Shape Analysis

Rupesh Nasre.

CS6843 Program Analysis
IIT Madras
Jan 2016

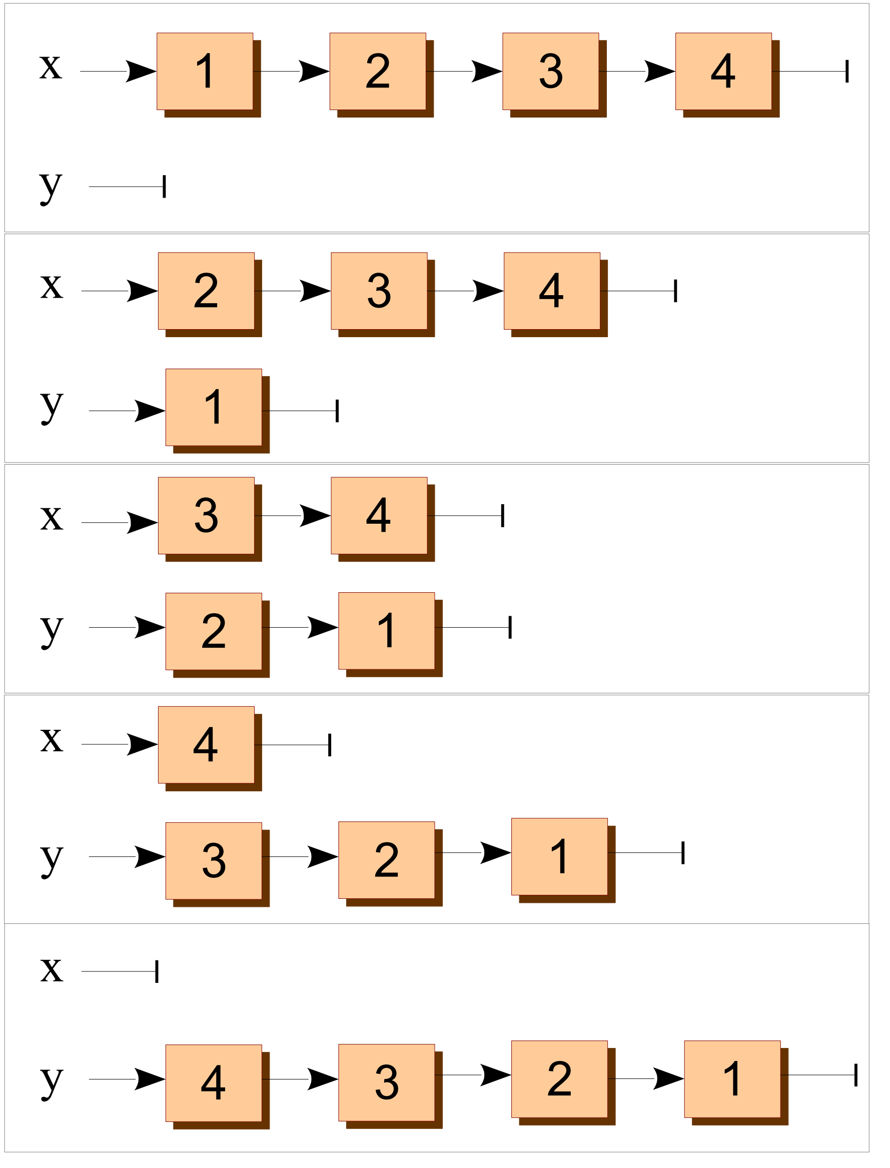
Outline

- 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;
}
```

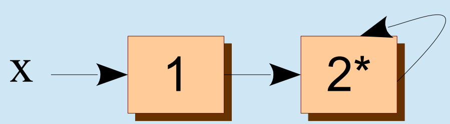
We want to check if x points to a singly linked list at the end of listReverse.
That is,
x → {4}, 4.next → {3}, …, 1.next → {null}

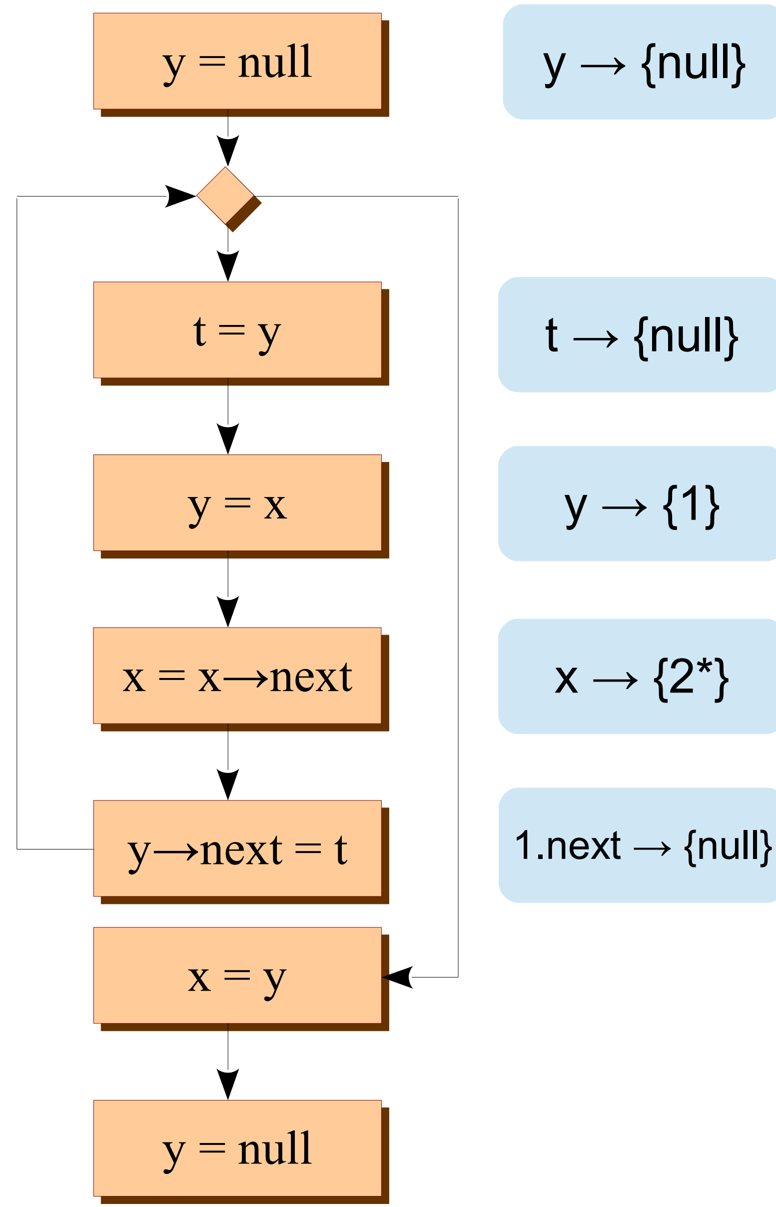


Limitations of Pointer Analysis

```
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;
}
```

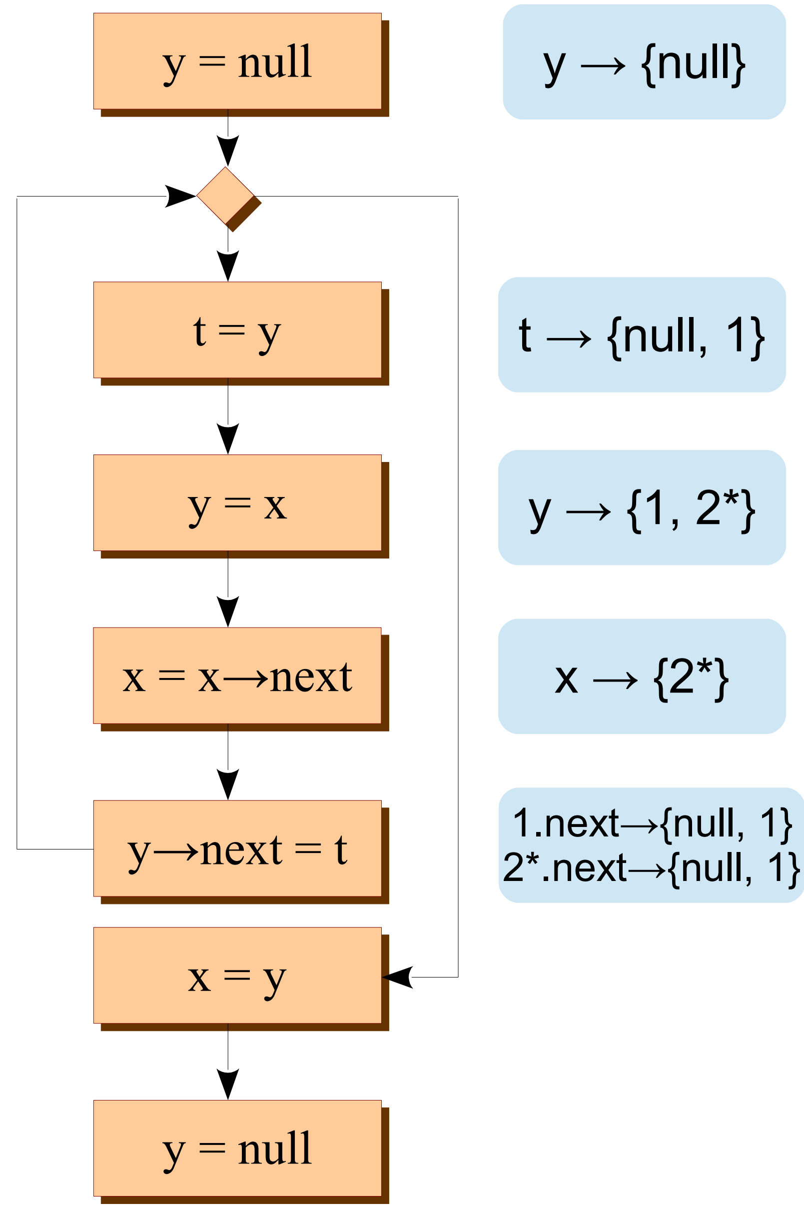
We model the list as a two node structure.





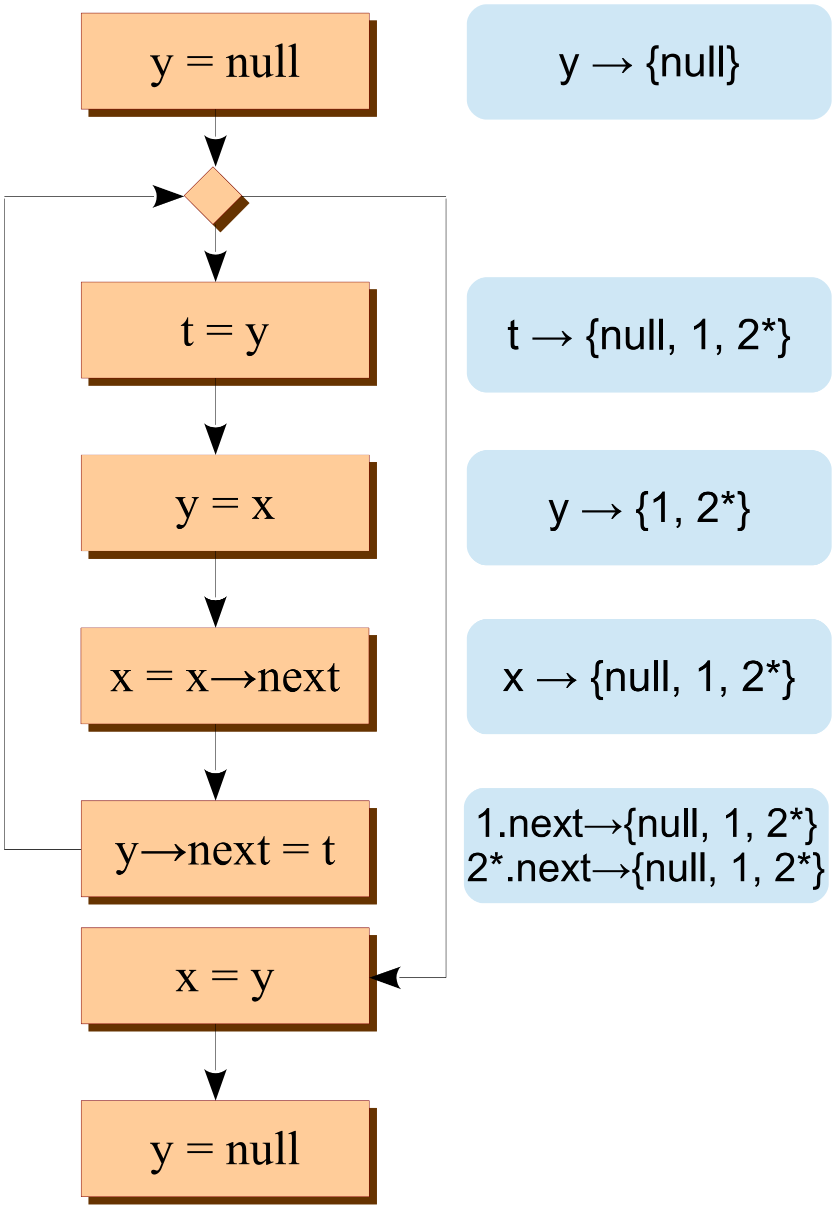
Limitations of Pointer Analysis

```
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;
}
```



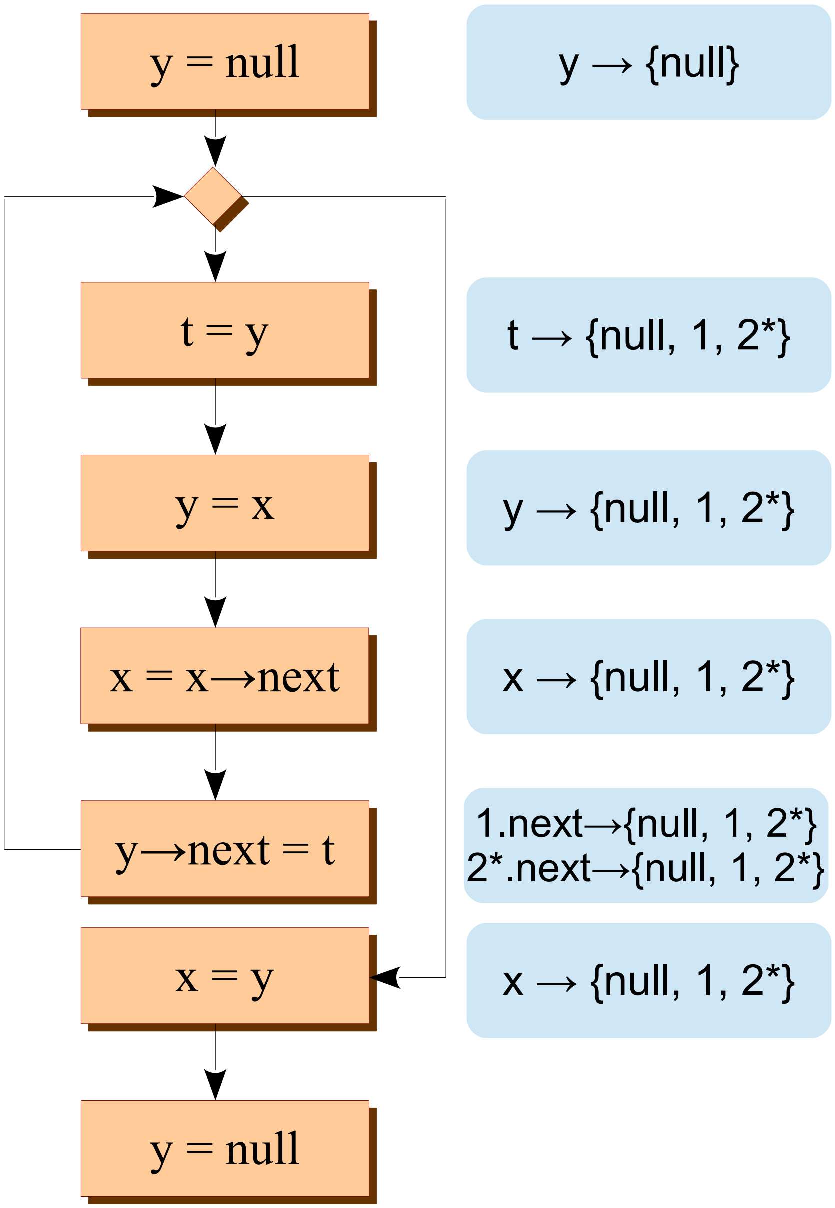
Limitations of Pointer Analysis

```
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;
}
```



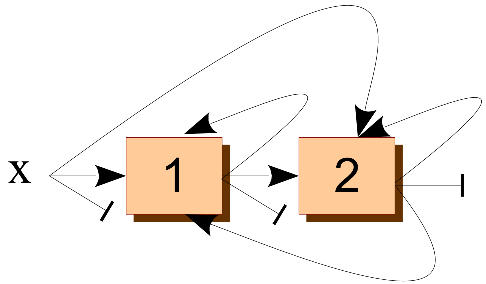
Limitations of Pointer Analysis

```
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*}

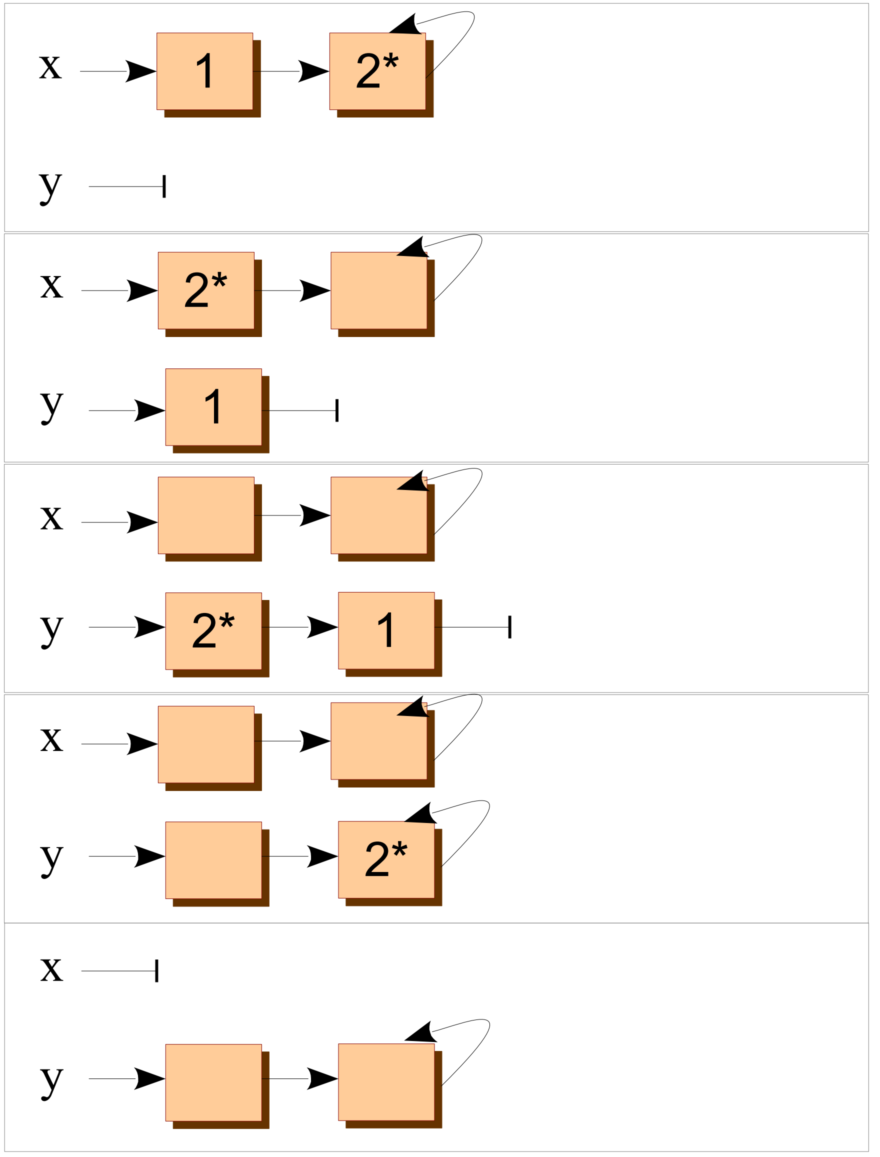
Shape Analysis

- Identify structural / topological properties of a data structure under manipulation.
- Usually categorized as slist, tree, DAG or cycle.
- Precision reduces along slist $\rightarrow$ tree $\rightarrow$ DAG $\rightarrow$ cycle.

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;
}
```

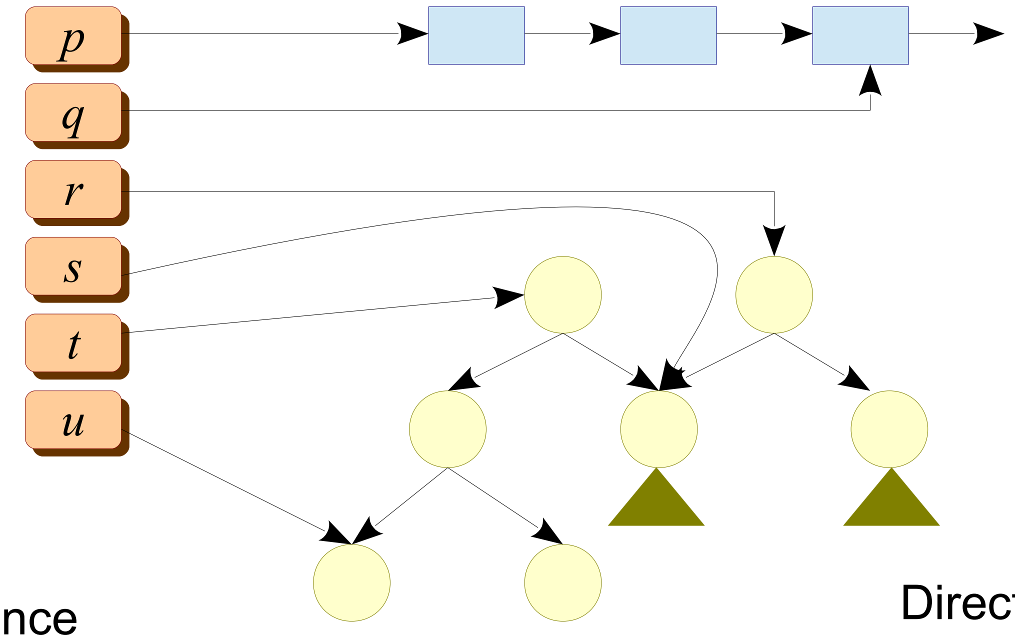
Maintain additional information with 2* that it is acyclic.
Use the fact that node removal maintains acyclicity.



Tree, DAG, Cycle?

- Maintains three data structures:
  - Interference matrix: encodes common reachability
  - Direction matrix: encodes direct reachability
  - Shape
- Performs iterative data-flow analysis to update D, I and shape information

Tree, DAG, Cycle?



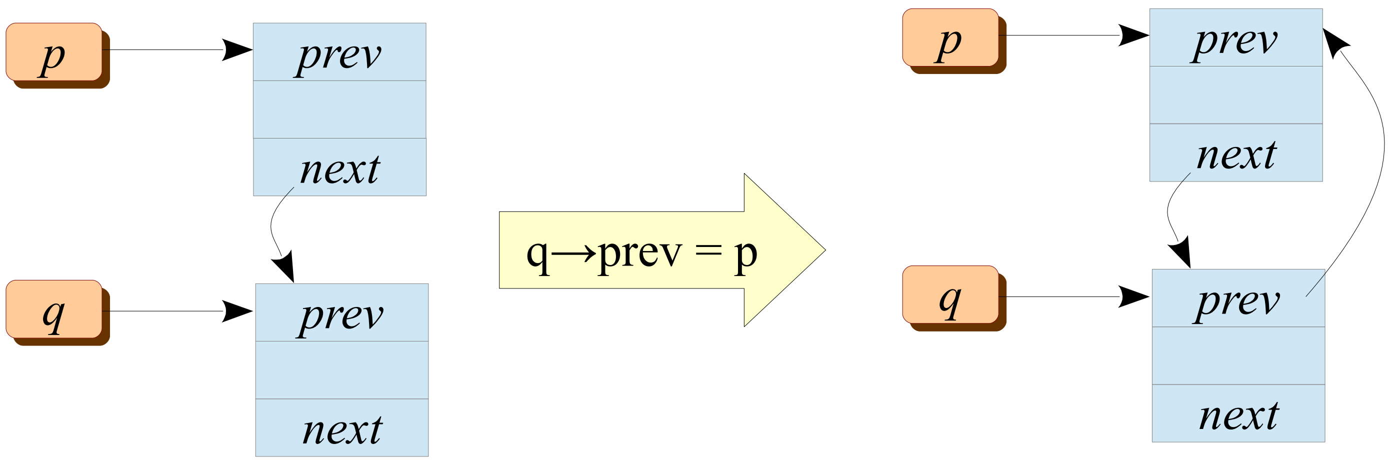
Interference

	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



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**

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

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

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

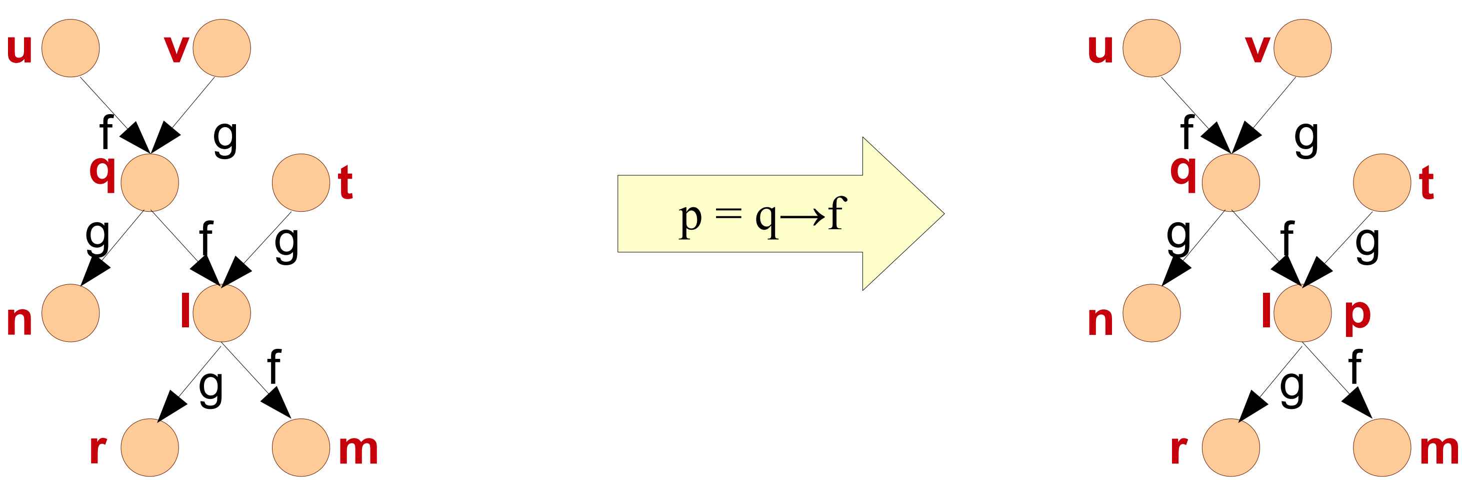
The implementation should create new D/I matrices from their current copies. In-situ update would lead to unsound or imprecise analysis.

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] | 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

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]}
p->f = q p->f = null	I_gen = {I[p][s] | I[q][s] and s ≠ p} U {I[p][p] | I[q][q]}
	p.shape = q.shape



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

<table>
<tr>
<td>p = malloc(...)

p = q
p = q→f
p = &(q→f)
p = q op k
p = null

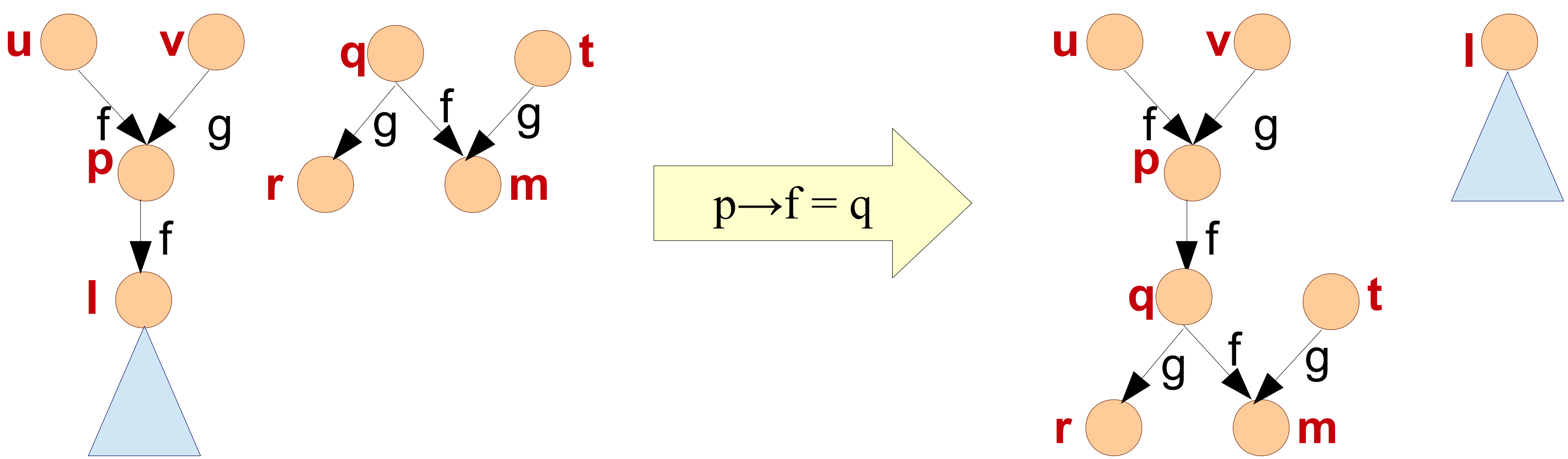
p->f = q
p->f = null</td>
<td>D_kill = { }, I_kill = { }

D_gen = {D[r][s] | D[r][p] and D[q][s]}
I_gen = {I[r][s] | D[r][p] and I[q][s]}

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)</td>
</tr>
</table>

Inference Rules

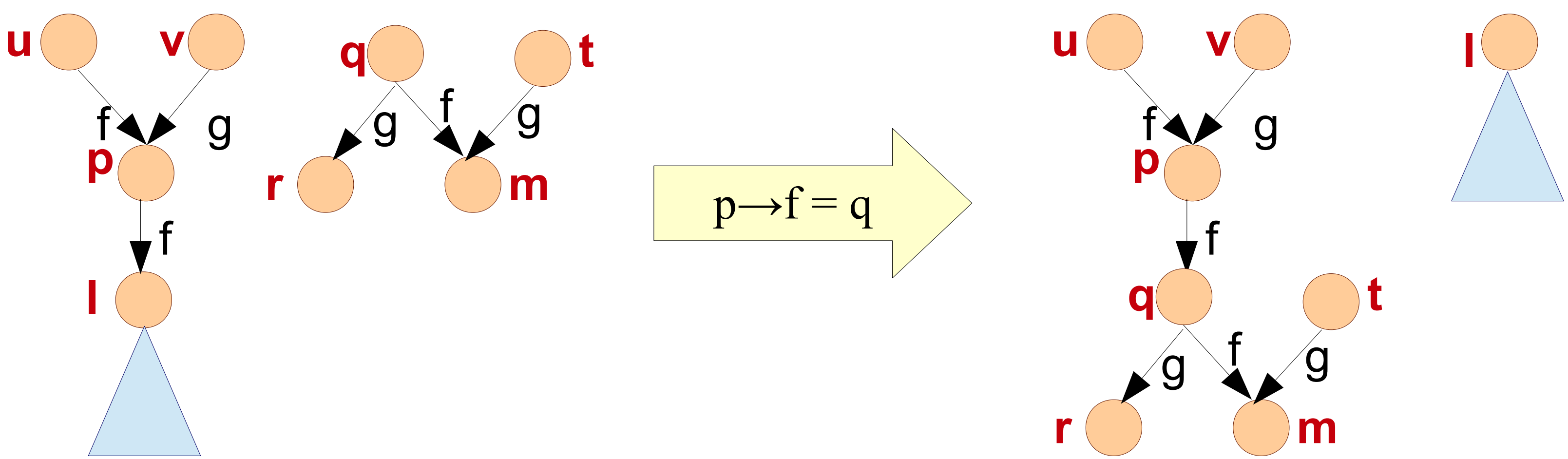
p = malloc(...)	D_kill = { }, I_kill = { }
p = q p = q→f p = &(q→f) p = q op k p = null	D_gen = {D[r][s] | D[r][p] and D[q][s]} I_gen = {I[r][s] | I[r][p] and I[q][s]}
p->f = q p->f = null	D[q][p] and D[s][q] ⇒ s.shape = Cycle D[q][p] and D[s][p] ⇒ s.shape = Cycle !D[q][p] and D[s][p] and I[s][q] and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

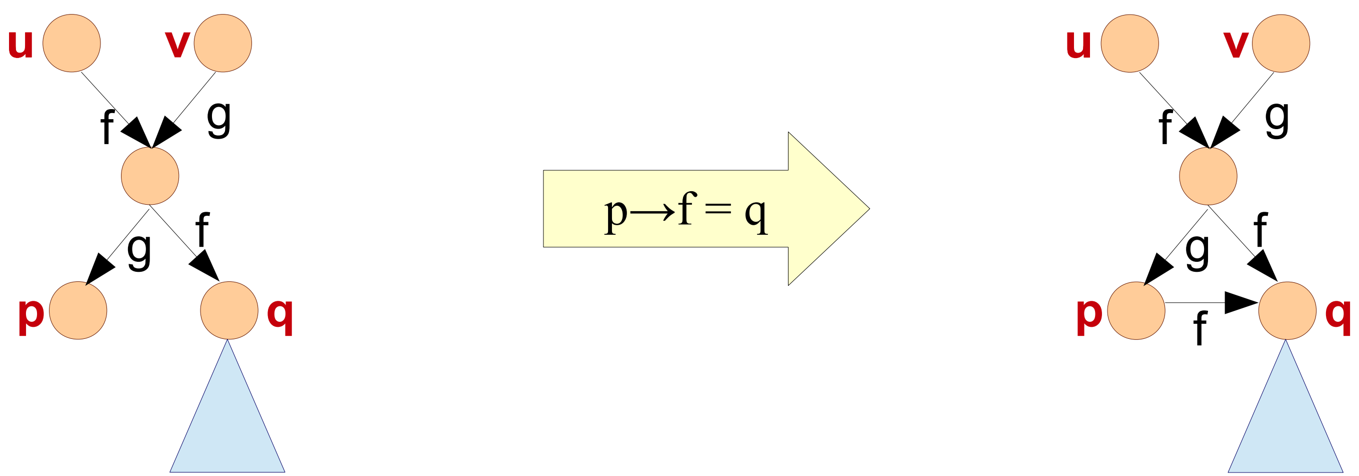
p = malloc(...)	D_kill = { }, I_kill = { }
p = q p = q→f p = &(q→f) p = q op k p = null	
p->f = q p->f = null	D_gen = {D[r][s] | D[r][p] and D[q][s]} I_gen = {I[r][s] | **D**[r][p] and I[q][s]} D[q][p] and D[s][q] ⇒ s.shape = Cycle D[q][p] and D[s][p] ⇒ s.shape = Cycle !D[q][p] and D[s][p] and I[s][q]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



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

Example

```
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;
}
```

Interference

	x	*y*	*t*
x		0	0
y			0
t			

Direction

	x	*y*	*t*
x	1	0	0
y	0	0	0
t	0	0	0

Shape

x	tree
y	tree
t	tree

Example

```
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;
}
```

Interference

	x	y	t
x		0	0
y			0
t			

Direction

	x	y	t
x	1	0	0
y	0	0	0
t	0	0	0

Shape

x	tree
y	tree
t	tree

Example

```
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;
}
```

Interference

	x	y	t
x		0	0
y			0
t			

Direction

	x	y	t
x	1	0	0
y	0	0	0
t	0	0	0

Shape

x	tree
y	tree
t	tree

Example

```
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;
}
```

Interference

	x	y	t
x		1	0
y			0
t			

Direction

	x	y	t
x	1	1	0
y	1	1	0
t	0	0	0

Shape

x	tree
y	tree
t	tree

Example

```
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 need to assume a finite representation for the data structure.

Interference

	x	*y*	*t*
x		1	0
y			0
t			

Direction

	x	*y*	*t*
x	1	0	0
y	1	1	0
t	0	0	0

Shape

x	tree
y	tree
t	tree

Example

```
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;
}
```

Since we do not model fields, we can't say that x is unreachable from y.

Interference

	x	y	t
x		1	0
y			0
t			

Direction

	x	y	t
x	1	0	0
y	1	1	0
t	0	0	0

Shape

x	tree
y	tree
t	tree

Example

```
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;
}
```

Interference

	x	y	t
x		1	1
y			1
t			

Direction

	x	y	t
x	1	0	0
y	1	1	1
t	1	1	1

Shape

x	tree
y	tree
t	tree

Example

```
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;
}
```

Interference

	x	y	t
x		1	1
y			1
t			

Direction

	x	y	t
x	1	1	0
y	1	1	0
t	1	1	1

Shape

x	tree
y	tree
t	tree

Example

```
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;
}
```

Interference

	x	*y*	*t*
x		1	1
y			1
t			

Direction

	x	*y*	*t*
x	1	0	0
y	1	1	0
t	1	1	1

Shape

x	tree
y	tree
t	tree

Example

```
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;
}
```

Interference

	x	y	t
x		1	1
y			1
t			

Direction

	x	y	t
x	1	0	0
y	1	1	1
t	1	1	1

Shape

x	tree
y	cycle
t	cycle

Example

```
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;
}
```

Interference

	x	y	t
x		1	1
y			1
t			

Direction

	x	y	t
x	1	0	0
y	1	1	1
t	1	1	1

Shape

x	tree
y	cycle
t	cycle

Example

```
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;
}
```

Interference

	x	y	t
x		1	1
y			1
t			

Direction

	x	y	t
x	1	1	0
y	1	1	0
t	1	1	1

Shape

x	tree
y	cycle
t	cycle

Example

```
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;
}
```

Interference

	x	*y*	*t*
x		1	1
y			1
t			

Direction

	x	*y*	*t*
x	1	0	0
y	1	1	0
t	1	1	1

Shape

x	tree
y	cycle
t	cycle

Example

```
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;
}
```

Interference

	x	*y*	*t*
x		1	1
y			1
t			

Direction

	x	*y*	*t*
x	1	0	0
y	1	1	1
t	1	1	1

Shape

x	tree
y	cycle
t	cycle

Example

```
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;
}
```

Interference

	x	*y*	*t*
x		1	1
y			1
t			

Direction

	x	*y*	*t*
x	1	1	1
y	1	1	1
t	1	1	1

Shape

x	cycle
y	cycle
t	cycle

Classwork

```
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.

- Check “Identifying Dynamic Data Structures by Learning Evolving Patterns in Memory” from TACAS 2013.
- http://dl.acm.org/citation.cfm?doid=2483760.24837
- http://dl.acm.org/citation.cfm?id=1760303&CFID=3
- http://dl.acm.org/citation.cfm?id=271517&picked=fo
- http://dl.acm.org/citation.cfm?id=1040331&CFID=3
- http://dl.acm.org/citation.cfm?id=1480917&CFID=3
- http://dl.acm.org/citation.cfm?id=1855759&CFID=3
- http://dl.acm.org/citation.cfm?id=1081721&CFID=3