# Automatic Numeric Abstractions for Heap-Manipulating Programs

STEPHEN MAGILL
CARNEGIE MELLON UNIVERSITY

JOINT WITH
JOSH BERDINE, BYRON COOK, PETER LEE,
MING-HSIEN TSAI, YIH-KUEN TSAY

# Abstractions

Numeric Operations (instrumentation analysis) — *length of list at p increases by 1*

Data Structure Operations (shape analysis) — *add node to front of list at p*

Concrete Pointer Operations (source code)

```
t = malloc(sizeof(ListNode));
t->next = p;
p = t;
```

# Abstractions

| | |
|---|---|
| Numeric Operations<br>(instrumentation analysis) | *length of list at p increases by 1*<br>**`k = k + 1;`** |
| Data Structure Operations<br>(shape analysis) | *add node to front of list at p* ***with length k*** |
| Concrete Pointer Operations<br>(source code) | `t = malloc(sizeof(ListNode));`<br>`t->next = p;`<br>`p = t;` |

# Uses

- Termination Proving
- Bounding Time / Space Usage
- Safety Properties (including Memory Safety)

# Example

```
while(x ≠ 0) {
  x = x->next;
}
```

# Example

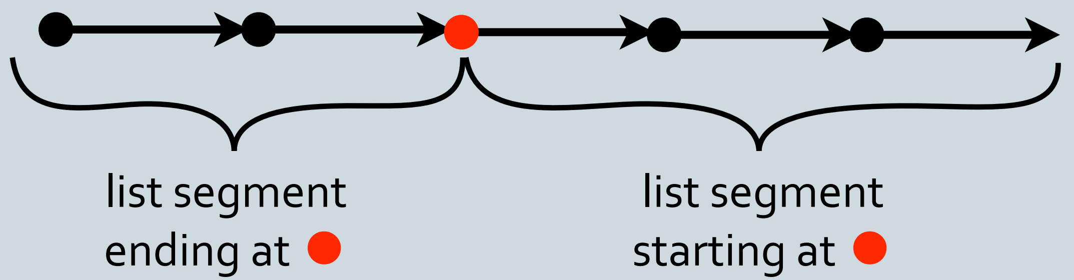
```
while(x ≠ 0) {
  x = x->next;
}
```

# Example

```
while(x ≠ 0) {
  x = x->next;
}
```

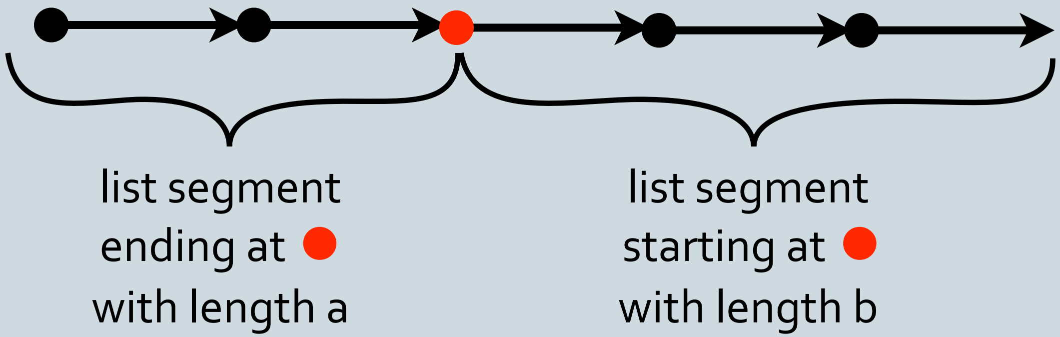
# Example

```
while(x ≠ 0) {
  x = x->next;
}
```



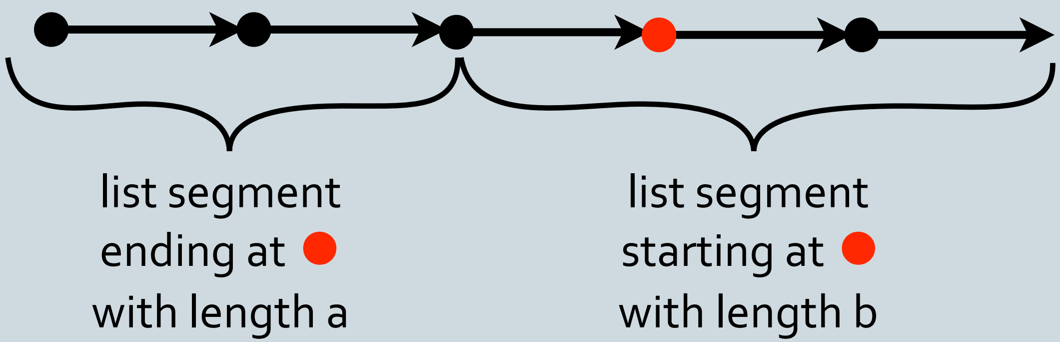
# Example

```
while(x ≠ 0) {
  x = x->next;
}
```



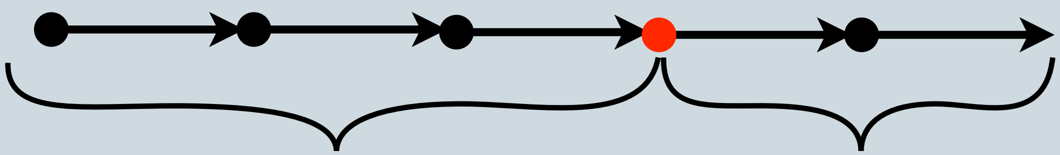
# Example

```
while(x ≠ 0) {
  x = x->next;
}
```



# Example

```
while(x ≠ 0) {
  x = x->next;
}
```



list segment ending at ● with length a

$a = a + 1$

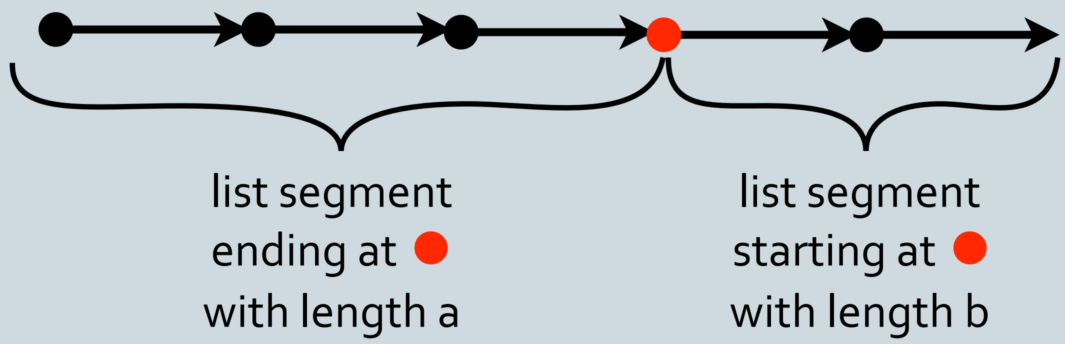
list segment starting at ● with length b

$b = b - 1$

$b \geq 0$

# Example

```
while(x ≠ 0) {
  x = x->next;
}
```




```
while(b ≥ 0) {
  a = a + 1;
  b = b - 1;
}
```

# Example

```
while(x ≠ 0) {
  x = x->next;
}
```

$\exists x'.\ ls(a; x', x) * ls(b; x, 0)$

```
while(b ≥ 0) {
  a = a + 1;
  b = b - 1;
}
```

# Example

```
x = x->next;
```

$$\exists x'.\ ls(a; x', x) * ls(b; x, 0)$$

$$\exists x'.\ ls(a; x', x) * ls(b; x, 0)$$

```
a = a + 1;
b = b - 1;
```

# Example

$$\exists x'.\ ls(a;x',x) * ls(b;x,0)$$

| | |
|---|---|
| `x = x->next;` | `a = a + 1;`<br>`b = b - 1;` |

$$\exists x'.\ ls(a;x',x) * ls(b;x,0)$$

| | |
|---|---|
| `x = x->next;` | `a = a + 1;`<br>`b = b - 1;` |

$$\exists x'.\ ls(a;x',x) * ls(b;x,0)$$

# Example

```
x = x->next;
```

$\exists x'.\ ls(a; x', x) * ls(b; x, 0)$

```
a = a + 1;
b = b - 1;
```

$\exists x'.\ ls(a; x', x) * ls(b; x, 0)$

```
x = x->next;
```

```
a = a + 1;
b = b - 1;
```

$\exists x'.\ ls(a; x', x) * ls(b; x, 0)$

…

…

$\exists x'.\ ls(a; x', x) * ls(b; x, 0)$

# Example

$$\exists a, b, x'.\ ls(a; x', x) * ls(b; x, 0)$$

```
x = x->next;
```

$$\exists a, b, x'.\ ls(a; x', x) * ls(b; x, 0)$$

```
x = x->next;
```

$$\exists a, b, x'.\ ls(a; x', x) * ls(b; x, 0)$$

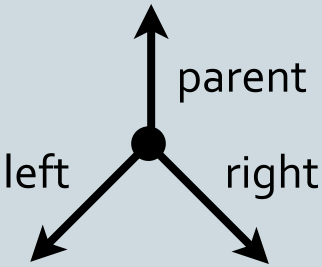
…

$$\exists a, b, x'.\ ls(a; x', x) * ls(b; x, 0)$$

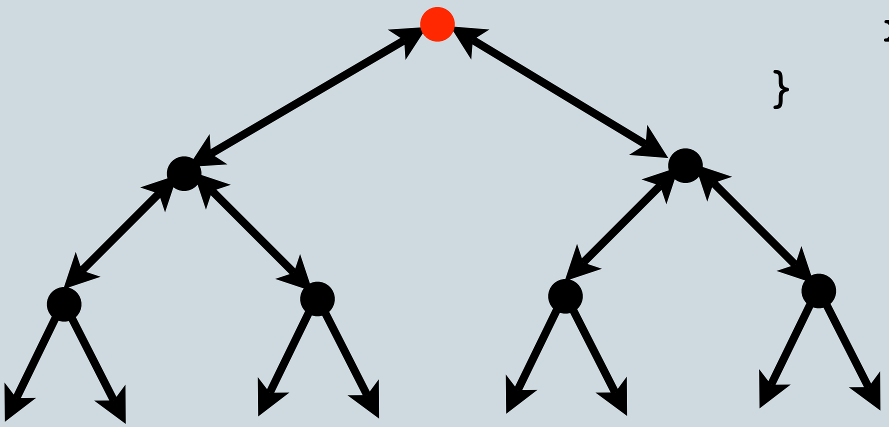
# Example

| | | |
|---|---|---|
| | $\exists x'.\ ls(a;x',x) * ls(b;x,0)$ | |
| `x = x->next;` | | `a = a + 1;`<br>`b = b - 1;` |
| | $\exists x'.\ ls(a;x',x) * ls(b;x,0)$ | |
| `x = x->next;` | | `a = a + 1;`<br>`b = b - 1;` |
| | $\exists x'.\ ls(a;x',x) * ls(b;x,0)$ | |
| … | | … |
| | $\exists x'.\ ls(a;x',x) * ls(b;x,0)$ | |

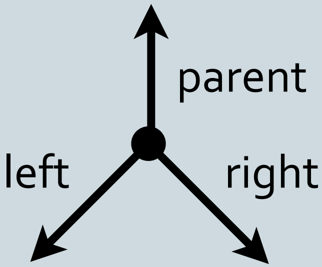
# Example



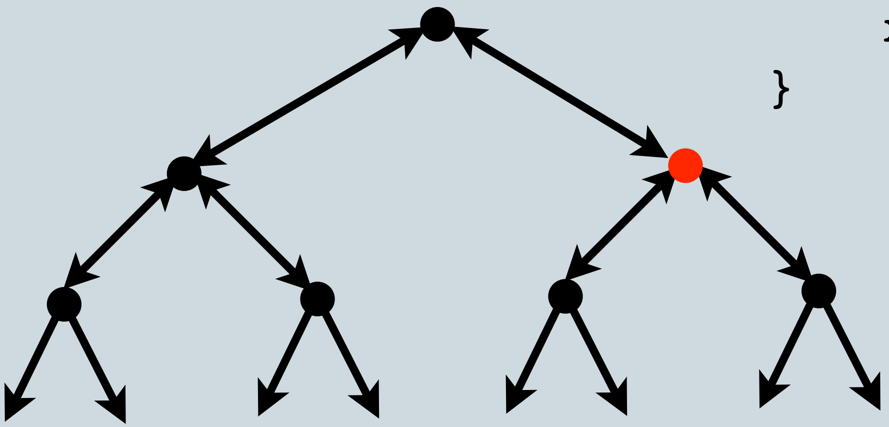

```
while(x ≠ 0) {
  if(-)
    x = x->right;
  else
    x = x->left;
}
```



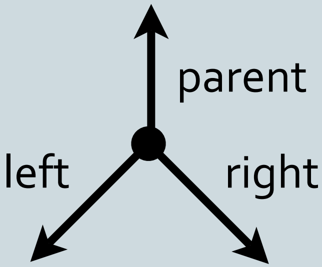
# Example



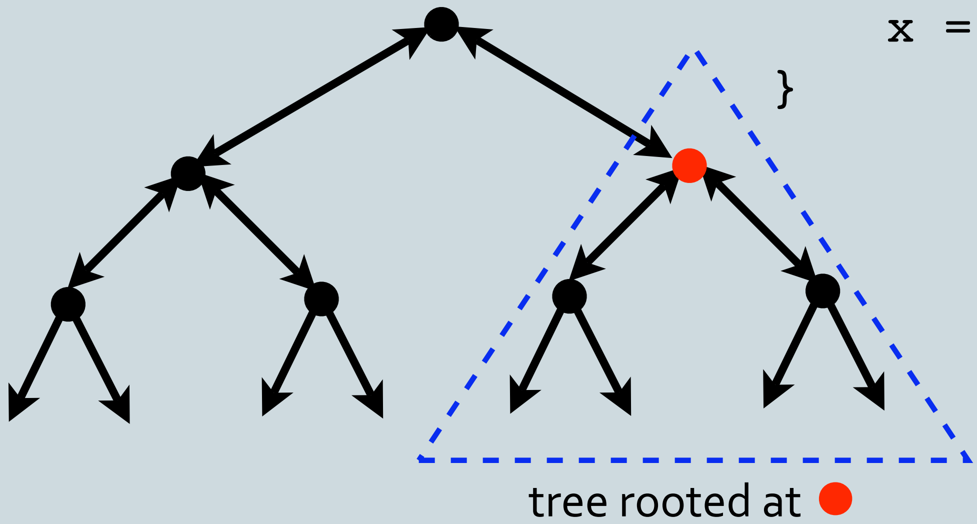

```
while(x ≠ 0) {
  if(-)
    x = x->right;
  else
    x = x->left;
}
```



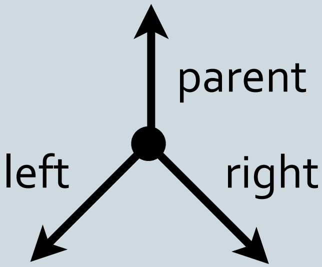
# Example



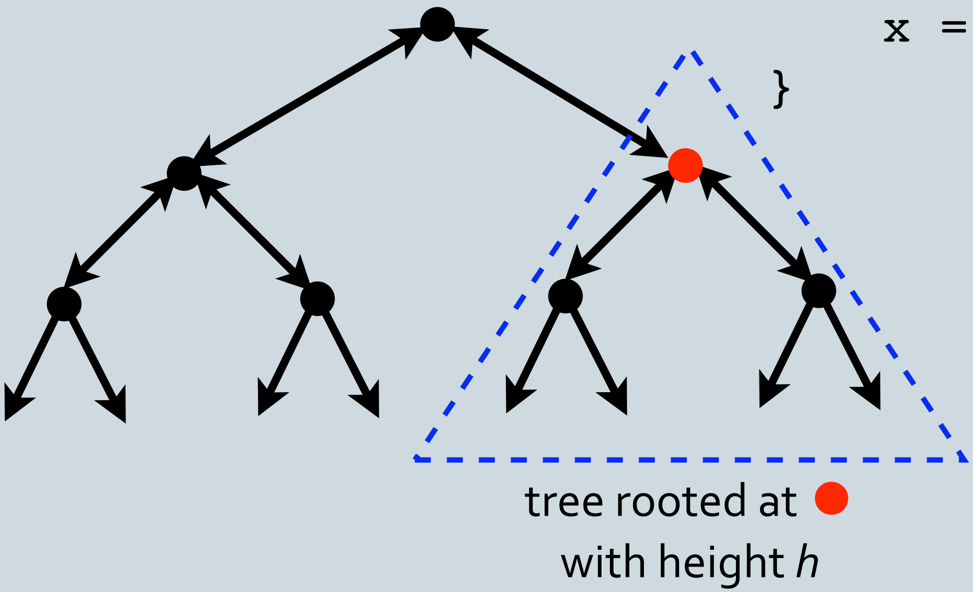

```
while(x ≠ 0) {
  if(-)
    x = x->right;
  else
    x = x->left;
}
```



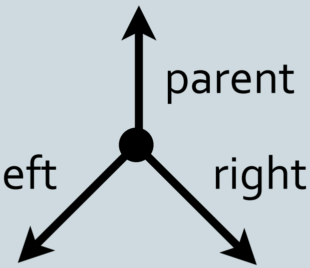
# Example



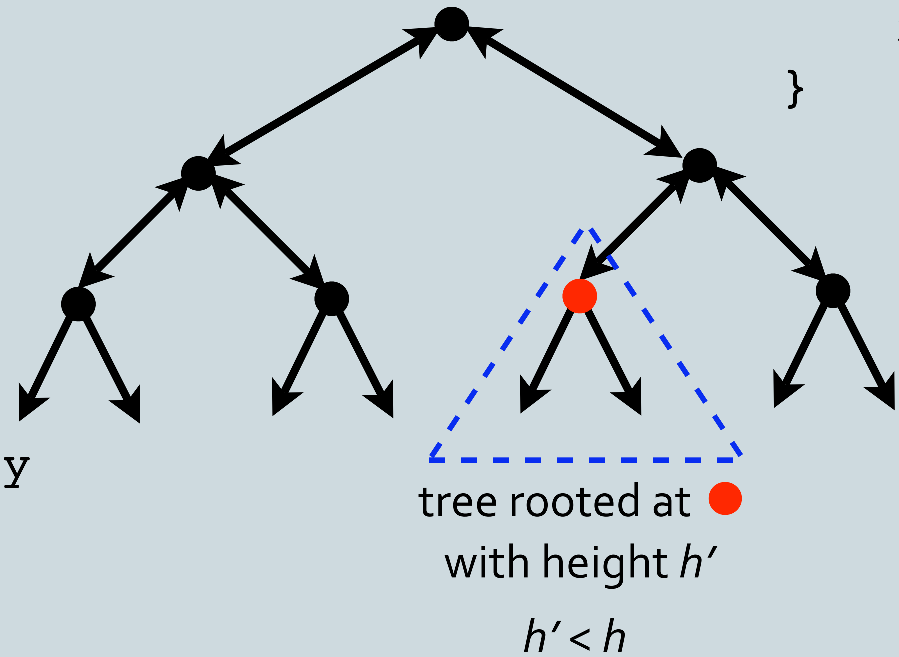

```
while(x ≠ 0) {
  if(-)
    x = x->right;
  else
    x = x->left;
}
```



# Example




```
while(x ≠ 0) {
  if(-)
    x = x->right;
  else
    x = x->left;
}
```




```
while(h > 0) {
  let h’ satisfy
    h’ < h
  in
    h = h’;
}
```

# Small Example

```
while( k > 0 ) {
   k = k - 1;
}
```

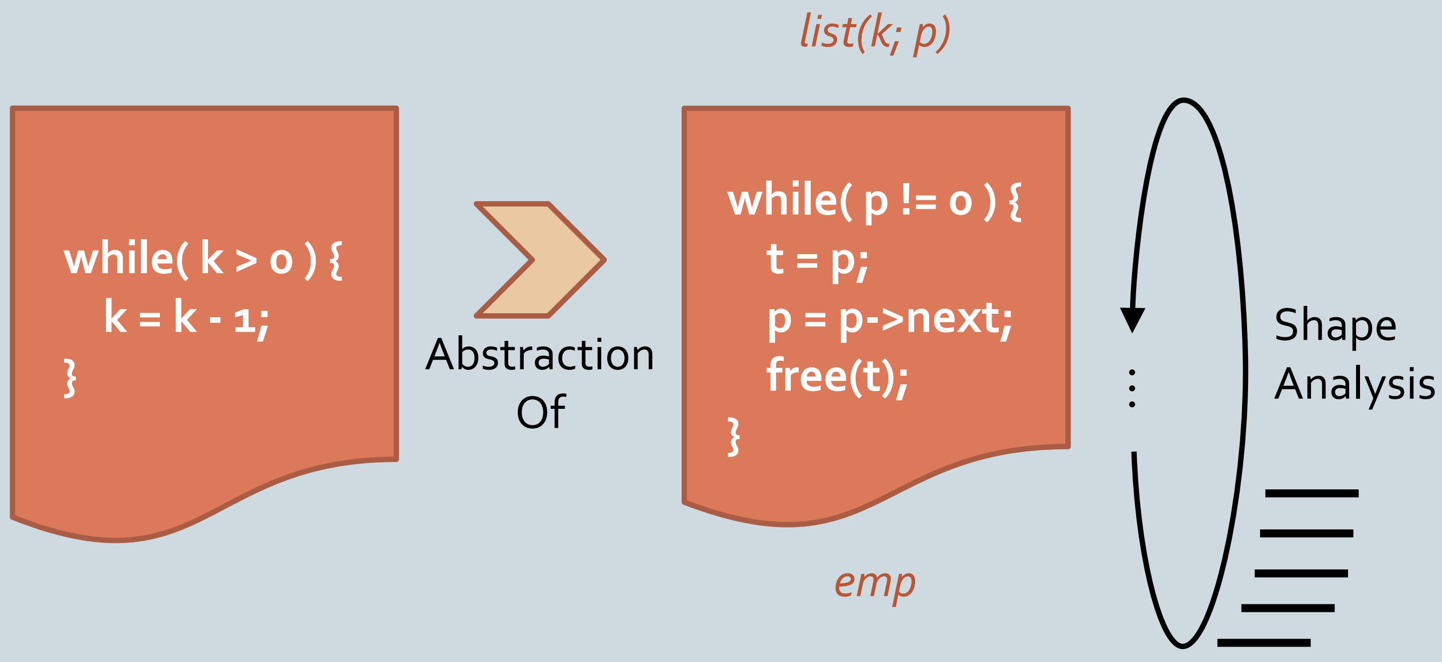
Abstraction Of

*list(k; p)*

```
while( p != 0 ) {
   t = p;
   p = p->next;
   free(t);
}
```

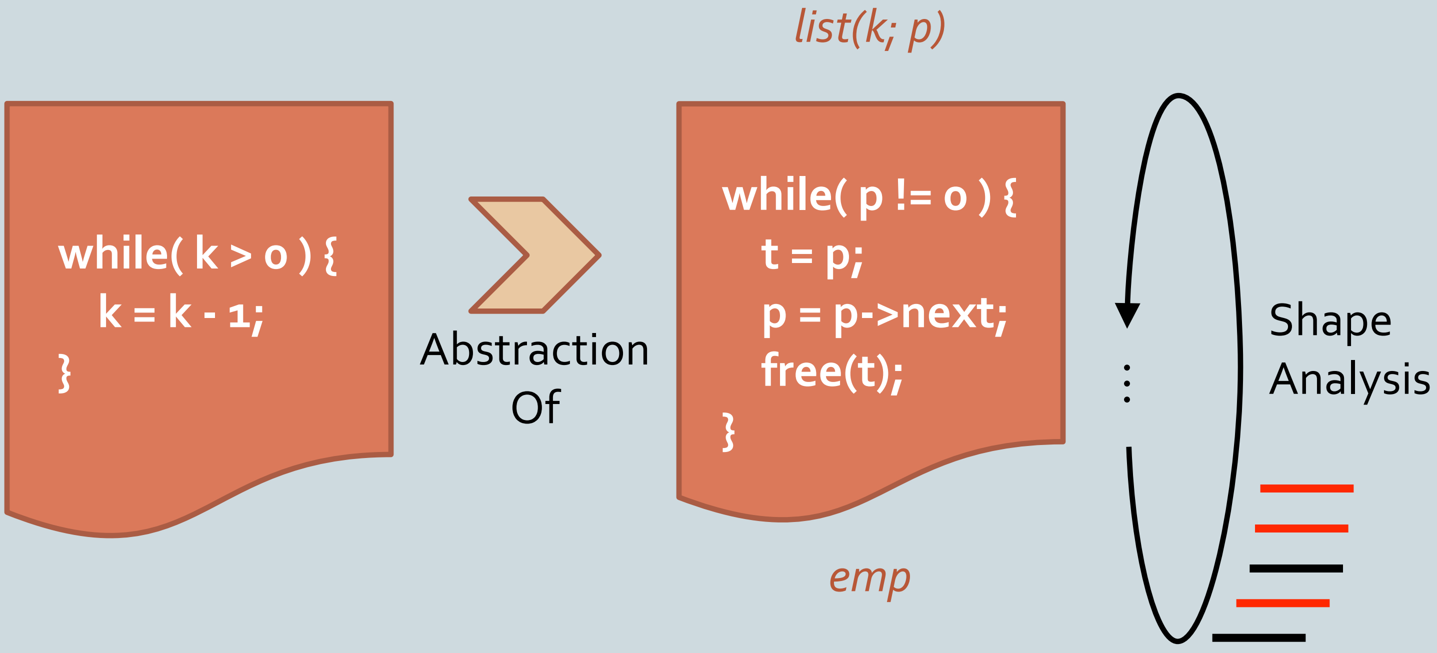
*emp*

# How?

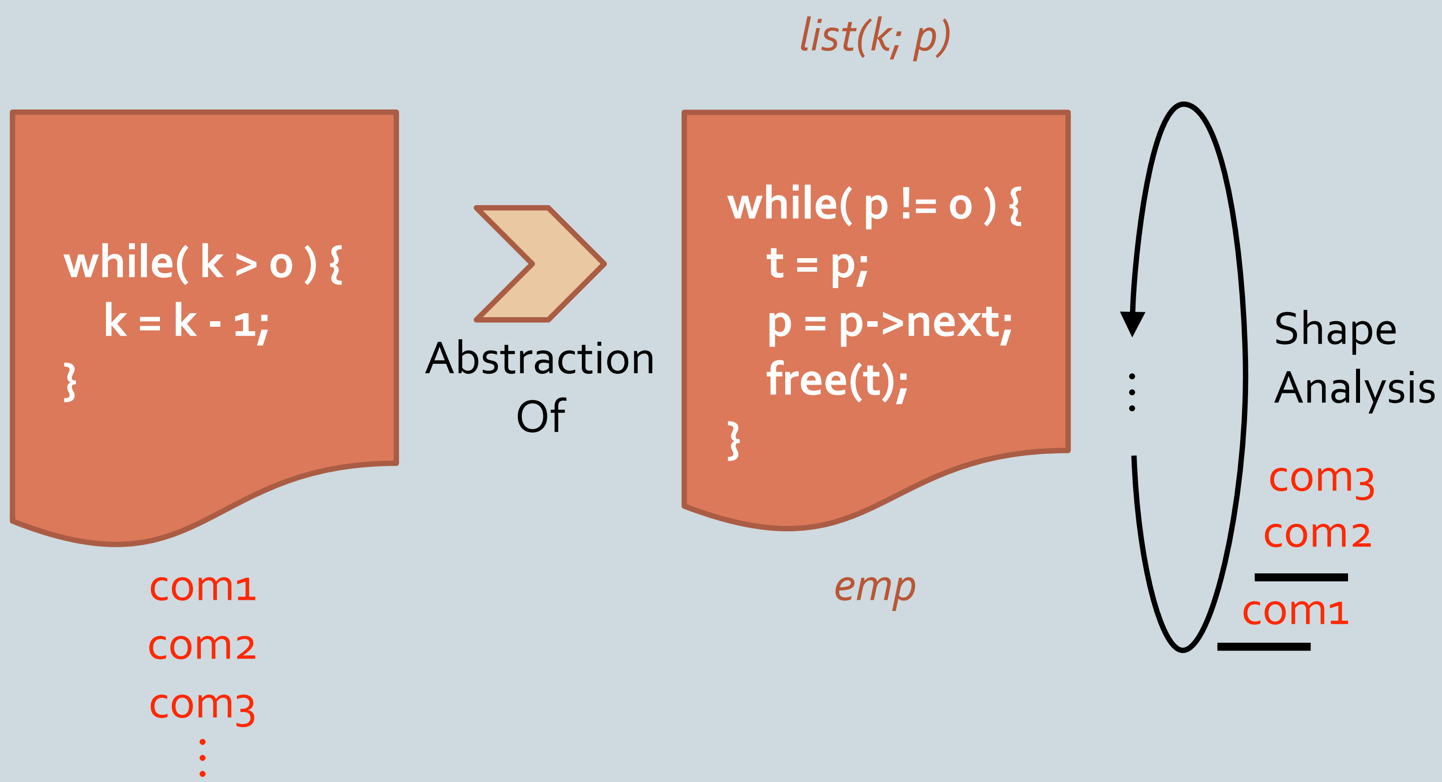


while( k > 0 ) {
k = k - 1;
}
Abstraction
Of
list(k; p)
while( p != 0 ) {
t = p;
p = p->next;
free(t);
}
emp
Shape
Analysis

# How?

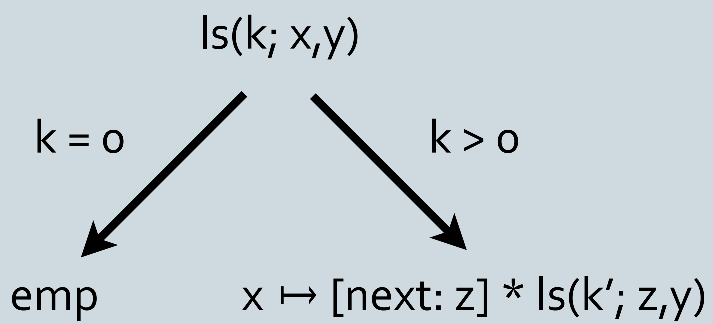

while( k > 0 ) {
k = k - 1;
}
Abstraction
Of
list(k; p)
while( p != 0 ) {
t = p;
p = p->next;
free(t);
}
emp
Shape
Analysis

# How?


while( k > 0 ) {
k = k - 1;
}
com1
com2
com3
Abstraction Of
list(k; p)
while( p != 0 ) {
t = p;
p = p->next;
free(t);
}
emp
Shape Analysis
com3
com2
com1

# Inductive Definitions

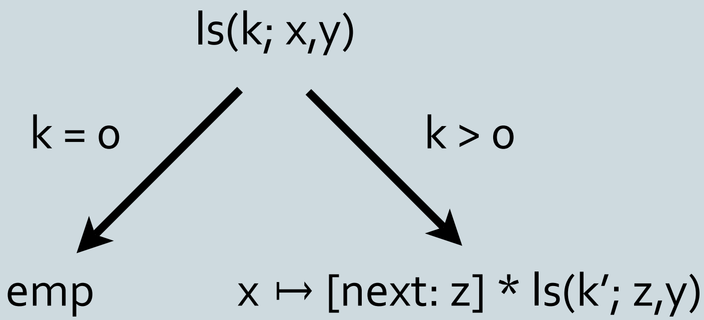
$$
\begin{aligned}
&ls(\underline{k};\, \mathit{first}, \mathit{next}) \equiv \\
&\qquad (\underline{k} = 0 \wedge \mathbf{emp} \wedge \mathit{first} = \mathit{next}) \\
&\quad \vee \ (\underline{k} > 0 \wedge \exists \underline{k}'.\ \underline{k} = \underline{k}' + 1 \wedge \\
&\qquad\qquad\qquad \exists z.\ (\mathit{first} \mapsto [\mathrm{next} : z]) * ls(\underline{k}'; z, \mathit{next}))
\end{aligned}
$$




`k' = k - 1`

# Inductive Definitions

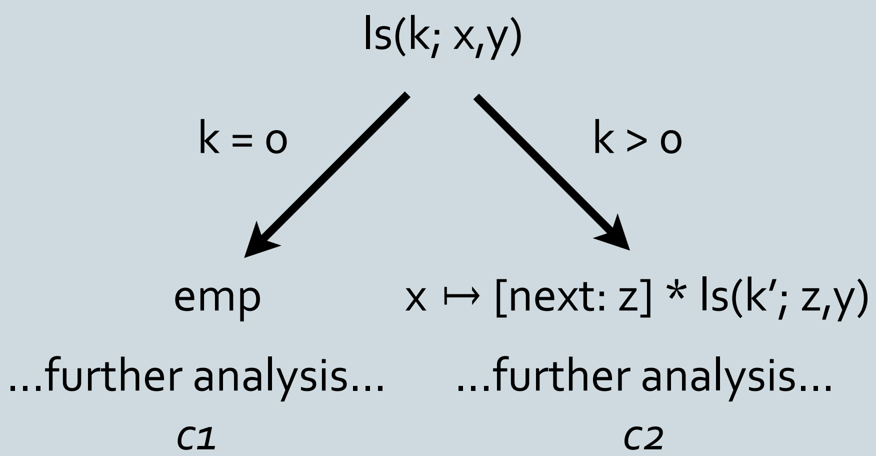
$$
\begin{aligned}
&ls(\underline{k};\, \mathit{first}, \mathit{next}) \equiv \\
&\qquad (\underline{k} = 0 \wedge \mathbf{emp} \wedge \mathit{first} = \mathit{next}) \\
&\quad \vee \quad (\underline{k} > 0 \wedge \exists \underline{k'}.\ \underline{k} = \underline{k'} + 1 \wedge \\
&\qquad\qquad\qquad \exists z.\ (\mathit{first} \mapsto [\mathrm{next} : z]) * ls(\underline{k'};\, z, \mathit{next}))
\end{aligned}
$$




```
k' = k - 1
```

# Inductive Definitions

$$\begin{aligned} &ls(\underline{k};\, \mathit{first}, \mathit{next}) \equiv \\ &\quad (\underline{k} = 0 \wedge \mathbf{emp} \wedge \mathit{first} = \mathit{next}) \\ &\vee \quad (\underline{k} > 0 \wedge \exists \underline{k}'.\ \underline{k} = \underline{k}' + 1 \wedge \\ &\qquad\qquad \exists z.\ (\mathit{first} \mapsto [\mathrm{next}: z]) * ls(\underline{k}';\, z, \mathit{next})) \end{aligned}$$




```
if(k==0)
  c1
else
  k' = k - 1;
  c2;
```

# Inductive Definitions

$$ls(\underline{k};\, \mathit{first}, \mathit{next}) \equiv$$
$$(\underline{k} = 0 \wedge \mathbf{emp} \wedge \mathit{first} = \mathit{next})$$
$$\vee \quad (\underline{k} > 0 \wedge \exists \underline{k}'.\ \underline{k} = \underline{k}' + 1 \wedge$$
$$\exists z.\ (\mathit{first} \mapsto [\mathrm{next} : z]) * ls(\underline{k}';\, z, \mathit{next}))$$

```
k' = k - 1;
```

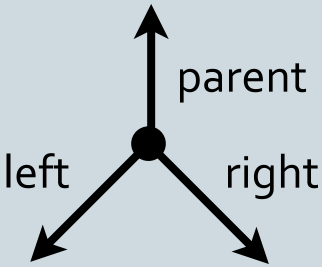
*or*

```
k' = non_det();
assume(k = k' + 1);
```

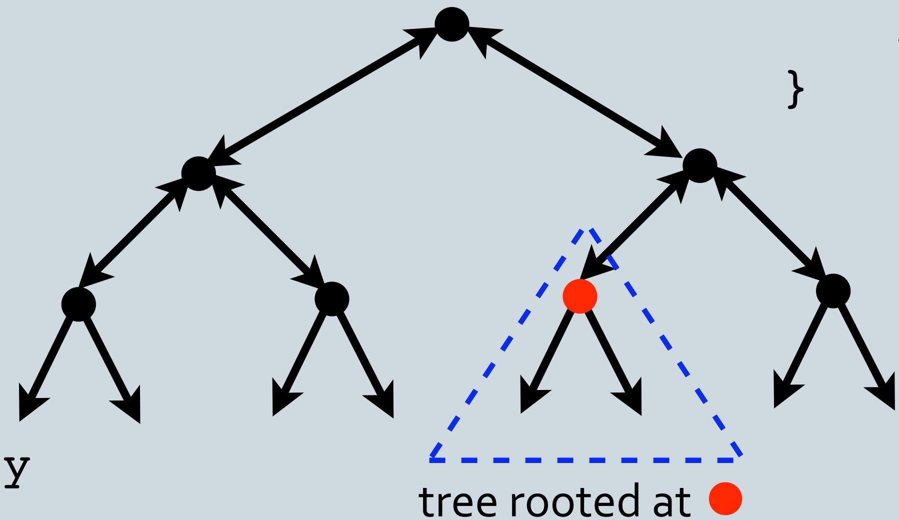
*or*

```
k' = non_det();
if(¬(k = k' + 1))
  return;
...
```

# Example

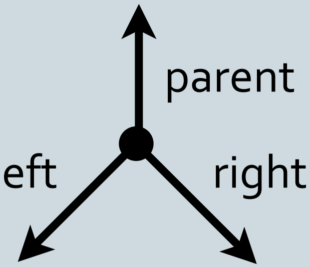



```
while(x ≠ 0) {
  if(-)
    x = x->right;
  else
    x = x->left;
}
```

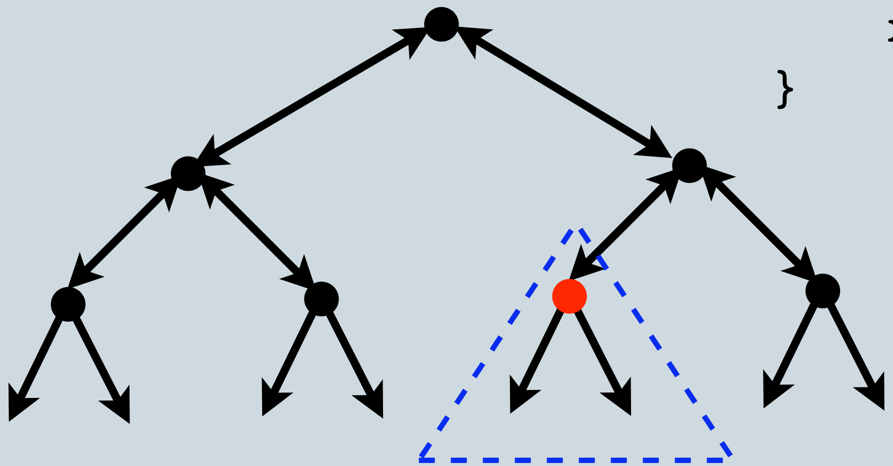



```
while(h > 0) {
  let h’ satisfy
    h’ < h
  in
    h = h’;
}
```

# Example




```
while(x ≠ 0) {
  if(-)
    x = x->right;
  else
    x = x->left;
}
```



tree rooted at ● with height *h*

$h' < h$

```
while(h > 0) {
  h' = nondet();
  assume(h' < h);
  h = h';
}
```

# Inductive Definitions

$$
\begin{aligned}
ls(\underline{k};\ \mathit{first}, \mathit{next}) \equiv\ & \\
& (\underline{k} = 0 \wedge \mathbf{emp} \wedge \mathit{first} = \mathit{next}) \\
\vee\ & (\underline{k} > 0 \wedge \exists \underline{k}'.\ \underline{k} = \underline{k}' + 1 \wedge \\
& \qquad \exists z.\ (\mathit{first} \mapsto [\text{next} : z]) * ls(\underline{k}';\ z, \mathit{next}))
\end{aligned}
$$

x ↦ [next: z] * ls(k; z,y) —abstract→ ls(k'; x,y)

x ↦ [next: z] * z ↦ [next: y] —abstract→ ls(k'; x,y)

# Inductive Definitions

$$
\begin{aligned}
&ls(\underline{k};\, \mathit{first}, \mathit{next}) \equiv \\
&\qquad (\underline{k} = 0 \wedge \mathbf{emp} \wedge \mathit{first} = \mathit{next}) \\
&\quad \vee \quad (\underline{k} > 0 \wedge \exists \underline{k}'.\ \underline{k} = \underline{k}' + 1 \wedge \\
&\qquad\qquad\qquad \exists z.\ (\mathit{first} \mapsto [\mathrm{next} : z]) * ls(\underline{k}';\, z, \mathit{next}))
\end{aligned}
$$

Reverse relation: k′ = k + 1

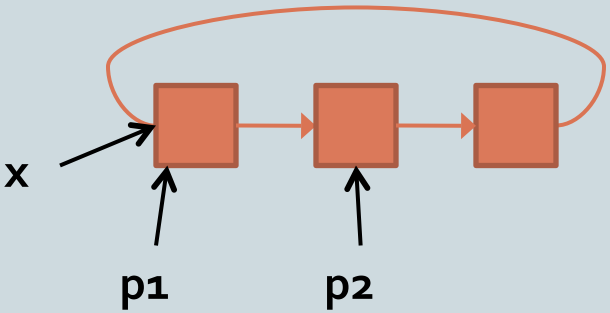
| | | | Numeric Command |
|---|---|---|---|
| x ↦ [next: z] * ls(k; z,y) | abstract → | ls(k′; x,y) | `k' = k + 1` |
| x ↦ [next: z] * z ↦ [next: y] | abstract → | ls(k′; x,y) | `k' = 2` |

# Example – An odd test for cyclicity

```
int has_cycle(Listp x) {


  Listp p1 = x;
  Listp p2 = x;

  p2 = p2->next;
  while (p1 != p2) {
    if (p2 == 0 || p2->next == 0)
      return 0;
    p2 = p2->next;
    p2 = p2->next;
    p1 = p1->next;
  }
  return 1;
}
```

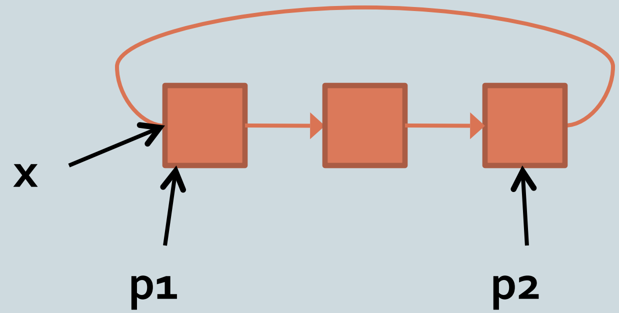


# Example – An odd test for cyclicity

```
int has_cycle(Listp x) {


  Listp p1 = x;
  Listp p2 = x;

  p2 = p2->next;
  while (p1 != p2) {
    if (p2 == 0 || p2->next == 0)
      return 0;
    p2 = p2->next;
    p2 = p2->next;
    p1 = p1->next;
  }
  return 1;
}
```

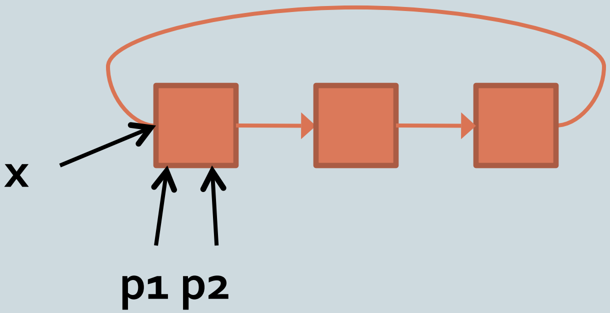


# Example – An odd test for cyclicity

```
int has_cycle(Listp x) {


  Listp p1 = x;
  Listp p2 = x;

  p2 = p2->next;
  while (p1 != p2) {
    if (p2 == 0 || p2->next == 0)
      return 0;
    p2 = p2->next;
    p2 = p2->next;
    p1 = p1->next;
  }
  return 1;
}
```

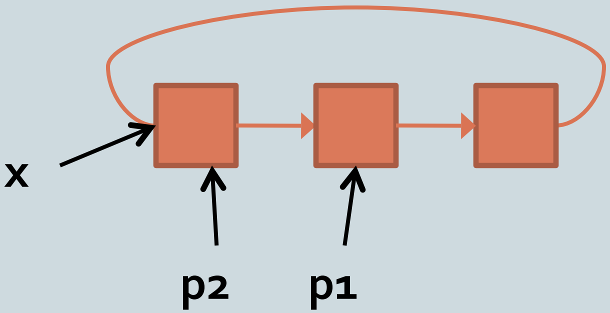


# Example – An odd test for cyclicity

```
int has_cycle(Listp x) {


  Listp p1 = x;
  Listp p2 = x;

  p2 = p2->next;
  while (p1 != p2) {
    if (p2 == 0 || p2->next == 0)
      return 0;
    p2 = p2->next;
    p2 = p2->next;
    p1 = p1->next;
  }
  return 1;
}
```

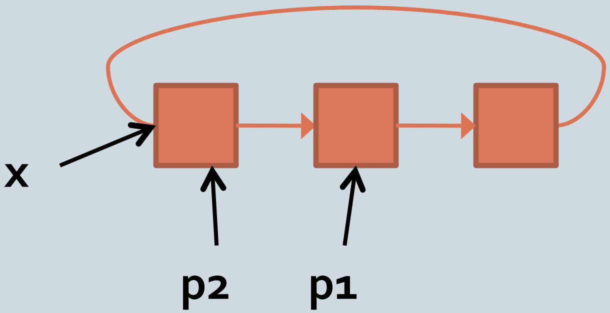


# Example – An odd test for cyclicity

```
int has_cycle(Listp x) {


  Listp p1 = x;
  Listp p2 = x;

  p2 = p2->next;
  while (p1 != p2) {
    if (p2 == 0 || p2->next == 0)
      return 0;
    p2 = p2->next;
    p2 = p2->next;
    p1 = p1->next;
  }
  return 1;
}
```

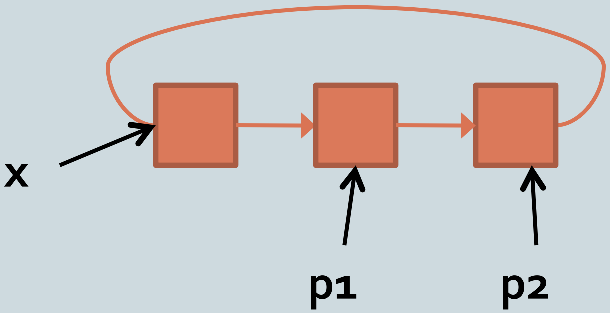


# Example – An odd test for cyclicity

```
int has_cycle(Listp x) {


  Listp p1 = x;
  Listp p2 = x;

  p2 = p2->next;
  while (p1 != p2) {
    if (p2 == 0 || p2->next == 0)
      return 0;
    p2 = p2->next;
    p2 = p2->next;
    p1 = p1->next;
  }
  return 1;
}
```

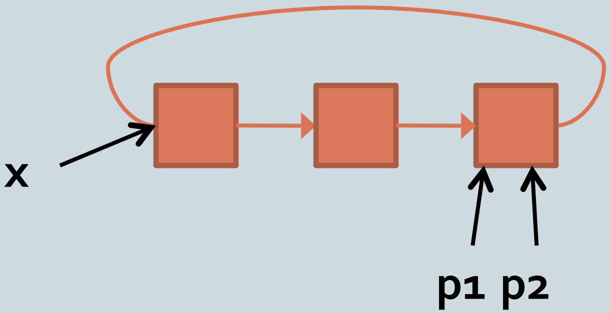


# Example – An odd test for cyclicity

```
int has_cycle(Listp x) {


  Listp p1 = x;
  Listp p2 = x;

  p2 = p2->next;
  while (p1 != p2) {
    if (p2 == 0 || p2->next == 0)
      return 0;
    p2 = p2->next;
    p2 = p2->next;
    p1 = p1->next;
  }
  return 1;
}
```

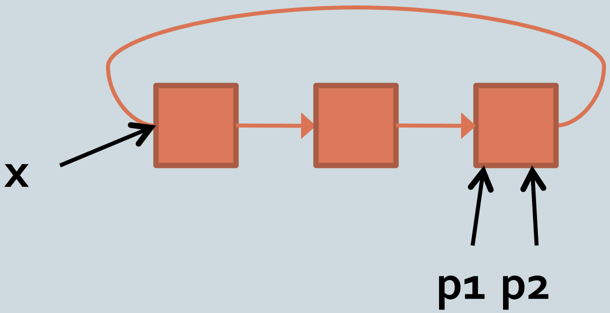


# Example – An odd test for cyclicity

```
int has_cycle(Listp x) {


  Listp p1 = x;
  Listp p2 = x;

  p2 = p2->next;
  while (p1 != p2) {
    if (p2 == 0 || p2->next == 0)
      return 0;
    p2 = p2->next;
    p2 = p2->next;
    p1 = p1->next;
  }
  return 1;
}
```

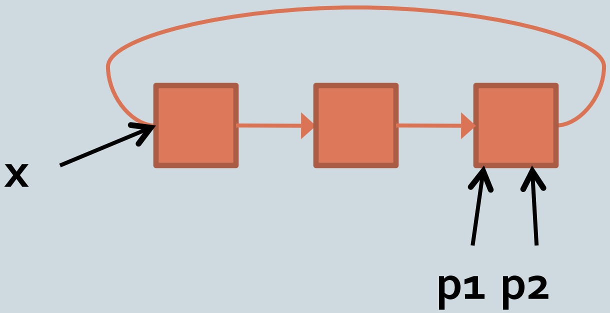


# Example – An odd test for cyclicity

```
int has_cycle(Listp x) {


  Listp p1 = x;
  Listp p2 = x;

  p2 = p2->next;
  while (p1 != p2) {
    if (p2 == 0 || p2->next == 0)
      return 0;
    p2 = p2->next;
    p2 = p2->next;
    p1 = p1->next;
  }
  return 1;
}
```

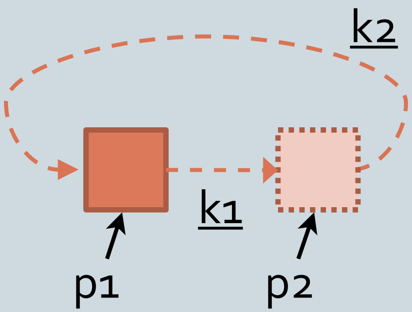


# Example – An odd test for cyclicity

```
int has_cycle(Listp x) {
  tassume("ls(k,x,x) & k > 0");

  Listp p1 = x;
  Listp p2 = x;

  p2 = p2->next;
  while (p1 != p2) {
    if (p2 == 0 || p2->next == 0)
      return 0;
    p2 = p2->next;
    p2 = p2->next;
    p1 = p1->next;
  }
  return 1;
}
```




$$ls(\underline{k1}; p1, p2) * ls(\underline{k2}; p2, p1)$$
$$\wedge\ \underline{k1} + \underline{k2} > 0$$

$$\begin{aligned} ls(\underline{k};\ \mathit{first}, \mathit{next}) \equiv\ & \\ & (\underline{k} = 0 \wedge \mathbf{emp} \wedge \mathit{first} = \mathit{next}) \\ \vee\ & (\underline{k} > 0 \wedge \exists \underline{k}'.\ \underline{k} = \underline{k}' + 1 \wedge \\ & \quad \exists z.\ (\mathit{first} \mapsto [\mathrm{next} : z]) * ls(\underline{k}'; z, \mathit{next})) \end{aligned}$$

# Example – An odd test for cyclicity

```
int has_cycle(Listp x) {
  tassume("ls(k,x,x) & k > 0");

  Listp p1 = x;
  Listp p2 = x;

  p2 = p2->next;
  while (p1 != p2) {
    if (p2 == 0 || p2->next == 0)
      return 0;
    p2 = p2->next;
    p2 = p2->next;
    p1 = p1->next;
  }
  return 1;
}
```

$$ls(\underline{k1}; p1, p2) * ls(\underline{k2}; p2, p1) \wedge \underline{k1} + \underline{k2} > 0$$

```
assume(k > 0);
k1 = 1;
k2 = k - 1;
while( k1 != 0 && k2 != 0 ) {
    if(k2 > 0)
        k2--; k1++;
    else
        k2 = k1 - 1;
        k1 = 1;
    ...
    if(k1 > 0)
        k1--; k2++;
    else
        k1 = k1 - 1;
        k2 = 1;
}
```

# Converting Branch Conditions

$$ls(\underline{k1}; p1, p2) * ls(\underline{k2}; p2, p1) \wedge \underline{k1} + \underline{k2} > 0 \quad \not\models \quad p1 \neq p2$$

Doesn't hold.

# Converting Branch Conditions

$$ls(\underline{k1}; p1, p2) * ls(\underline{k2}; p2, p1) \wedge \; \underline{k1} + \underline{k2} > 0 \quad \nvdash \quad p1 \neq p2$$

Doesn't hold.

But…

$$\begin{array}{c} ls(\underline{k1}; p1, p2) * ls(\underline{k2}; p2, p1) \\ \wedge \; \underline{k1} + \underline{k2} > 0 \wedge \boxed{(\underline{k1} \neq 0 \wedge \underline{k2} \neq 0)} \end{array} \quad \vdash \quad p1 \neq p2$$

# Converting Branch Conditions

$$ls(\underline{k1}; p1, p2) * ls(\underline{k2}; p2, p1) \wedge \underline{k1} + \underline{k2} > 0 \quad \nvdash \quad p1 \neq p2$$

Doesn't hold.

But…

$$\begin{array}{c} ls(\underline{k1}; p1, p2) * ls(\underline{k2}; p2, p1) \\ \wedge\ \underline{k1} + \underline{k2} > 0 \wedge \boxed{(\underline{k1} \neq 0 \wedge \underline{k2} \neq 0)} \end{array} \quad \vdash \quad p1 \neq p2$$

<u>Abduction</u>

# Example – An odd test for cyclicity

```
int has_cycle(Listp x) {
  tassume("ls(k,x,x) & k > 0");

  Listp p1 = x;
  Listp p2 = x;

  p2 = p2->next;
  while (p1 != p2) {
    if (p2 == 0 || p2->next == 0)
      return 0;
    p2 = p2->next;
    p2 = p2->next;
    p1 = p1->next;
  }
  return 1;
}
```

$$ls(\underline{k1}; p1, p2) * ls(\underline{k2}; p2, p1) \wedge \underline{k1} + \underline{k2} > 0$$

```
assume(k > 0);
k1 = 1;
k2 = k - 1;
while( k1 != 0 && k2 != 0 ) {
   if(k2 > 0)
      k2--; k1++;
   else
      k2 = k1 - 1;
      k1 = 1;
   ...
   if(k1 > 0)
      k1--; k2++;
   else
      k1 = k1 - 1;
      k2 = 1;
}
```

# Example – An odd test for cyclicity

```
int has cycle(Listp x) {
  ta   e("ls(k,x,x) & k > 0");

  Li   p1 = x;
  Li   p2 = x;

  p2   2->next;
  wh   (p1 != p2) {
       p2 == 0 || p2->next == 0)
       turn 0;
       p2->next;
       p2->next;
       p1->next;

  return 1;
}
```

p1 = p2

$$ls(\underline{k1}; p1, p2) * ls(\underline{k2}; p2, p1) \wedge \underline{k1} + \underline{k2} > 0$$

```
assume(k > 0);
k1 = 1;
k2 = k - 1;
while( k1 != 0 && k2      ) {
    if(k2 > 0)
        k2--; k1++;
    else
        k2 = k1 - 1;
        k1 = 1;
    ...
    if(k1 > 0)
        k1--; k2++;
    else
        k1 = k1 - 1
        k2 = 1;
}
```

$k1 = 0 \vee k2 = 0$

# Generalized has_cycle

```
p2 = p2->next;
p2 = p2->next;
while (p1 != p2) {
    if (p2 == 0 ||
        p2->next == 0)
      return 0;
    p2 = p2->next;
    p2 = p2->next;
    p2 = p2->next;
    p1 = p1->next;
    p1 = p1->next;
}
return 1;
```

C (the first two lines `p2 = p2->next;`)

A (the three lines `p2 = p2->next;` inside the loop)

B (the two lines `p1 = p1->next;`)

$$ls(\underline{k}; x, x) \wedge \underline{k} > 0$$

# Generalized has_cycle

```
p2 = p2->next;          // C
p2 = p2->next;          // C
while (p1 != p2) {
    if (p2 == 0 ||
        p2->next == 0)
      return 0;
    p2 = p2->next;      // A
    p2 = p2->next;      // A
    p2 = p2->next;      // A
    p1 = p1->next;      // B
    p1 = p1->next;      // B
}
return 1;
```

$ls(\underline{k}; x, x) \wedge \underline{k} > 0$

$(A \cdot k1 + B \cdot k2 + C) \bmod k = 0$

# Summary

- Can automatically track changes in data structure sizes
  - Implemented in THOR
- Resulting numeric abstraction can be passed to a variety of tools
  - Termination
  - Safety
  - Time / Space Bounds
- Support for user-defined inductive data structures