Automated Termination Proofs for Java Bytecode with Cyclic Data

M. Brockschmidt, R. Musiol, C. Otto, J. Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

WST 2012, Obergurgl

Termination Analysis for Imperative Programs

Termination Analysis for Imperative Programs

- Synthesis of Linear Ranking Functions
  (Colon & Sipma, 01), (Podelski & Rybalchenko, 04), ...

Termination Analysis for Imperative Programs

- Synthesis of Linear Ranking Functions
  (Colon & Sipma, 01), (Podelski & Rybalchenko, 04), ...
- Terminator
  Termination Analysis by Abstraction & Model Checking
  (Cook, Podelski, Rybalchenko et al., since 05)

Termination Analysis for Imperative Programs

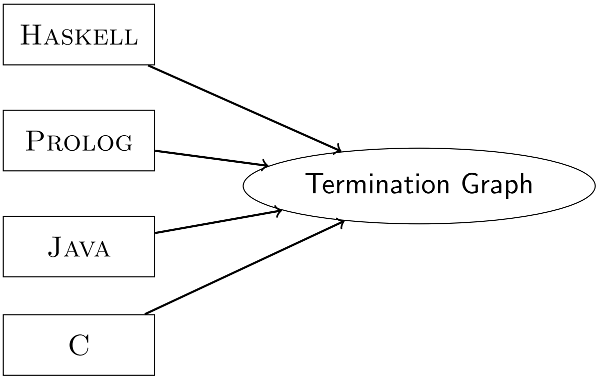
- Synthesis of Linear Ranking Functions
  (Colon & Sipma, 01), (Podelski & Rybalchenko, 04), ...
- Terminator
  Termination Analysis by Abstraction & Model Checking
  (Cook, Podelski, Rybalchenko et al., since 05)
- Julia & COSTA
  Termination Analysis of Java Bytecode (JBC)
  Fixed abstraction, via Constraint Logic Programs
  (Spoto, Mesnard, Payet, 10)
  (Albert, Arenas, Codish, Genaim, Puebla, Zanardini, 08)

Rewriting-based approach: Structure

- Programming languages *hard* $\curvearrowright$ Simpler representation needed

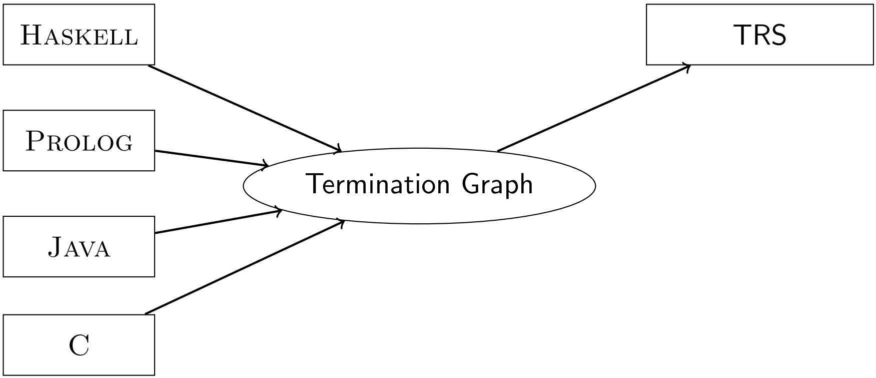
Rewriting-based approach: Structure

- Programming languages *hard* $\curvearrowright$ Simpler representation needed
- Termination Graphs: Simple, all information



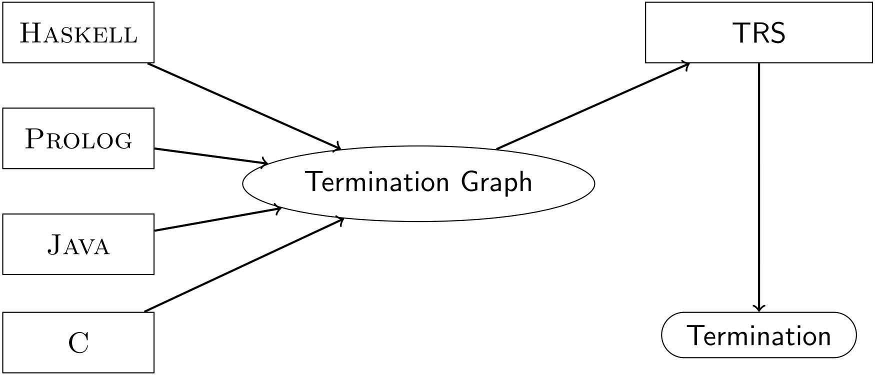
Rewriting-based approach: Structure

- Programming languages *hard* $\curvearrowright$ Simpler representation needed
- Termination Graphs: Simple, all information
- Term Rewrite Systems (TRSs) generated from Termination Graph



Rewriting-based approach: Structure

- Programming languages *hard* $\curvearrowright$ Simpler representation needed
- Termination Graphs: Simple, all information
- Term Rewrite Systems (TRSs) generated from Termination Graph
- Prove TRS termination using existing provers



Rewriting-based approach: Advantages

Handling of user-defined data structures:

```
public class List {
  int value;
  List next;
}
```

Rewriting-based approach: Advantages

Handling of user-defined data structures:

- Other techniques:
  Fixed abstraction to **number**
- List [2, 4, 6] abstracted to
  length 3

```
public class List {
  int value;
  List next;
}
```

Rewriting-based approach: Advantages

Handling of user-defined data structures:

```
public class List {
  int value;
  List next;
}
```

- Other techniques:
  Fixed abstraction to **number**
- List [2, 4, 6] abstracted to
  length 3
- Our technique:
  Abstraction to **terms**
- List [2, 4, 6] becomes
  List(2, List(4, List(6, null)))

Rewriting-based approach: Advantages

Handling of user-defined data structures:

```
public class List {
  int value;
  List next;
}
```

- Other techniques:
  Fixed abstraction to **number**
- List [2, 4, 6] abstracted to
  length 3
- Our technique:
  Abstraction to **terms**
- List [2, 4, 6] becomes
  List(2, List(4, List(6, null)))

- **TRS techniques** search for suitable orders automatically

⇒ Complex orders for user-defined data structures possible

Rewriting-based approach: Challenges

Handling of user-defined cyclic data structures:

```
public class List {
  int value;
  List next;
}
```

- Our technique:
  Abstraction to **terms** impossible

Rewriting-based approach: Challenges

Handling of user-defined cyclic data structures:

```
public class List {
  int value;
  List next;
}
```

- Our technique:
  Abstraction to **terms** impossible
- List [2, 4, 6, 2, 4, 6, . . .] is abstracted to free variable

↷ Suitable order cannot be found

Rewriting-based approach: Challenges

Handling of user-defined cyclic data structures:

```
public class List {
  int value;
  List next;
}
```

- Our technique:
  Abstraction to **terms** impossible
- List [2, 4, 6, 2, 4, 6, . . .] is abstracted to free variable

↷ Suitable order cannot be found

- Solution:
  1. Find suitable measures on Termination Graph level
  2. Encode (numeric) measures into TRS

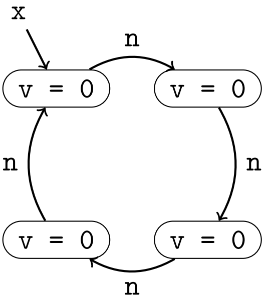
Overview

visit the example

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

visit the example

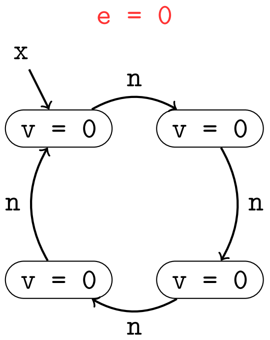
```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```



1. Store first v
2. Continue if obj. unvisited
3. Change v
4. Go to next element

visit the example

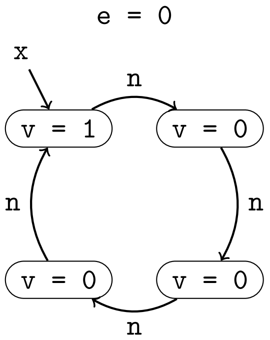
```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```



1. Store first v
2. Continue if obj. unvisited
3. Change v
4. Go to next element

visit the example

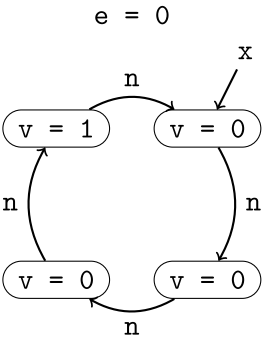
```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```



1. Store first v
2. Continue if obj. unvisited
3. Change v
4. Go to next element

visit the example

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

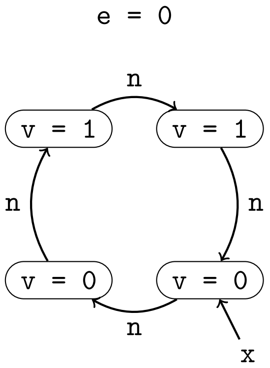


1. Store first v
2. Continue if obj. unvisited
3. Change v
4. Go to next element

visit the example

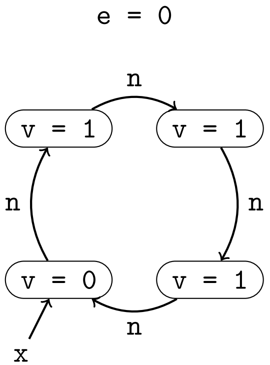
```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

1. Store first v
2. Continue if obj. unvisited
3. Change v
4. Go to next element



visit the example

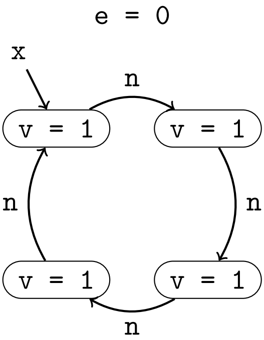
```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```



1. Store first v
2. Continue if obj. unvisited
3. Change v
4. Go to next element

visit the example

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```



1. Store first v
2. Continue if obj. unvisited
3. Change v
4. Go to next element

visit the example

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

```
00: aload_0       #load x
01: getfield v    #get v from x
04: istore_1      #store to e
05: aload_0       #load x
06: getfield v    #get v from x
09: iload_1       #load e
10: if_icmpne 28 #jump if x.v != e
13: aload_0       #load x
14: iload_1       #load e
15: iconst_1      #load 1
16: iadd          #add e and 1
17: putfield v    #store to x.v
20: aload_0       #load x
21: getfield n    #get n from x
24: astore_0      #store to x
25: goto 5
28: return
```

Abstract JAVA virtual machine states

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

```
00: aload_0       #load x
01: getfield v    #get v from x
04: istore_1      #store to e
05: aload_0       #load x
06: getfield v    #get v from x
09: iload_1       #load e
10: if_icmpne 28 #jump if x.v != e
13: aload_0       #load x
14: iload_1       #load e
15: iconst_1      #load 1
16: iadd          #add e and 1
17: putfield v    #store to x.v
20: aload_0       #load x
21: getfield n    #get n from x
24: astore_0      #store to x
25: goto 5
28: return
```

$$\frac{05 \,|\, \mathtt{x}: o_1, \mathtt{e}: i_1 \,|\, \varepsilon}{o_1 : \mathtt{L}(?) \qquad i_1 : \mathbb{Z} \qquad o_1 \circlearrowleft}$$

Abstract JAVA virtual machine states

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

```
00: aload_0        #load x
01: getfield v     #get v from x
04: istore_1       #store to e
05: aload_0        #load x
06: getfield v     #get v from x
09: iload_1        #load e
10: if_icmpne 28 #jump if x.v != e
13: aload_0        #load x
14: iload_1        #load e
15: iconst_1       #load 1
16: iadd           #add e and 1
17: putfield v     #store to x.v
20: aload_0        #load x
21: getfield n     #get n from x
24: astore_0       #store to x
25: goto 5
28: return
```

$05 \mid \mathtt{x}: o_1, \mathtt{e}: i_1 \mid \varepsilon$		
o_1: L(?)	$i_1 : \mathbb{Z}$	$o_1 ↺$

Stack frame:

- Next program instruction

Abstract JAVA virtual machine states

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

```
00: aload_0        #load x
01: getfield v     #get v from x
04: istore_1       #store to e
05: aload_0        #load x
06: getfield v     #get v from x
09: iload_1        #load e
10: if_icmpne 28 #jump if x.v != e
13: aload_0        #load x
14: iload_1        #load e
15: iconst_1       #load 1
16: iadd           #add e and 1
17: putfield v     #store to x.v
20: aload_0        #load x
21: getfield n     #get n from x
24: astore_0       #store to x
25: goto 5
28: return
```

$05 \mid \mathtt{x} : o_1, \mathtt{e} : i_1 \mid \varepsilon$

$o_1 : \mathtt{L}(?) \quad i_1 : \mathbb{Z} \quad o_1 ↺$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Abstract JAVA virtual machine states

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

```
00: aload_0       #load x
01: getfield v    #get v from x
04: istore_1      #store to e
05: aload_0       #load x
06: getfield v    #get v from x
09: iload_1       #load e
10: if_icmpne 28 #jump if x.v != e
13: aload_0       #load x
14: iload_1       #load e
15: iconst_1      #load 1
16: iadd          #add e and 1
17: putfield v    #store to x.v
20: aload_0       #load x
21: getfield n    #get n from x
24: astore_0      #store to x
25: goto 5
28: return
```

$05 \mid \mathtt{x}: o_1, \mathtt{e}: i_1 \mid \varepsilon$
o_1: L(?) $\quad i_1: \mathbb{Z} \quad o_1 \circlearrowleft$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Heap information:

Abstract JAVA virtual machine states

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

```
00: aload_0       #load x
01: getfield v    #get v from x
04: istore_1      #store to e
05: aload_0       #load x
06: getfield v    #get v from x
09: iload_1       #load e
10: if_icmpne 28 #jump if x.v != e
13: aload_0       #load x
14: iload_1       #load e
15: iconst_1      #load 1
16: iadd          #add e and 1
17: putfield v    #store to x.v
20: aload_0       #load x
21: getfield n    #get n from x
24: astore_0      #store to x
25: goto 5
28: return
```

$05 \mid \mathtt{x}: o_1, \mathtt{e}: i_1 \mid \varepsilon$		
$o_1 : \mathtt{L}(?)$	$i_1 : \mathbb{Z}$	$o_1 \circlearrowleft$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Heap information:

- At o_1 is `L` object or `null`

Abstract Java virtual machine states

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

```
00: aload_0        #load x
01: getfield v     #get v from x
04: istore_1       #store to e
05: aload_0        #load x
06: getfield v     #get v from x
09: iload_1        #load e
10: if_icmpne 28   #jump if x.v != e
13: aload_0        #load x
14: iload_1        #load e
15: iconst_1       #load 1
16: iadd           #add e and 1
17: putfield v     #store to x.v
20: aload_0        #load x
21: getfield n     #get n from x
24: astore_0       #store to x
25: goto 5
28: return
```

$05 \mid \mathtt{x}: o_1, \mathtt{e}: i_1 \mid \varepsilon$

$o_1: \mathtt{L}(?) \quad i_1: \mathbb{Z} \quad o_1 \circlearrowleft$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Heap information:

- At o_1 is `L` object or `null`
- At i_1 is unknown integer

Abstract JAVA virtual machine states

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

```
00: aload_0       #load x
01: getfield v    #get v from x
04: istore_1      #store to e
05: aload_0       #load x
06: getfield v    #get v from x
09: iload_1       #load e
10: if_icmpne 28 #jump if x.v != e
13: aload_0       #load x
14: iload_1       #load e
15: iconst_1      #load 1
16: iadd          #add e and 1
17: putfield v    #store to x.v
20: aload_0       #load x
21: getfield n    #get n from x
24: astore_0      #store to x
25: goto 5
28: return
```

$05 \,|\, \mathtt{x}: o_1, \mathtt{e}: i_1 \,|\, \varepsilon$

$o_1 : \mathtt{L}(?) \quad i_1 : \mathbb{Z} \quad o_1 \circlearrowleft$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Heap information:

- At o_1 is L object or `null`
- At i_1 is unknown integer
- Known L object: $o_2 : \mathtt{L}(\mathtt{v} = i_2, \mathtt{n} = o_3)$

Abstract JAVA virtual machine states

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

```
00: aload_0       #load x
01: getfield v    #get v from x
04: istore_1      #store to e
05: aload_0       #load x
06: getfield v    #get v from x
09: iload_1       #load e
10: if_icmpne 28  #jump if x.v != e
13: aload_0       #load x
14: iload_1       #load e
15: iconst_1      #load 1
16: iadd          #add e and 1
17: putfield v    #store to x.v
20: aload_0       #load x
21: getfield n    #get n from x
24: astore_0      #store to x
25: goto 5
28: return
```

$05 \mid \mathtt{x} : o_1, \mathtt{e} : i_1 \mid \varepsilon$

$o_1 : \mathtt{L}(?) \quad i_1 : \mathbb{Z} \quad o_1 \circlearrowleft$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Heap information:

- At o_1 is L object or `null`
- At i_1 is unknown integer
- Known L object: $o_2 : \mathtt{L}(\mathtt{v} = i_2, \mathtt{n} = o_3)$

Only explicit sharing

Abstract Java virtual machine states

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

```
00: aload_0       #load x
01: getfield v    #get v from x
04: istore_1      #store to e
05: aload_0       #load x
06: getfield v    #get v from x
09: iload_1       #load e
10: if_icmpne 28 #jump if x.v != e
13: aload_0       #load x
14: iload_1       #load e
15: iconst_1      #load 1
16: iadd          #add e and 1
17: putfield v    #store to x.v
20: aload_0       #load x
21: getfield n    #get n from x
24: astore_0      #store to x
25: goto 5
28: return
```

$05 \mid \mathtt{x}: o_1, \mathtt{e}: i_1 \mid \varepsilon$		
o_1: L(?)	i_1: $\mathbb{Z}$	$o_1 ↻$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Heap information:

- At o_1 is L object or null
- At i_1 is unknown integer
- Known L object: $o_2 : \mathtt{L}(\mathtt{v} = i_2, \mathtt{n} = o_3)$

Heap annotations: Only explicit sharing

- Reference might be cyclic: $o_1 ↻$

Abstract JAVA virtual machine states

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

```
00: aload_0        #load x
01: getfield v     #get v from x
04: istore_1       #store to e
05: aload_0        #load x
06: getfield v     #get v from x
09: iload_1        #load e
10: if_icmpne 28   #jump if x.v != e
13: aload_0        #load x
14: iload_1        #load e
15: iconst_1       #load 1
16: iadd           #add e and 1
17: putfield v     #store to x.v
20: aload_0        #load x
21: getfield n     #get n from x
24: astore_0       #store to x
25: goto 5
28: return
```

$05 \mid \mathtt{x} : o_1, \mathtt{e} : i_1 \mid \varepsilon$
$o_1 : \mathtt{L}(?)$ $\quad i_1 : \mathbb{Z}$ $\quad o_1 ↺$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Heap information:

- At o_1 is `L` object or `null`
- At i_1 is unknown integer
- Known `L` object: $o_2 : \mathtt{L}(\mathtt{v} = i_2, \mathtt{n} = o_3)$

Heap annotations: Only explicit sharing

- Reference might be cyclic: $o_1 ↺$
- Two references may be equal: $o_1 =^? o_2$

Abstract JAVA virtual machine states

```
class L {
  int v;    List n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

```
00: aload_0        #load x
01: getfield v     #get v from x
04: istore_1       #store to e
05: aload_0        #load x
06: getfield v     #get v from x
09: iload_1        #load e
10: if_icmpne 28   #jump if x.v != e
13: aload_0        #load x
14: iload_1        #load e
15: iconst_1       #load 1
16: iadd           #add e and 1
17: putfield v     #store to x.v
20: aload_0        #load x
21: getfield n     #get n from x
24: astore_0       #store to x
25: goto 5
28: return
```

$05 | \mathtt{x} : o_1, \mathtt{e} : i_1 | \varepsilon$

$o_1 : \mathtt{L}(?) \quad i_1 : \mathbb{Z} \quad o_1 \circlearrowleft$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

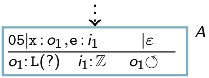
Heap information:

- At o_1 is `L` object or `null`
- At i_1 is unknown integer
- Known `L` object: $o_2 : \mathtt{L}(\mathtt{v} = i_2, \mathtt{n} = o_3)$

Heap annotations: Only explicit sharing

- Reference might be cyclic: $o_1 \circlearrowleft$
- Two references may be equal: $o_1 =^? o_2$
- Two references may share: $o_1 \searrow\swarrow o_2$

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```

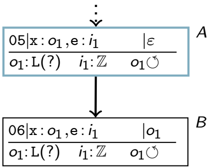


State *A*:

- x some (possibly cyclic) list
- e some integer

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```

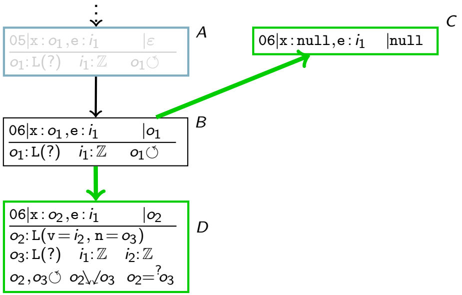


State *B*:

- *Evaluation* between *A* and *B*
- Need field of o_1

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```

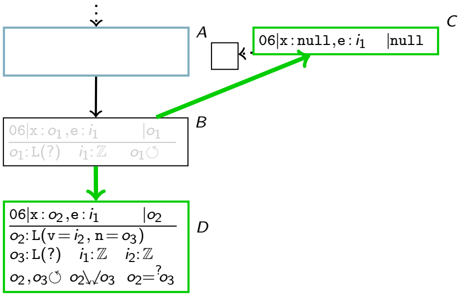



```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

States *B*, *C*, *D*:

- *Evaluation* between *A* and *B*
- Need field of o_1 ⇒ Refinement:
  - In *C*: o_1 is `null`
  - In *D*: o_1 renamed to o_2, pointing to L-object with successor o_3:
    - o_3 possibly cyclic
    - o_3 possibly equal to o_2 and may reach o_2

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```

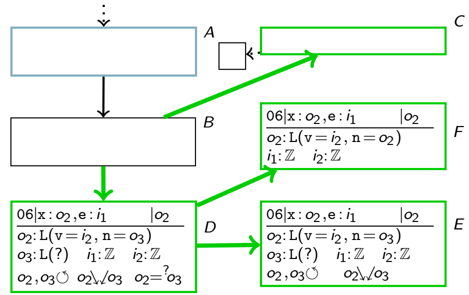


States *B*, *C*, *D*:

- *Evaluation* between *A* and *B*
- Need field of o_1 $\Rightarrow$ Refinement:
  - In *C*: o_1 is `null` (program crashes)
  - In *D*: o_1 renamed to o_2, pointing to L-object with successor o_3:
    - o_3 possibly cyclic
    - o_3 possibly equal to o_2 and may reach o_2

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```

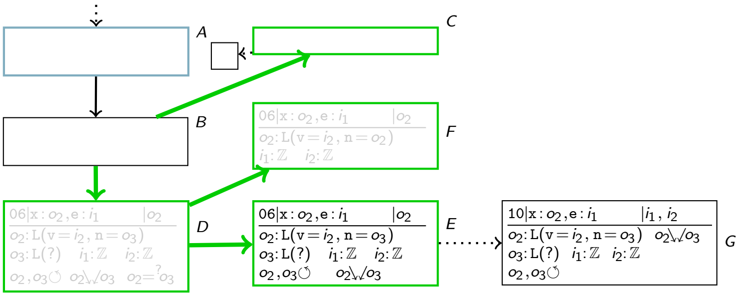


States E, F:

- Need to read field of o_2 $\Rightarrow$ Refinement
  - In E: $o_2 \neq o_3$
  - In F: $o_2 = o_3$

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```

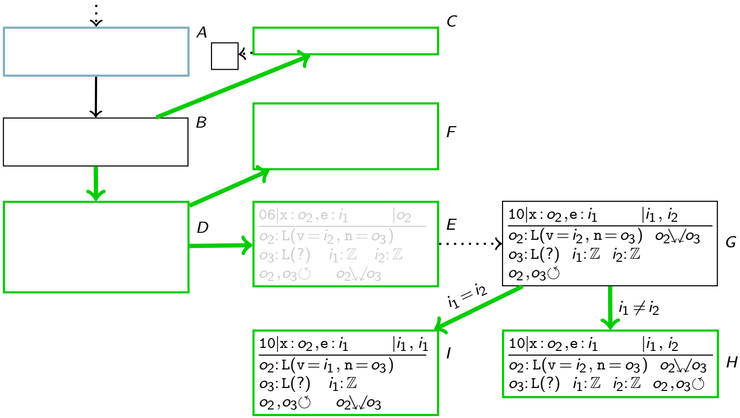


State G:

- Evaluation: Read v, loaded e
- Need to decide $i_1 \neq i_2$

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```

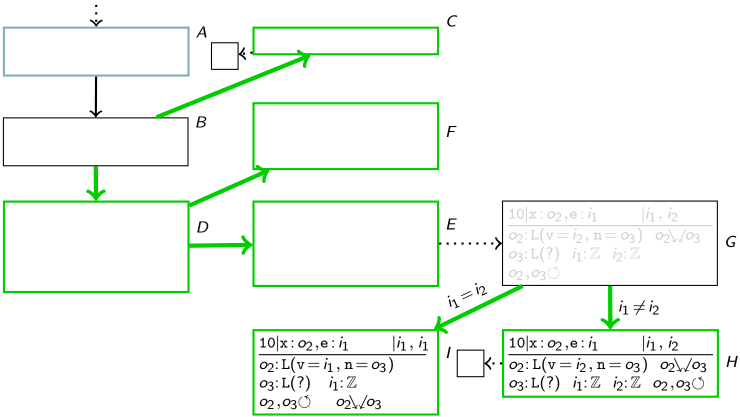



```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

States G, I, H:

- Evaluation: Read v, loaded e
- Need to decide $i_1 \neq i_2 \Rightarrow$ Refinement:
  - In I: $i_1 = i_2$
  - In H: $i_1 \neq i_2$

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```

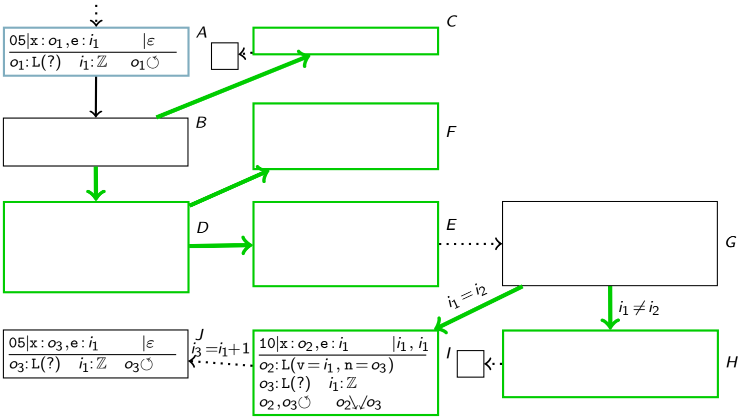



```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

States G, I, H:

- Evaluation: Read v, loaded e
- Need to decide $i_1 \neq i_2 \Rightarrow$ Refinement:
  - In I: $i_1 = i_2$ (program ends)
  - In H: $i_1 \neq i_2$

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```

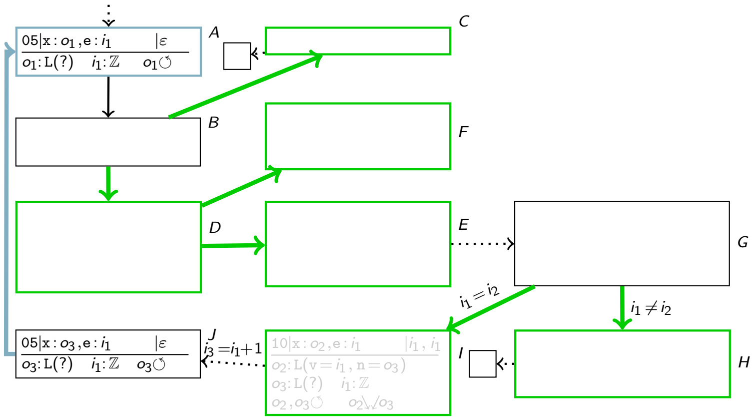


States G, I, H:

- Evaluation: Read v, loaded e
- Need to decide $i_1 \neq i_2 \Rightarrow$ Refinement:
  - In I: $i_1 = i_2$ (program ends)
  - In H: $i_1 \neq i_2$
- State J reached by evaluation

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```

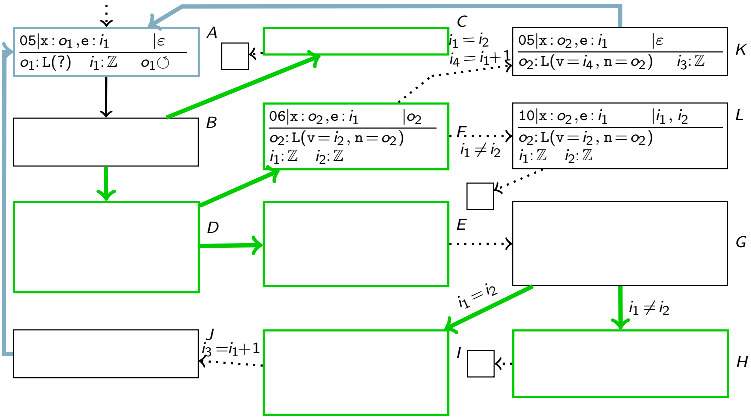



```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

States G, I, H:

- Evaluation: Read v, loaded e
- Need to decide $i_1 \neq i_2 \Rightarrow$ Refinement:
  - In I: $I_1 = i_2$ (program ends)
  - In H: $i_1 \neq i_2$
- State J reached by evaluation, represented by (instance of) A

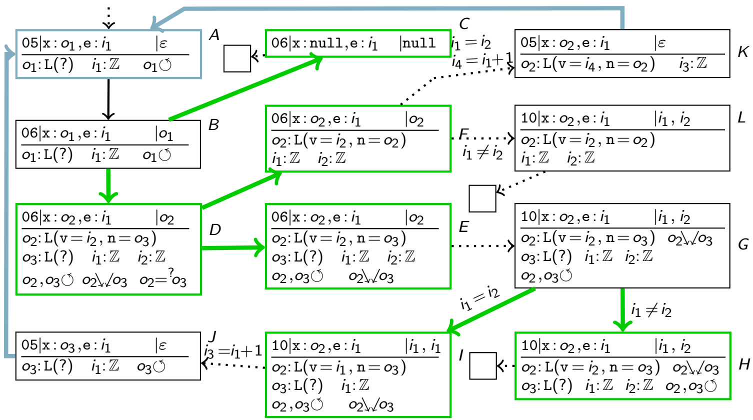
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



States K, L: Analogous for one-element list

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

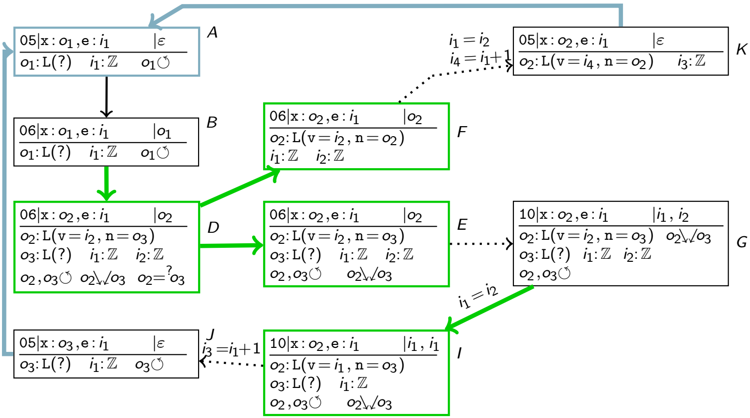
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



- All leaves program ends $\Rightarrow$ Graph finished
- How can we prove termination?

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

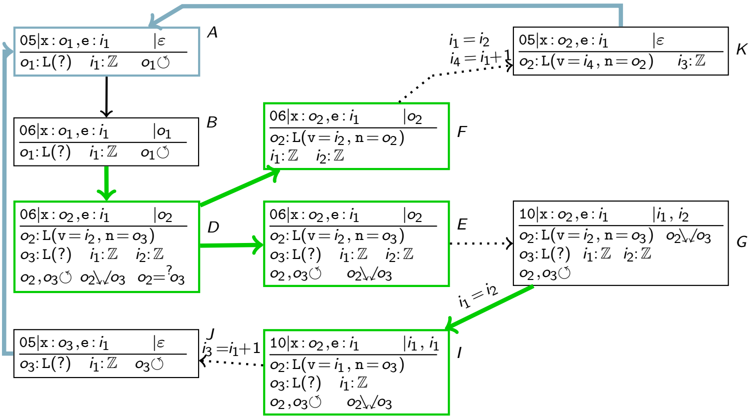
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



- All leaves program ends ⇒ Graph finished
- How can we prove termination?
- Only consider SCCs

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```

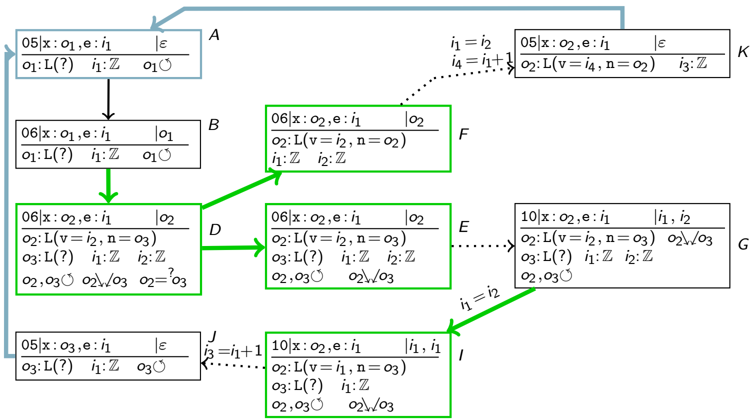


- All leaves program ends ⇒ Graph finished
- How can we prove termination?
- Only consider SCCs

High-level argument: Number of unvisited elements strictly decreasing

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



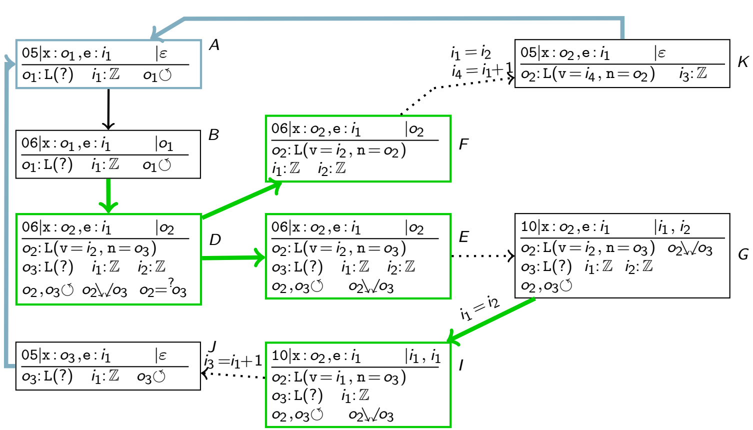
- All leaves program ends ⇒ Graph finished
- How can we prove termination?
- Only consider SCCs

  High-level argument: Number of unvisited elements strictly decreasing

... Let's drag that down to our level!

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

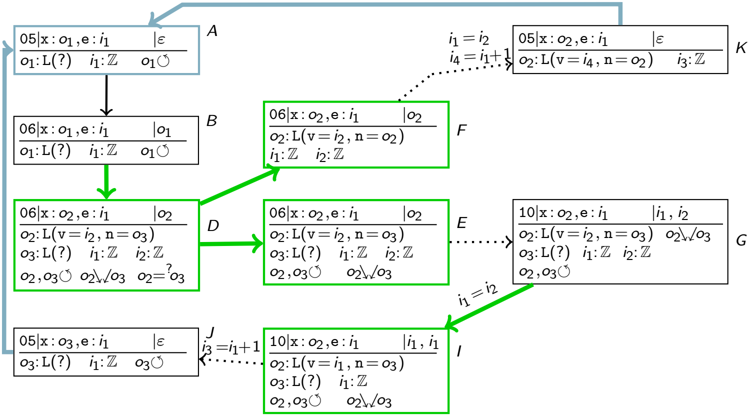
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



Q: What is an "unvisited element", formally?

```java
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```

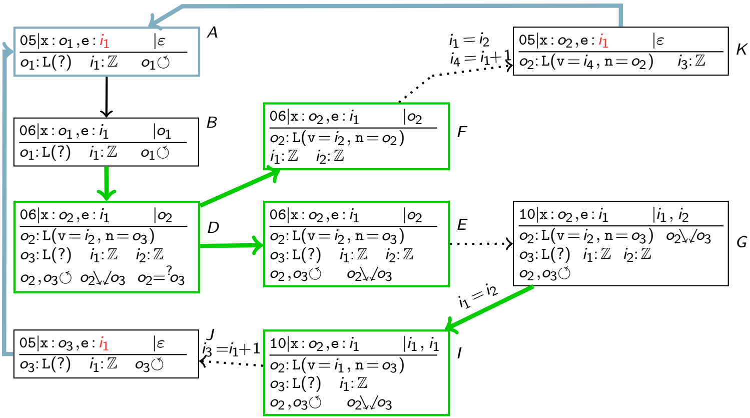


Q: What is an “unvisited element”, formally?

A: One with $\mathtt{L.v} = i_1 = \mathtt{e}$

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



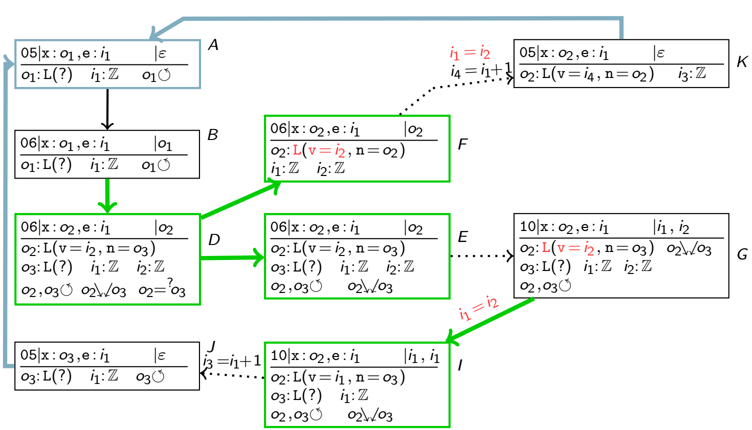
Q: What is an "unvisited element", formally?

A: One with $\mathrm{L.v} = i_1 = \mathrm{e}$

- Automatically finding this relation:
  1. Identify constant c in SCC

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



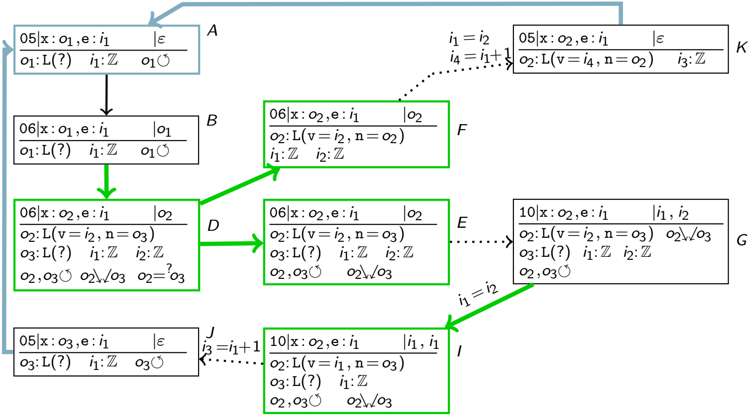

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

Q: What is an "unvisited element", formally?

A: One with $\mathtt{L.v} = i_1 = \mathtt{e}$

- Automatically finding this relation:
  1. Identify constant c in SCC
  2. Search property $M = \mathtt{C.f} \bowtie c$ checked on all cycles

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



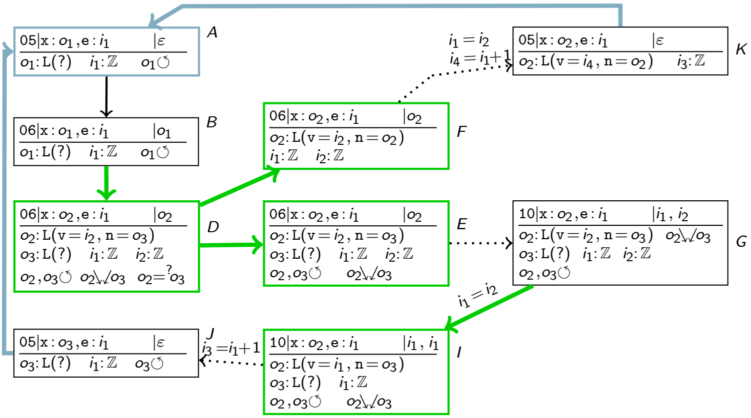

```java
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

Q: What is an "unvisited element", formally?

A: One with $\mathtt{L.v} = i_1 = \mathtt{e}$

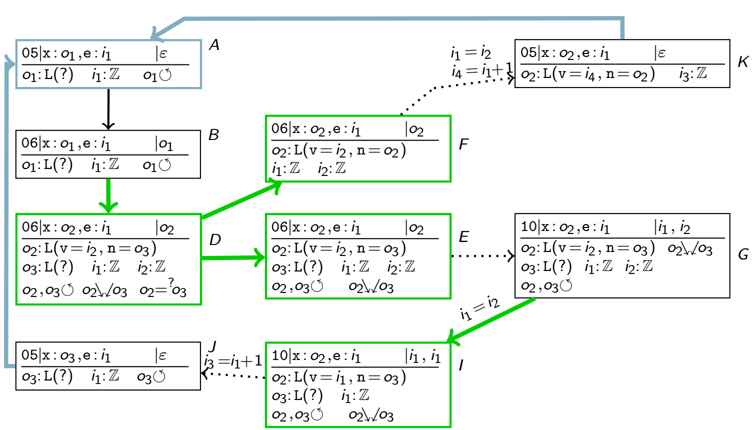
- Automatically finding this relation:
  1. Identify constant c in SCC
  2. Search property $M = \mathtt{C.f} \bowtie c$ checked on all cycles
- Track number of objects where $\mathtt{C.f} \bowtie c$ holds ($\#_M$)

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



Property $M = \texttt{C.f} \bowtie \mathsf{c}$ (here: $\mathsf{c} = i_1$). When does $\#_M$ change?

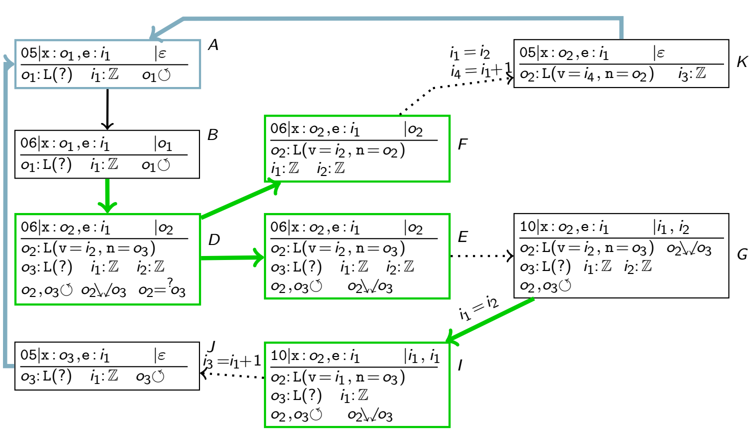
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



Property $M = \texttt{C.f} \bowtie \mathsf{c}$ (here: $\mathsf{c} = i_1$). When does $\#_M$ change?

- C.f written (old value u, new value w):

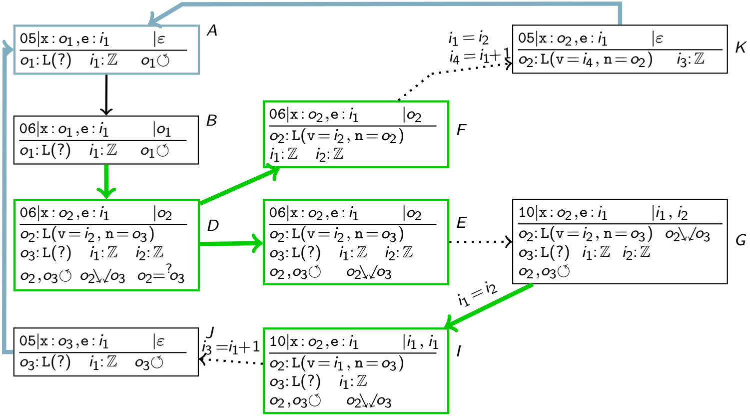
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



Property $M = \texttt{C.f} \bowtie \texttt{c}$ (here: $\texttt{c} = i_1$). When does $\#_M$ change?

- C.f written (old value u, new value w):
  - $u \bowtie c \wedge \neg w \bowtie c$ tautology $\Rightarrow \#_M$ decremented by 1

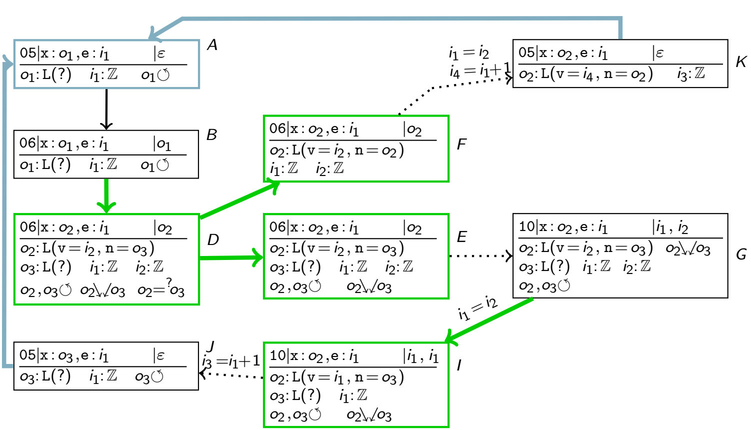
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



Property $M = \texttt{C.f} \bowtie c$ (here: $c = i_1$). When does $\#_M$ change?

- `C.f` written (old value u, new value w):
  - $u \bowtie c \wedge \neg w \bowtie c$ tautology $\Rightarrow$ $\#_M$ decremented by 1
  - $u \bowtie c \leftrightarrow w \bowtie c$ tautology $\Rightarrow$ $\#_M$ unchanged

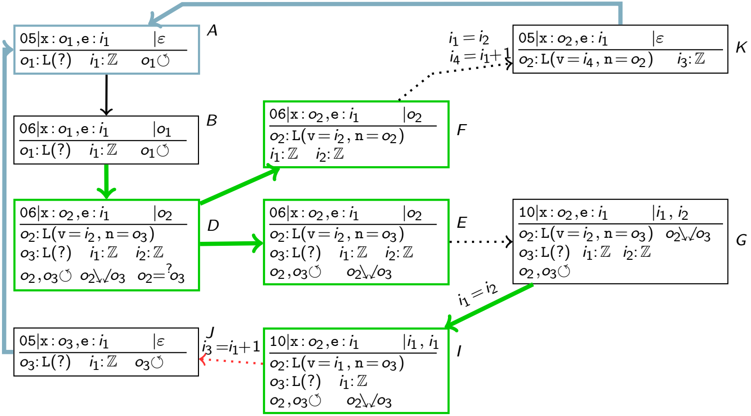
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



Property $M = \mathtt{C.f} \bowtie \mathtt{c}$ (here: $\mathtt{c} = i_1$). When does $\#_M$ change?

- $\mathtt{C.f}$ written (old value u, new value w):
  - $u \bowtie c \wedge \neg w \bowtie c$ tautology $\Rightarrow \#_M$ decremented by 1
  - $u \bowtie c \leftrightarrow w \bowtie c$ tautology $\Rightarrow \#_M$ unchanged
  - Otherwise: $\#_M$ incremented by 1.

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



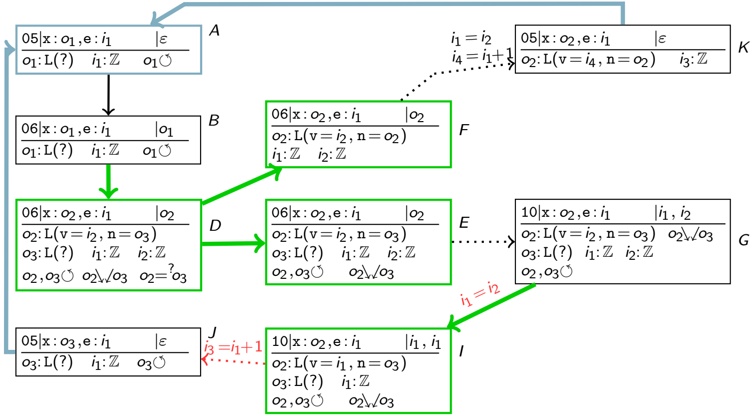
Property $M = \texttt{C.f} \bowtie \mathtt{c}$ (here: $\mathtt{c} = i_1$). When does $\#_M$ change?

- $\texttt{C.f}$ written (old value u, new value w):
  - $u \bowtie c \wedge \neg w \bowtie c$ tautology $\Rightarrow \#_M$ decremented by 1
  - $u \bowtie c \leftrightarrow w \bowtie c$ tautology $\Rightarrow \#_M$ unchanged
  - Otherwise: $\#_M$ incremented by 1.

In example: $I \to J$: i_1 old, i_3 new

$\Rightarrow$ $i_1 = i_1 \wedge \neg i_3 = i_1$

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



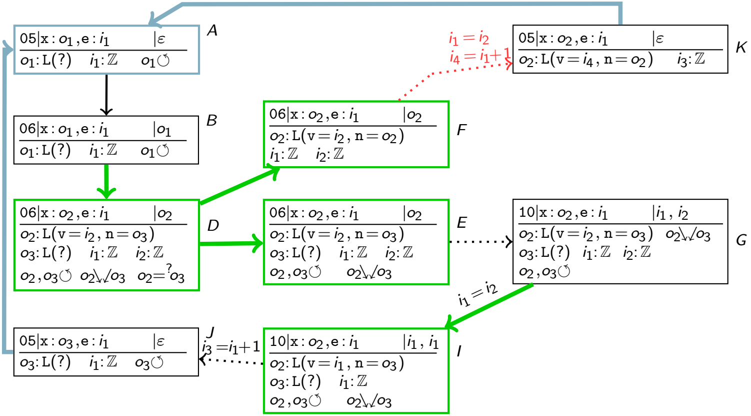
Property $M = \mathtt{C.f} \bowtie c$ (here: $c = i_1$). When does $\#_M$ change?

- `C.f` written (old value u, new value w):
  - $u \bowtie c \wedge \neg w \bowtie c$ tautology $\Rightarrow \#_M$ decremented by 1
  - $u \bowtie c \leftrightarrow w \bowtie c$ tautology $\Rightarrow \#_M$ unchanged
  - Otherwise: $\#_M$ incremented by 1.

In example: $I \to J$: i_1 old, i_3 new

$\Rightarrow i_1 = i_2 \wedge i_3 = i_1 + 1 \ \rightarrow\ i_1 = i_1 \wedge \neg i_3 = i_1$

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



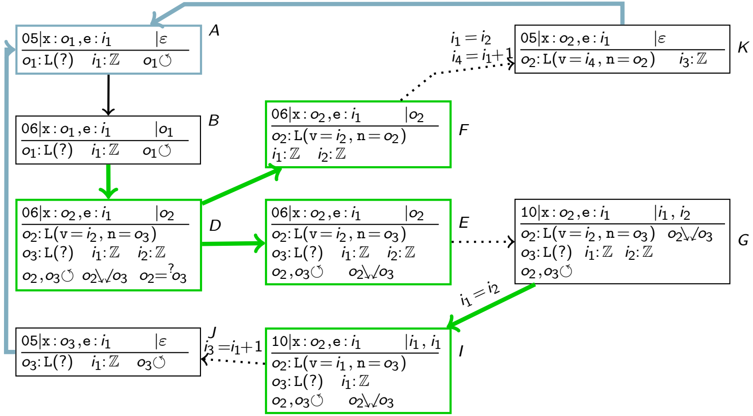
Property $M = \texttt{C.f} \bowtie \texttt{c}$ (here: $\texttt{c} = i_1$). When does $\#_M$ change?

- $\texttt{C.f}$ written (old value u, new value w):
  - $u \bowtie c \wedge \neg w \bowtie c$ tautology $\Rightarrow \#_M$ decremented by 1
  - $u \bowtie c \leftrightarrow w \bowtie c$ tautology $\Rightarrow \#_M$ unchanged
  - Otherwise: $\#_M$ incremented by 1.

In example: $F \to K$: i_1 old, i_4 new

$\Rightarrow i_1 = i_2 \wedge i_4 = i_1 + 1 \;\to\; i_1 = i_1 \wedge \neg i_4 = i_1$

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```

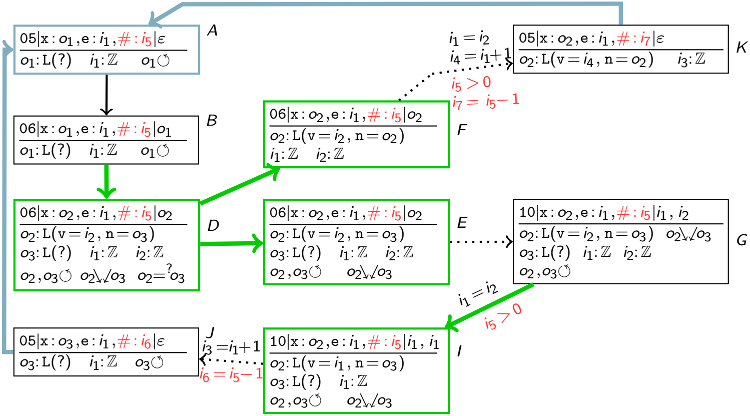


Property $M = \texttt{C.f} \bowtie \texttt{c}$ (here: $\texttt{c} = i_1$). When does $\#_M$ change?

- `C.f` written (old value u, new value w):
  - $u \bowtie c \wedge \neg w \bowtie c$ tautology $\Rightarrow \#_M$ decremented by 1
  - $u \bowtie c \leftrightarrow w \bowtie c$ tautology $\Rightarrow \#_M$ unchanged
  - Otherwise: $\#_M$ incremented by 1.

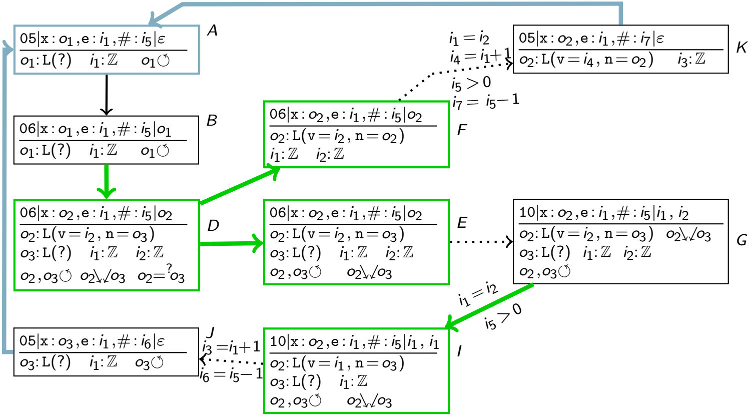
- New `L` object is created: Same for default value

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



- Add variable for counter to states, changes to edges
- Require counter > 0 at checks

```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



- Add variable for counter to states, changes to edges
- Require counter > 0 at checks
- Termination proof via TRS now trivial:

$$f_A(\dots, i_5) \to f_I(\dots, i_5) \quad | \; i_5 > 0$$

$$f_I(\dots, i_5) \to f_J(\dots, i_5 - 1)$$

$$f_J(\dots, i_6) \to f_A(\dots, i_6)$$

$$f_A(\dots, i_5) \to f_F(\dots, i_5)$$

$$f_F(\dots, i_5) \to f_K(\dots, i_5 - 1) \quad | \; i_5 > 0$$

$$f_F(\dots, i_7) \to f_A(\dots, i_7)$$

Overview

The iterate example

```
00: aload_0        #load this
01: getfield n     #get n from this
04: astore_1       #store to x
05: aload_1        #load x
06: aload_0        #load this
07: if_acmpeq 18 #jump if x == this
10: aload_1        #load x
11: getfield n     #get n from x
14: astore_1       #store x
15: goto 05
18: return
```

```
class L {
  L n;
  void iterate() {
    L x = this.n;
    while (x != this)
      x = x.n; }}
```

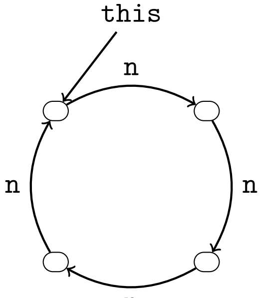
1. Keep first element in `this`
2. Iterate until reaching it again

The iterate example

```
00: aload_0         #load this
01: getfield n      #get n from this
04: astore_1        #store to x
05: aload_1         #load x
06: aload_0         #load this
07: if_acmpeq 18    #jump if x == this
10: aload_1         #load x
11: getfield n      #get n from x
14: astore_1        #store x
15: goto 05
18: return
```

```
class L {
  L n;
  void iterate() {
    L x = this.n;
    while (x != this)
      x = x.n; }}
```

1. Keep first element in `this`
2. Iterate until reaching it again

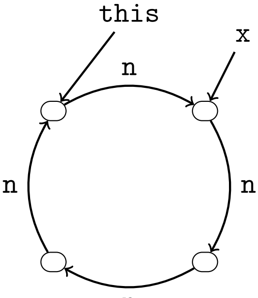


The iterate example

```
00: aload_0        #load this
01: getfield n     #get n from this
04: astore_1       #store to x
05: aload_1        #load x
06: aload_0        #load this
07: if_acmpeq 18   #jump if x == this
10: aload_1        #load x
11: getfield n     #get n from x
14: astore_1       #store x
15: goto 05
18: return
```

```
class L {
  L n;
  void iterate() {
    L x = this.n;
    while (x != this)
      x = x.n; }}
```

1. Keep first element in `this`
2. Iterate until reaching it again

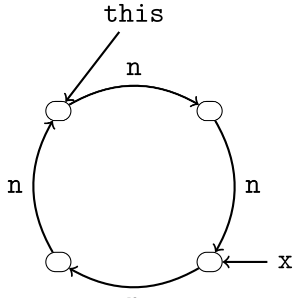


The iterate example

```
00: aload_0         #load this
01: getfield n      #get n from this
04: astore_1        #store to x
05: aload_1         #load x
06: aload_0         #load this
07: if_acmpeq 18    #jump if x == this
10: aload_1         #load x
11: getfield n      #get n from x
14: astore_1        #store x
15: goto 05
18: return
```

```
class L {
  L n;
  void iterate() {
    L x = this.n;
    while (x != this)
      x = x.n; }}
```

1. Keep first element in `this`
2. Iterate until reaching it again

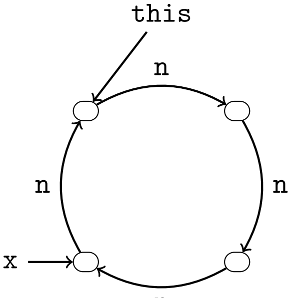


The iterate example

```
00: aload_0        #load this
01: getfield n     #get n from this
04: astore_1       #store to x
05: aload_1        #load x
06: aload_0        #load this
07: if_acmpeq 18 #jump if x == this
10: aload_1        #load x
11: getfield n     #get n from x
14: astore_1       #store x
15: goto 05
18: return
```

```
class L {
  L n;
  void iterate() {
    L x = this.n;
    while (x != this)
      x = x.n; }}
```

1. Keep first element in this
2. Iterate until reaching it again

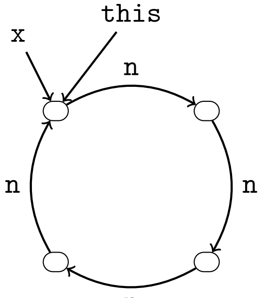


The iterate example

```
00: aload_0         #load this
01: getfield n      #get n from this
04: astore_1        #store to x
05: aload_1         #load x
06: aload_0         #load this
07: if_acmpeq 18    #jump if x == this
10: aload_1         #load x
11: getfield n      #get n from x
14: astore_1        #store x
15: goto 05
18: return
```

```
class L {
  L n;
  void iterate() {
    L x = this.n;
    while (x != this)
      x = x.n; }}
```

1. Keep first element in `this`
2. Iterate until reaching it again



Definite Reachability

```
00: aload_0        #load this
01: getfield n     #get n from this
04: astore_1       #store to x
05: aload_1        #load x
06: aload_0        #load this
07: if_acmpeq 18 #jump if x == this
10: aload_1        #load x
11: getfield n     #get n from x
14: astore_1       #store x
15: goto 05
18: return
```

```
class L {
  L n;
  void iterate() {
    L x = this.n;
    while (x != this)
      x = x.n; }}
```

1. Keep first element in `this`
2. Iterate until reaching it again

New annotation: Definite reachability $\overset{F}{\dashrightarrow}!$

- $o \overset{F}{\dashrightarrow}! o' \Rightarrow$
  All paths from o using fields from F reach o'

Definite Reachability

```
00: aload_0       #load this
01: getfield n    #get n from this
04: astore_1      #store to x
05: aload_1       #load x
06: aload_0       #load this
07: if_acmpeq 18 #jump if x == this
10: aload_1       #load x
11: getfield n    #get n from x
14: astore_1      #store x
15: goto 05
18: return
```

```
class L {
  L n;
  void iterate() {
    L x = this.n;
    while (x != this)
      x = x.n; }}
```

1. Keep first element in `this`
2. Iterate until reaching it again

New annotation: Definite reachability $\overset{F}{\dashrightarrow}!$

- $o \overset{F}{\dashrightarrow}! o' \Rightarrow$
  All paths from o using fields from F reach o'
- $=^?, \curlyvee, \circlearrowleft$ *extending* annotations:
  Allow (not enforce) sharing/shapes

Definite Reachability

```
00: aload_0        #load this
01: getfield n     #get n from this
04: astore_1       #store to x
05: aload_1        #load x
06: aload_0        #load this
07: if_acmpeq 18   #jump if x == this
10: aload_1        #load x
11: getfield n     #get n from x
14: astore_1       #store x
15: goto 05
18: return
```

```
class L {
  L n;
  void iterate() {
    L x = this.n;
    while (x != this)
      x = x.n; }}
```

1. Keep first element in `this`
2. Iterate until reaching it again

New annotation: Definite reachability $\overset{F}{\dashrightarrow}{}^{!}$

- $o \overset{F}{\dashrightarrow}{}^{!} o' \Rightarrow$
  All paths from o using fields from F reach o'
- $=^?, \curlyvee, \circlearrowleft$ *extending* annotations:
  Allow (not enforce) sharing/shapes
- $\overset{F}{\dashrightarrow}{}^{!}$ *restricting* annotation:
  Enforce sharing

Definite Reachability

```
00: aload_0        #load this
01: getfield n     #get n from this
04: astore_1       #store to x
05: aload_1        #load x
06: aload_0        #load this
07: if_acmpeq 18 #jump if x == this
10: aload_1        #load x
11: getfield n     #get n from x
14: astore_1       #store x
15: goto 05
18: return
```

```
class L {
  L n;
  void iterate() {
    L x = this.n;
    while (x != this)
      x = x.n; }}
```

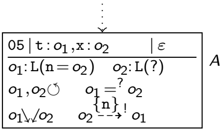
1. Keep first element in `this`
2. Iterate until reaching it again

New annotation: Definite reachability $\overset{F}{\dashrightarrow}^{!}$

- $o \overset{F}{\dashrightarrow}^{!} o' \Rightarrow$
  All paths from o using fields from F reach o'
- $=^?$, $\bigvee\!\!\bigvee$, $\circlearrowleft$ *extending* annotations:
  Allow (not enforce) sharing/shapes
- $\overset{F}{\dashrightarrow}^{!}$ *restricting* annotation:
  Enforce sharing

$$\begin{array}{l} 05 \mid \mathtt{t}: o_1, \mathtt{x}: o_2 \mid \varepsilon \\ \hline o_1 : \mathtt{L}(\mathtt{n} = o_2) \quad o_2 : \mathtt{L}(?) \\ o_1, o_2 \circlearrowleft \quad o_1 =^? o_2 \\ o_1 \bigvee\!\!\bigvee o_2 \quad o_2 \overset{\{\mathtt{n}\}}{\dashrightarrow}^{!} o_1 \end{array}$$

```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return
```

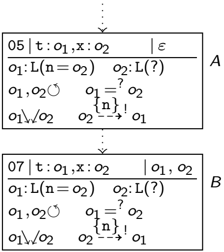


State *A*:

- t some definitely cyclic list
- x second element in list

```
void iterate() {
  L x = this.n;
  while (x != this)
    x = x.n; }
```

```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return
```



State *A*:

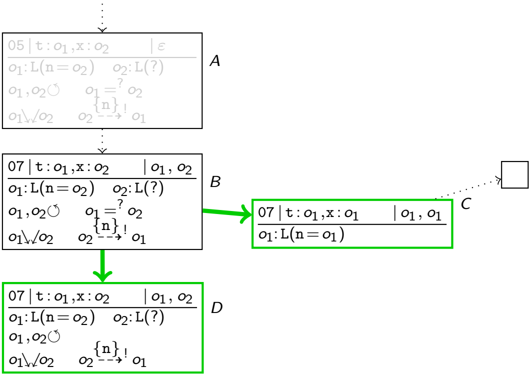
- t some definitely cyclic list
- x second element in list

State *B*:

- First equals second element?

```
void iterate() {
  L x = this.n;
  while (x != this)
    x = x.n; }
```

```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return
```




```
void iterate() {
  L x = this.n;
  while (x != this)
    x = x.n; }
```

State *A*:

- t some definitely cyclic list
- x second element in list

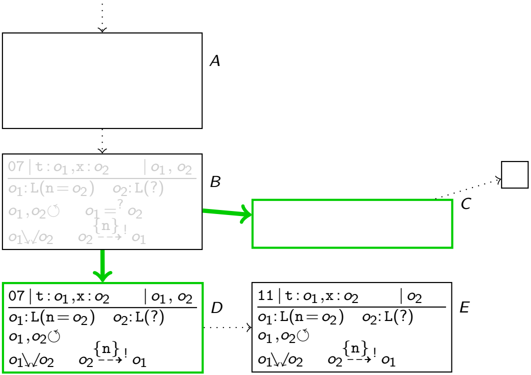
State *B*:

- First equals second element?

⇒ Refinement

- In *C*: References equal (↷ program ends)
- In *D*: References not equal

```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return
```

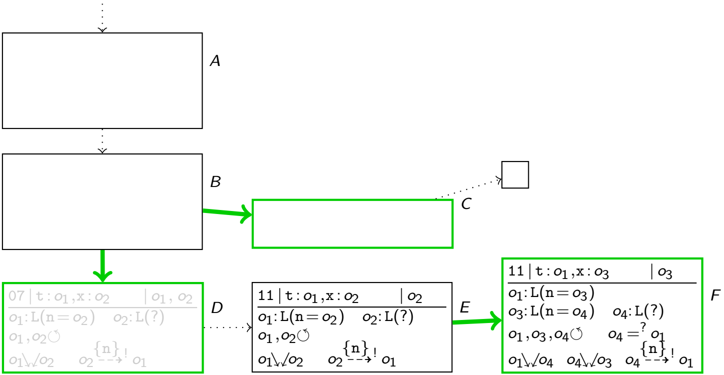


State *E*:

- Access to unknown object o_2

```
void iterate() {
  L x = this.n;
  while (x != this)
    x = x.n; }
```

```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return
```

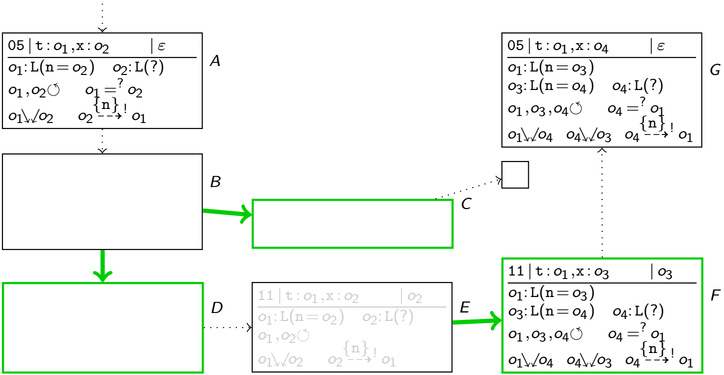



```
void iterate() {
  L x = this.n;
  while (x != this)
    x = x.n; }
```

States E, F:

- Access to unknown object o_2
  $\Rightarrow$ Refinement
- Case $o_2 = \texttt{null}$ not possible (implies o_2 not reaching o_1)

```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return
```



State G:

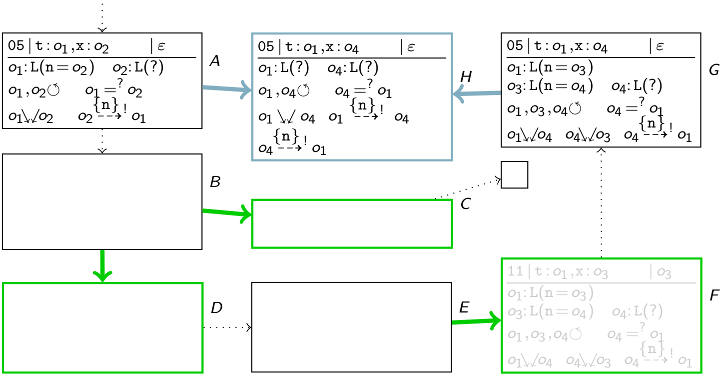
- Same program position as A $\Rightarrow$ Instantiate

  In A: $\texttt{this} = o_1 \xrightarrow{\texttt{n}} o_2 = \texttt{x}$

  In G: $\texttt{this} = o_1 \xrightarrow{\texttt{n}} o_3 \xrightarrow{\texttt{n}} o_4 = \texttt{x}$

```
void iterate() {
  L x = this.n;
  while (x != this)
    x = x.n; }
```

```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return
```



States G, H:

- Same program position as $A \Rightarrow$ Instantiate

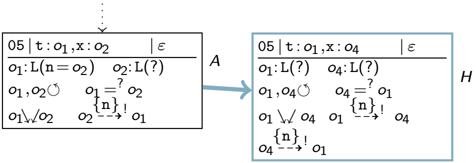
  In A: $\texttt{this} = o_1 \xrightarrow{\texttt{n}} o_2 = \texttt{x}$

  In G: $\texttt{this} = o_1 \xrightarrow{\texttt{n}} o_3 \xrightarrow{\texttt{n}} o_4 = \texttt{x}$

$\Rightarrow$ In H: Abstract to $\texttt{this} = o_1 \overset{\{\texttt{n}\}}{\dashrightarrow}! \; o_4 = \texttt{x}$

```
void iterate() {
  L x = this.n;
  while (x != this)
    x = x.n; }
```

```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return
```



States G, H:

- Same program position as $A \Rightarrow$ Instantiate

  In A: $\texttt{this} = o_1 \stackrel{\texttt{n}}{\rightarrow} o_2 = \texttt{x}$

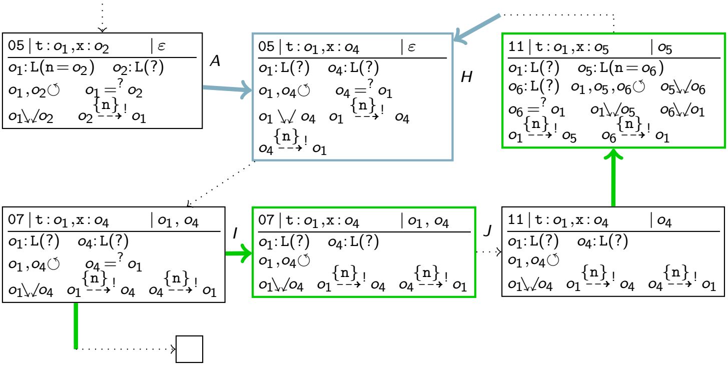
  In G: $\texttt{this} = o_1 \stackrel{\texttt{n}}{\rightarrow} o_3 \stackrel{\texttt{n}}{\rightarrow} o_4 = \texttt{x}$

$\Rightarrow$ In H: Abstract to $\texttt{this} = o_1 \stackrel{\{\texttt{n}\}}{\dashrightarrow}{}^{!}\ o_4 = \texttt{x}$

- Restart construction from more general state

```
void iterate() {
  L x = this.n;
  while (x != this)
    x = x.n; }
```

```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return
```



States G, H:

- Same program position as $A \Rightarrow$ Instantiate

  In A: $\mathtt{this} = o_1 \xrightarrow{\mathtt{n}} o_2 = \mathtt{x}$

  In G: $\mathtt{this} = o_1 \xrightarrow{\mathtt{n}} o_3 \xrightarrow{\mathtt{n}} o_4 = \mathtt{x}$

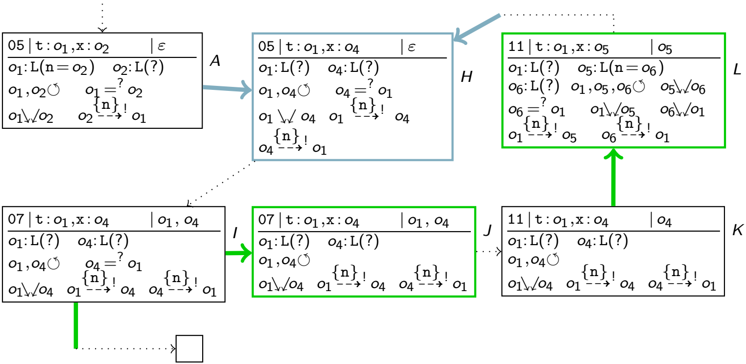
$\Rightarrow$ In H: Abstract to $\mathtt{this} = o_1 \overset{\{\mathtt{n}\}}{\dashrightarrow}{}^{!}\, o_4 = \mathtt{x}$

- Restart construction from more general state

States I, J, K, L: As before

```
void iterate() {
  L x = this.n;
  while (x != this)
    x = x.n; }
```

```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return
```

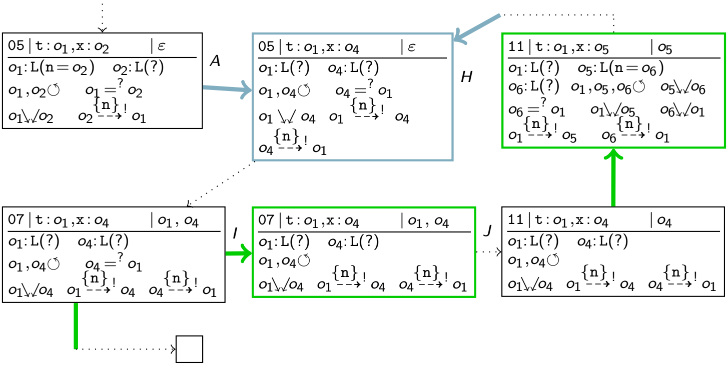


Proving termination with $R = o \overset{F}{\dashrightarrow}{}^{!}\, o'$:

1. Associate length ℓ_R with each R

```
void iterate() {
  L x = this.n;
  while (x != this)
    x = x.n; }
```

```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return
```

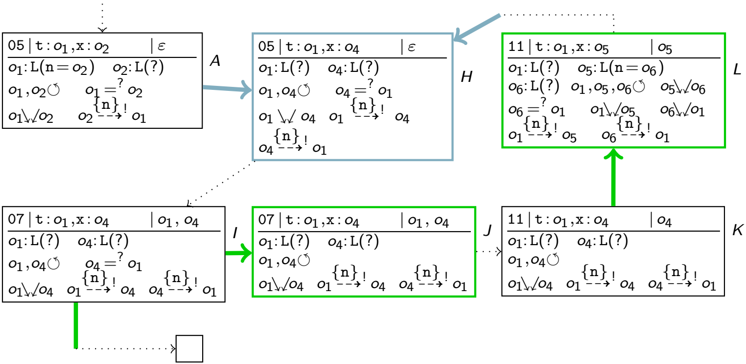


Proving termination with $R = o \overset{F}{\dashrightarrow}^{!} o'$:

1. Associate length ℓ_R with each R
2. In refinements:
   - $o =^{?} o'$ resolved to $o \neq o'$: $\ell_R > 0$

```
void iterate() {
  L x = this.n;
  while (x != this)
    x = x.n; }
```

```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return
```

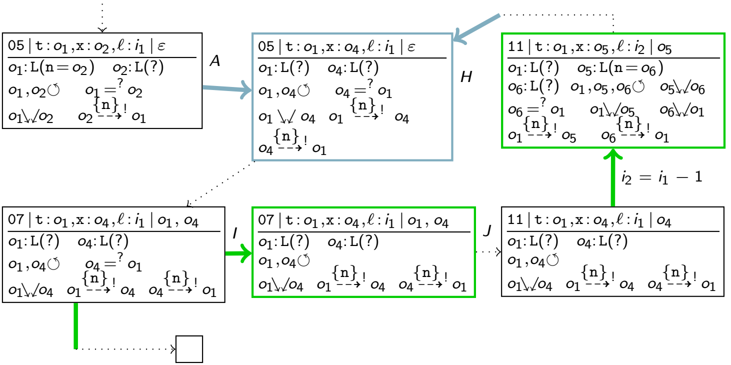


Proving termination with $R = o \overset{F}{\dashrightarrow}! \; o'$:

1. Associate length ℓ_R with each R
2. In refinements:
   - $o =^? o'$ resolved to $o \neq o'$: $\ell_R > 0$
   - $\tilde{o}$ new F-child of o: $\ell_{R'} = \ell_R - 1$ (for $R' = \tilde{o} \overset{F}{\dashrightarrow}! \; o'$)

```
void iterate() {
  L x = this.n;
  while (x != this)
    x = x.n; }
```

```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return
```

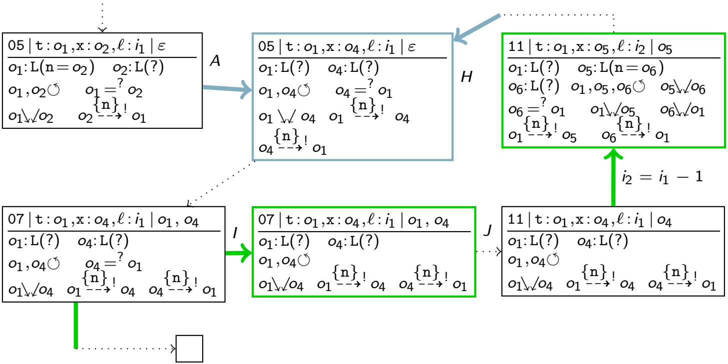


Proving termination with $R = o \overset{F}{\dashrightarrow}{}^{!}\ o'$:

1. Associate length ℓ_R with each R
2. In refinements:
   - $o =^? o'$ resolved to $o \neq o'$: $\ell_R > 0$
   - $\tilde{o}$ new F-child of o: $\ell_{R'} = \ell_R - 1$ (for $R' = \tilde{o} \overset{F}{\dashrightarrow}{}^{!}\ o'$)
3. Add variable for lengths to graphs (here: only done for $\ell_{o_4 \overset{\{n\}}{\dashrightarrow}{}^{!} o_1}$)

```
void iterate() {
  L x = this.n;
  while (x != this)
    x = x.n; }
```

```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return
```



Proving termination with $R = o \overset{F}{\dashrightarrow}! \; o'$:

1. Associate length ℓ_R with each R
2. In refinements:
   - $o =^? o'$ resolved to $o \neq o'$: $\ell_R > 0$
   - $\tilde{o}$ new F-child of o: $\ell_{R'} = \ell_R - 1$ (for $R' = \tilde{o} \overset{F}{\dashrightarrow}! \; o'$)
3. Add variable for lengths to graphs (here: only done for $\ell_{o_4 \overset{\{n\}}{\dashrightarrow}! o_1}$)

```
void iterate() {
  L x = this.n;
  while (x != this)
    x = x.n; }
```

Resulting TRS:

$$f(\ldots, \ell_{o_4 \overset{\{n\}}{\dashrightarrow}! o_1}) \to f(\ldots, \ell_{o_4 \overset{\{n\}}{\dashrightarrow}! o_1} - 1) \quad | \; \ell_{o_4 \overset{\{n\}}{\dashrightarrow}! o_1} > 0$$

Automated Termination Proofs for Java Bytecode with Cyclic Data

- Implemented in AProVE for full single-threaded Java

Automated Termination Proofs for Java Bytecode with Cyclic Data

- Implemented in AProVE for full single-threaded JAVA
- Evaluated on collection of 387 programs (including the *Termination Problem Data Base*):

	Y	N	F	T	R	Y	N	F	T	R
AProVE	267	81	11	28	9.5	51	0	6	3	15.8
AProVE '11	225	81	45	36	11.4	23	0	29	8	18.3
Julia	191	22	174	0	4.7	32	0	28	0	8.2
COSTA	160	0	181	46	11.0	29	0	5	26	30.4

all examples (first Y–R group) LinkedList + HashMap (second Y–R group)

Automated Termination Proofs for Java Bytecode with Cyclic Data

- Implemented in AProVE for full single-threaded JAVA
- Evaluated on collection of 387 programs (including the *Termination Problem Data Base*):

	all examples					`LinkedList` + `HashMap`				
	Y	**N**	**F**	**T**	**R**	**Y**	**N**	**F**	**T**	**R**
AProVE	267	81	11	28	9.5	51	0	6	3	15.8
AProVE '11	225	81	45	36	11.4	23	0	29	8	18.3
Julia	191	22	174	0	4.7	32	0	28	0	8.2
COSTA	160	0	181	46	11.0	29	0	5	26	30.4

- Termination depending on cyclic data requires early abstraction