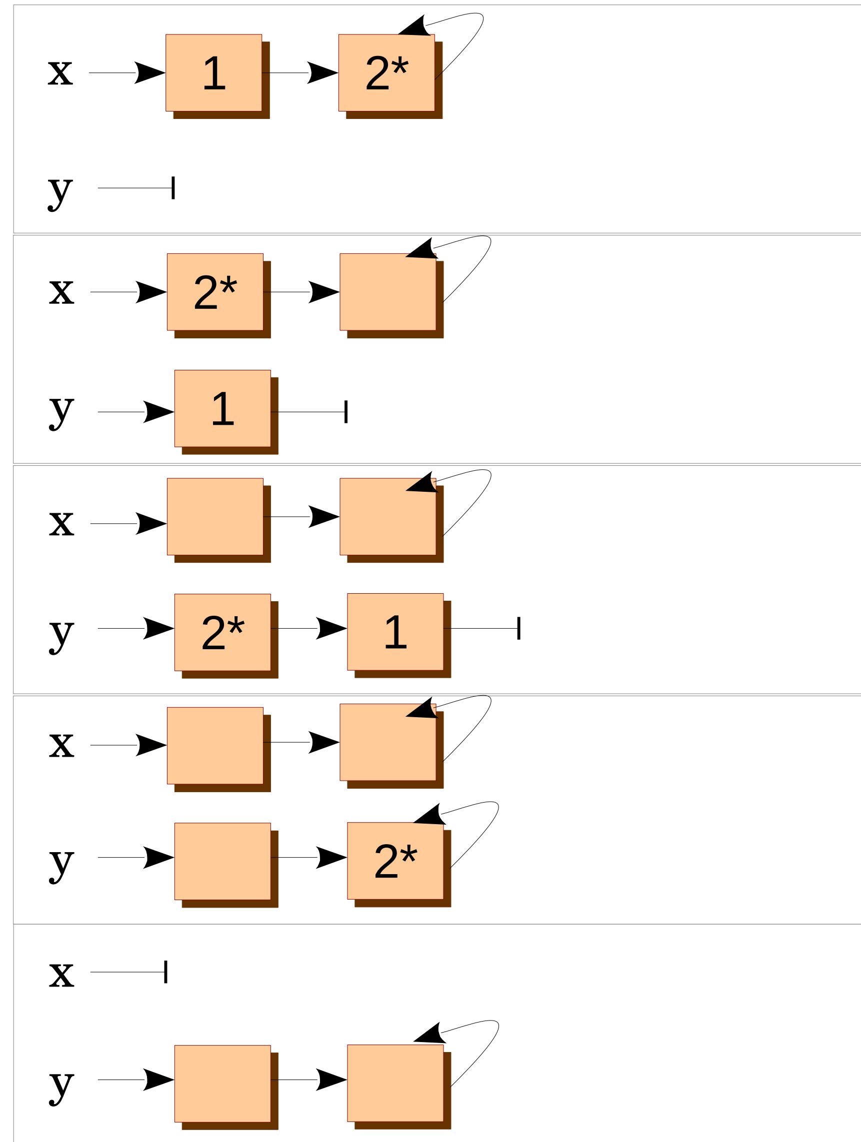
1.next → {null, 1, 2*}
2*.next → {null, 1, 2*}

x → {null, 1, 2*}

Shape Analysis

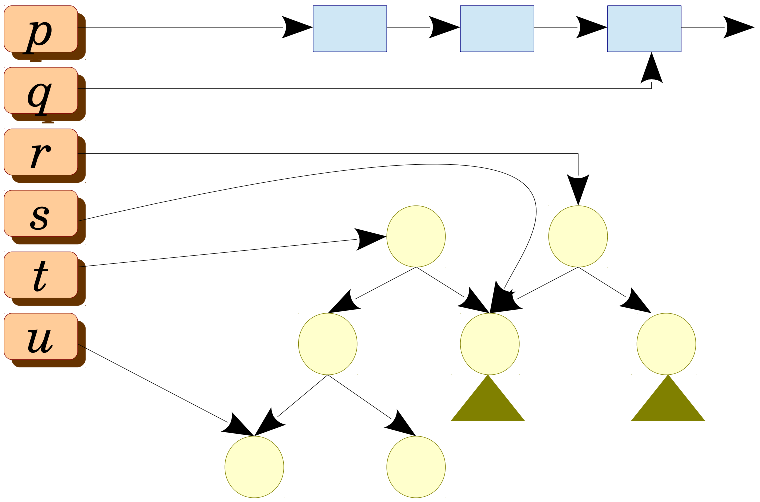
```
listReverse(List x) {
  assert("x is an acyclic singly linked list");

  for (y = null; x;) {
       t = y;
       y = x;
       x = x→next;
       y→next = t;
  }
  x = y;
  t = null;
  y = null;
}
```

Maintain additional information with 2* that it is acyclic.
Use the fact that node removal maintains acyclicity.



Shape Analysis

- Identify structural / topological properties of a data structure under manipulation.
- Usually categorized as slist, tree, DAG or cycle.
- Precision reduces along slist → tree → DAG → cycle.
- **Challenges:**
  - Size of the data structure is statically unknown.
  - Same pointer iterates through various nodes.
  - Shape needs to be determined from program variables.



Classwork

- Identify shapes of data structures.

```
n = sizeof(node);
p = malloc(n);
q = p;
q→data = 20;
q→next = NULL;
```

```
n = sizeof(node);
p = malloc(n);
q = p;
q→data = 20;
q→next = p;
```

```
n = sizeof(node);
p = ~~createTree()~~; SarthakPandey();
q = p;
q→data = 20;
q→left = p→right;
```

Takeaways:

- Analysis is restricted to program-defined pointers.
- We cannot afford to represent the heap!
- Meanings of function or variables names are useless for the analysis.

Classwork

- Identify shapes of data structures.

```
q = malloc(...);
q→f = malloc(...);
t = malloc(...);
t→g = malloc(...);
x = t→g;
x→g = q→f;
p = q→f;
p→f = t;
```

There could be a long chain of links from t to q→f.

Tree, DAG, Cycle?

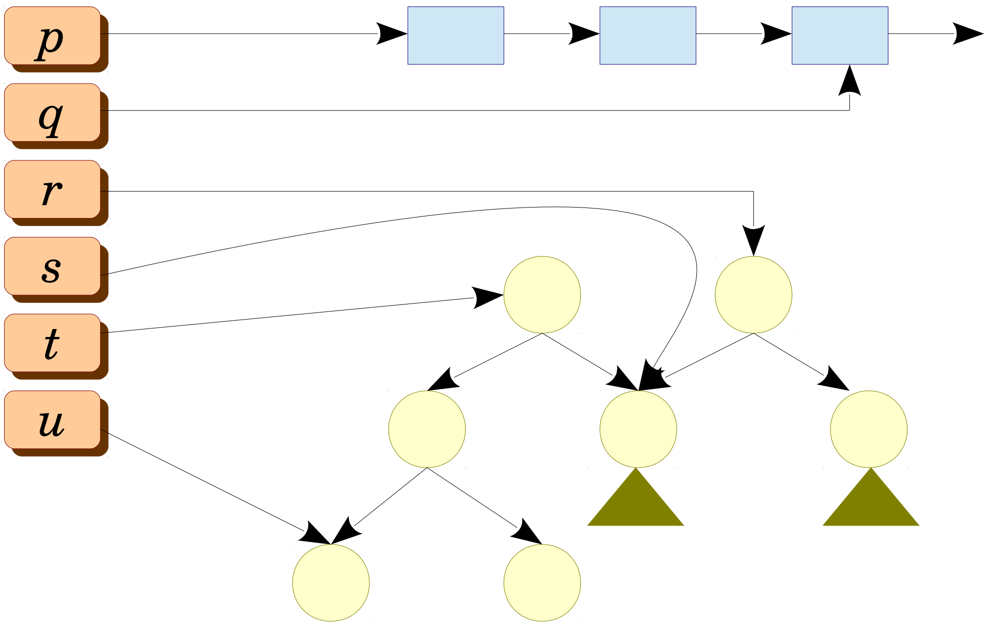
Heap abstraction

- Proposed by Ghiya and Hendren
- Maintains three data structures:
  - **Shape** reachable from a pointer.
  - **Interference** matrix: encodes common reachability
  - **Direction** matrix: encodes direct reachability
- Performs iterative data-flow analysis to update shape, I and D information

Takeaways:

- *Analysis is restricted to program-defined pointers.*
- *We cannot afford to represent the heap!*
- *Meanings of function or variables names are useless for the analysis.*

Tree, DAG, Cycle?



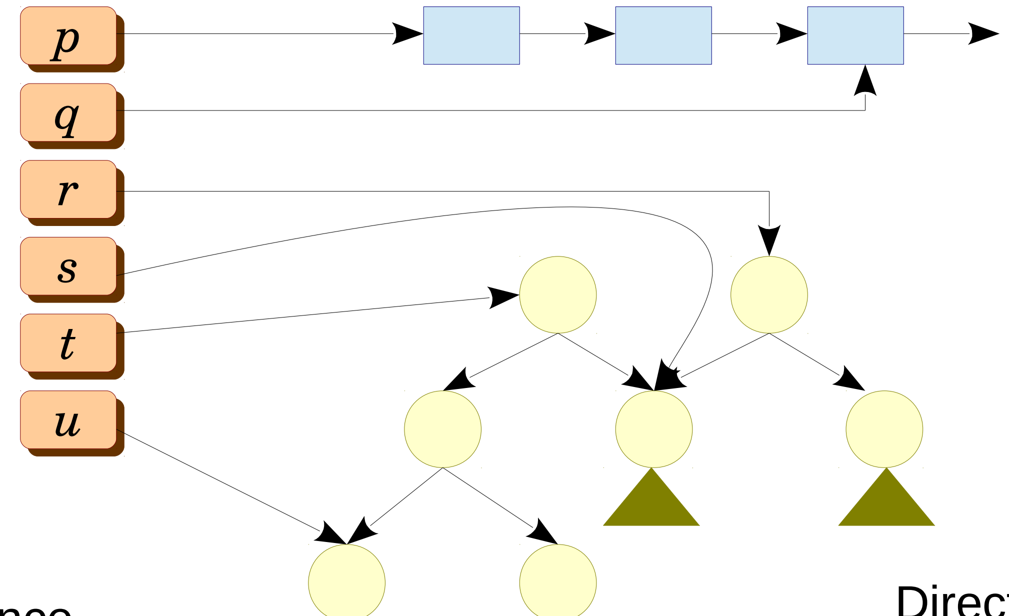
Interference(x, y)

Whether x and y have a common reachable node.

Direction(x, y)

Whether x has a path to the node pointed to by y.

Tree, DAG, Cycle?



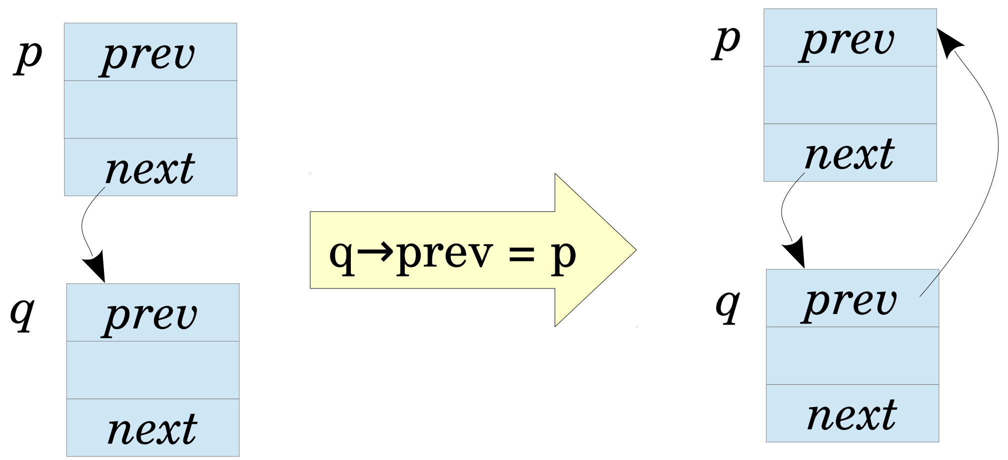
Interference

	p	q	r	s	t	u
p	1	**1**	0	0	0	0
q		1	0	0	0	0
r			1	**1**	**1**	0
s				1	**1**	0
t					1	**1**
u						1

Direction

	p	q	r	s	t	u
p	1	**1**	0	0	0	0
q	**0**	1	0	0	0	0
r	0	0	1	**1**	**0**	0
s	0	0	**0**	1	**0**	0
t	0	0	**0**	**1**	1	**1**
u	0	0	0	0	**0**	1

Shape Estimation Example



D[p][q] = 1, D[q][p] = 0
p.shape = Tree
q.shape = Tree

D[p][q] = 1, D[q][p] = **1**
p.shape = **Cycle**
q.shape = **Cycle**

Classwork:
- What is the data-flow information?
- Write down the statements of interest for this shape analysis.
- How are D, I and shape initialized?

Relevant Statements and Inference Rules

p = malloc(...) p = q p = q→f p = &(q→f) p = q op k p = null p->f = q p->f = null	Inference rules define local processing; they generate and kill data-flow facts. What are the gen and kill sets for these relevant statements?

Inference Rules

p = malloc(...)	D_kill = {D[p][s] ǀ D[p][s] == 1} U {D[s][p] ǀ D[s][p] == 1} I_kill = {I[p][s] ǀ I[p][s] == 1}
p = q p = q→f p = &(q→f) p = q op k p = null	D_gen = {D[p][p]} I_gen = {I[p][p]}
p->f = q p->f = null	p.shape = Tree

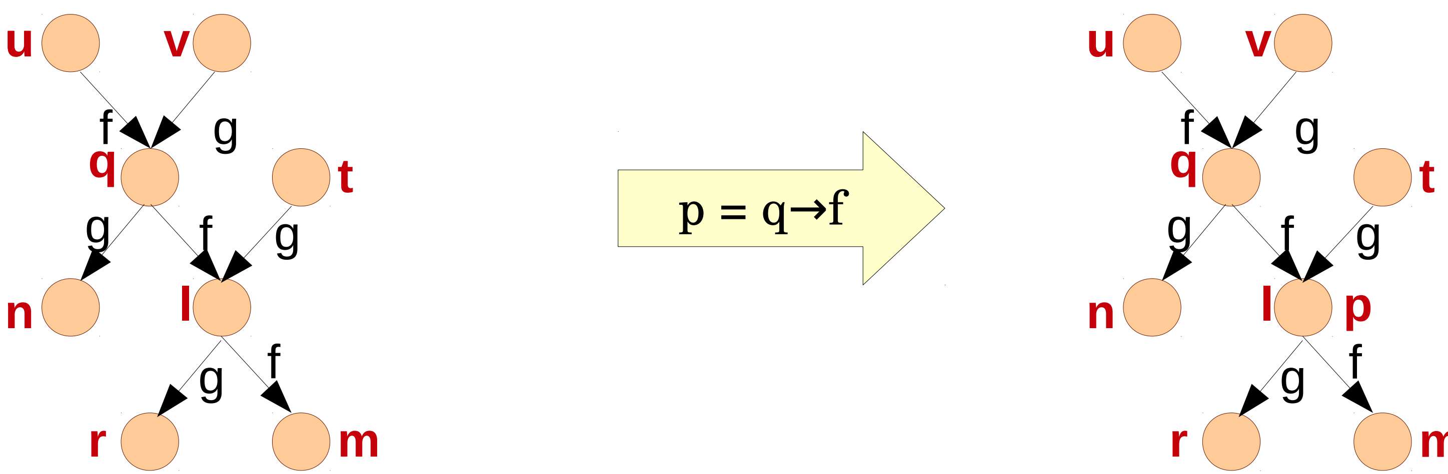
Inference Rules

p = malloc(...)	D_kill and I_kill sets same as for allocation statement.
p = q p = q→f p = &(q→f) p = q op k p = null	D_gen = {D[s][p] ∣ D[s][q] and s ≠ p} U {D[p][s] ∣ D[q][s] and s ≠ p} U {D[p][p] ∣ D[q][q]} I_gen = {I[p][s] ∣ I[q][s] and s ≠ p} U {I[p][p] ∣ I[q][q]}
p->f = q p->f = null	p.shape = q.shape

The implementation should create new D/I matrices from their current copies. In-situ update would lead to unsound or imprecise analysis.

Inference Rules

p = malloc(...)	D_kill and I_kill sets same as for allocation statement.
p = q **p = q→f** p = &(q→f) p = q op k p = null	D_gen = {D[s][p] ǀ D[s][q] and s ≠ p} U {D[p][s] ǀ D[q][s] and s ≠ p and s ≠ q} U {D[p][p] ǀ D[q][q]} U {D[p][q] ǀ q.shape == Cycle} I_gen = {I[p][s] ǀ I[q][s] and s ≠ p} U {I[p][p] ǀ I[q][q]}
p->f = q p->f = null	p.shape = q.shape

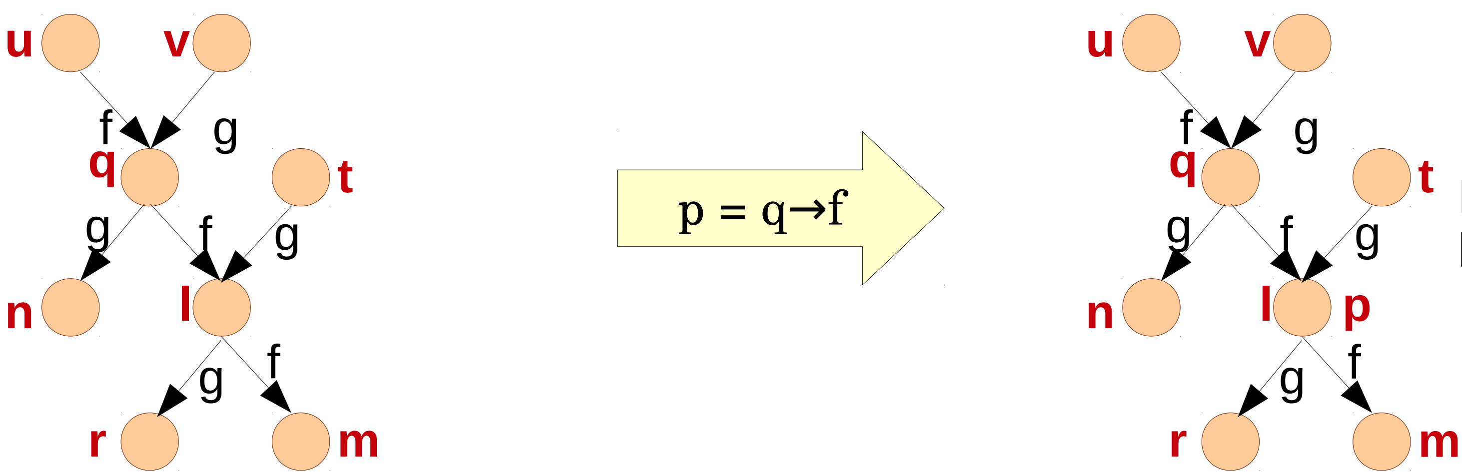


D[t][p] should be 1.

Which D_gen rule sets D[t][p]?

Inference Rules

p = malloc(…)	D_kill and I_kill sets same as for allocation statement.
p = q **p = q→f** p = &(q→f) p = q op k p = null p->f = q p->f = null	D_gen = {D[s][p] ∣ **I**[s][q] and s ≠ p} U {D[p][s] ∣ D[q][s] and s ≠ p and s ≠ q} U {D[p][p] ∣ D[q][q]} U {D[p][q] ∣ q.shape == Cycle} I_gen = {I[p][s] ∣ I[q][s] and s ≠ p} U {I[p][p] ∣ I[q][q]} p.shape = q.shape



I[t][q] was 1.

Hence, D[t][p] becomes 1.

Inference Rules

p = malloc(...)	D_kill and I_kill sets same as for allocation statement.
p = q **p = q→f** p = &(q→f) p = q op k p = null	D_gen = {D[s][p] ∣ **I**[s][q] and s ≠ p} U {D[p][s] ∣ D[q][s] and s ≠ p and s ≠ q} U {D[p][q] ∣ q.shape == Cycle} U {D[p][p] ∣ D[q][q]} I_gen = {I[p][s] ∣ I[q][s] and s ≠ p} U {I[p][p] ∣ I[q][q]}
p->f = q p->f = null	p.shape = q.shape

Classwork: Find D, I and shape for the following program fragment.

```
q = malloc(...);
p = q;
r = q→f;
```

Inference Rules

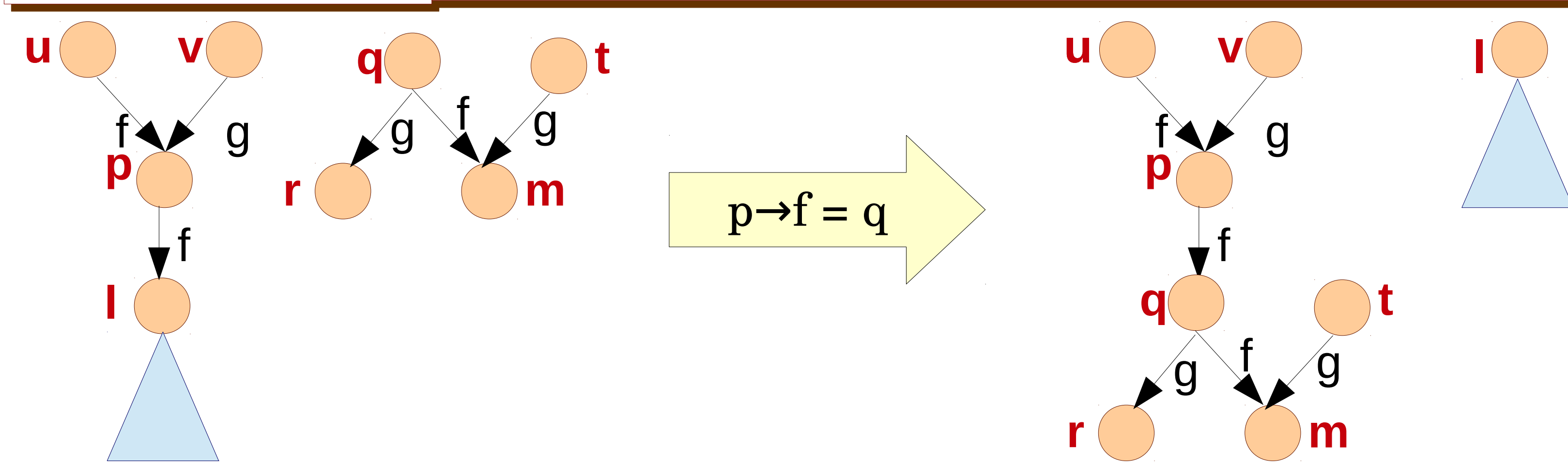
p = malloc(...) p = q p = q→f **p = &(q→f)** **p = q op k** p = null p->f = q p->f = null	Processing is the same as for p = q statement. This means the analysis loses field-sensitivity. A former work from IITK (Dasgupta, Karkare, Reddy) addresses this issue.

Inference Rules

p = malloc(…)	D_kill and I_kill sets same as for allocation statement.
p = q p = q→f p = &(q→f) p = q op k **p = null**	D_gen = { } I_gen = { } p.shape = Tree
p->f = q p->f = null	

Inference Rules

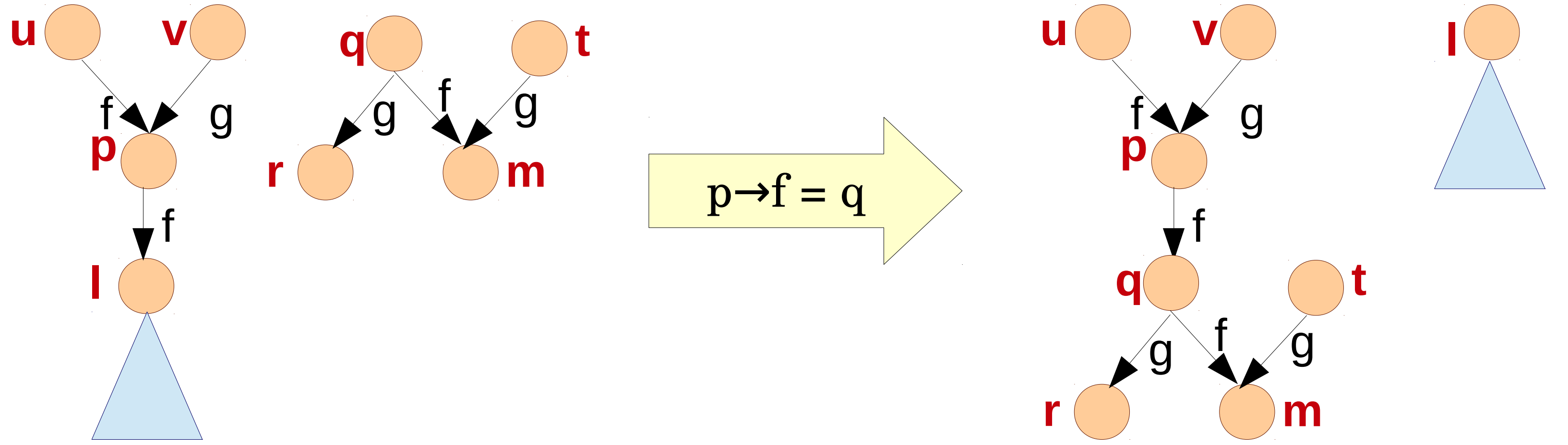
p = malloc(...)	D_kill = { }, I_kill = { }
p = q p = q→f p = &(q→f) p = q op k p = null	D_gen = {D[r][s] ∣ D[r][p] and D[q][s]} I_gen = {I[r][s] ∣ I[r][p] and I[q][s]}
p->f = q p->f = null	D[q][p] and D[s][q] ⇒ s.shape = Cycle D[q][p] and D[s][p] ⇒ s.shape = Cycle !D[q][p] and D[s][p] and I[s][q] and q.shape == Tree ⇒ s.shape = max(s.shape, DAG) !D[q][p] and D[s][p] and q.shape != Tree ⇒ s.shape = max(s.shape, q.shape)



Can you improve precision?

Inference Rules

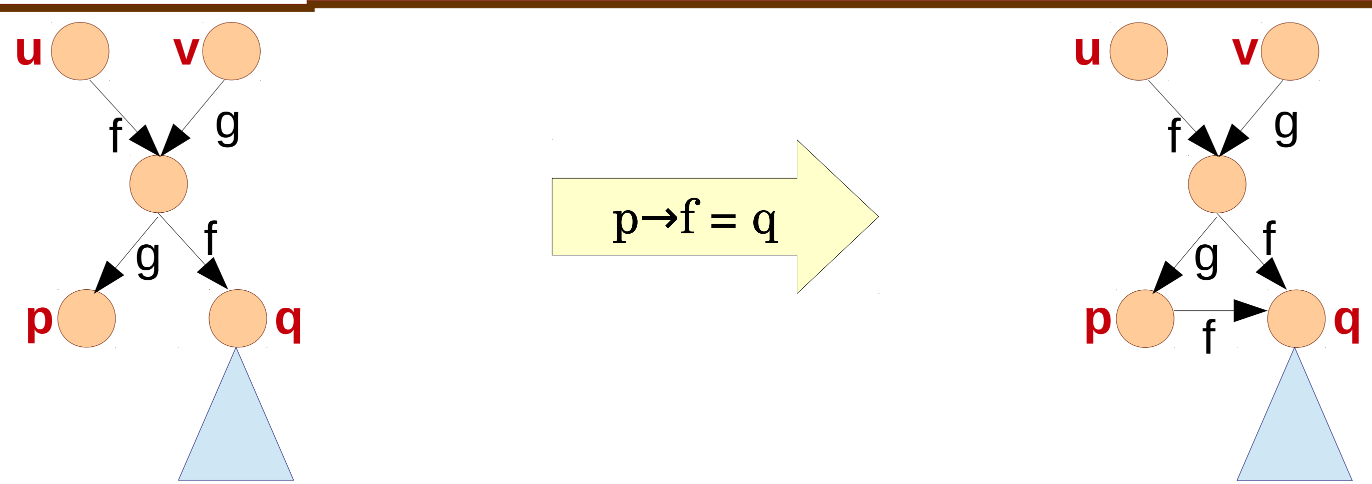
p = malloc(...)	D_kill = { }, I_kill = { }
p = q p = q→f p = &(q→f) p = q op k p = null	D_gen = {D[r][s] ∣ D[r][p] and D[q][s]} I_gen = {I[r][s] ∣ **D**[r][p] and I[q][s]}
p->f = q p->f = null	D[q][p] and D[s][q] ⇒ s.shape = Cycle D[q][p] and D[s][p] ⇒ s.shape = Cycle !D[q][p] and D[s][p] and I[s][q] and q.shape == Tree ⇒ s.shape = max(s.shape, DAG) !D[q][p] and D[s][p] and q.shape != Tree ⇒ s.shape = max(s.shape, q.shape)



For max() consider D[v][r] == 1.

Inference Rules

p = malloc(…)	D_kill = { }, I_kill = { }
p = q p = q→f p = &(q→f) p = q op k p = null	D_gen = {D[r][s] ǀ D[r][p] and D[q][s]} I_gen = {I[r][s] ǀ D[r][p] and I[q][s]}
p->f = q p->f = null	D[q][p] and D[s][q] ⇒ s.shape = Cycle D[q][p] and D[s][p] ⇒ s.shape = Cycle !D[q][p] and D[s][p] and I[s][q] and q.shape == Tree ⇒ s.shape = max(s.shape, DAG) !D[q][p] and D[s][p] and q.shape != Tree ⇒ s.shape = max(s.shape, q.shape)



Inference Rules

p = malloc(...)	D_kill = { }, I_kill = { }
p = q p = q→f p = &(q→f) p = q op k p = null	D_gen = {D[r][s] ∣ D[r][p] and D[q][s]} I_gen = {I[r][s] ∣ D[r][p] and I[q][s]}
p->f = q p->f = null	D[q][p] and D[s][q] ⇒ s.shape = Cycle D[q][p] and D[s][p] ⇒ s.shape = Cycle !D[q][p] and D[s][p] and I[s][q] and q.shape == Tree ⇒ s.shape = max(s.shape, DAG) !D[q][p] and D[s][p] and q.shape != Tree ⇒ s.shape = max(s.shape, q.shape)

	Tree	DAG	Cycle
Tree	Tree	DAG	Cycle
DAG	DAG	DAG	Cycle
Cycle	Cycle	Cycle	Cycle

max(shape1, shape2)

Inference Rules

p = malloc(...) p = q p = q→f p = &(q→f) p = q op k p = null p->f = q **p->f = null**	D_kill = { }, I_kill = { } D_gen = { } I_gen = { } No changes to the shape of p.

Classwork

```
r = malloc(10);
p = malloc(10);
p->f1 = null;
q = p->f2;
q = &(r->f2);
q->f2 = p;
```

Improvements

- Field-sensitivity
- Heap modeling
- Path-sensitivity

Summary

- Shape analysis helps several transforms.
- Existing techniques often trade off precision for efficiency.
- We are still far away from a precise and scalable analysis.