Automated Termination Proofs for Java Programs with Cyclic Data

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

LuFG Informatik 2, RWTH Aachen University, Germany

CAV 2012

Automated Termination Analysis for Imperative Programs

Automated Termination Analysis for Imperative Programs

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

Automated Termination Analysis for Imperative Programs

- Synthesis of Linear Ranking Functions
  (Colon & Sipma, '01), (Podelski & Rybalchenko, '04), ...
- Termination Analysis for C (via Transition Invariants)
  Terminator – (*Cook, Podelski, Rybalchenko et al.*, since '05)
  CProver – (*Kroening, Sharygina, Tsitovich, Wintersteiger*, since '10)

Automated Termination Analysis for Imperative Programs

- Synthesis of Linear Ranking Functions
  (Colon & Sipma, '01), (Podelski & Rybalchenko, '04), ...
- Termination Analysis for C (via Transition Invariants)
  Terminator – (*Cook, Podelski, Rybalchenko et al.*, since '05)
  CProver – (*Kroening, Sharygina, Tsitovich, Wintersteiger*, since '10)
- Termination Analysis for C (via Polynomial Orders)
  KITTeL – (*Falke, Kapur, Sinz*, since '11)

Automated Termination Analysis for Imperative Programs

- Synthesis of Linear Ranking Functions
  (Colon & Sipma, '01), (Podelski & Rybalchenko, '04), ...
- Termination Analysis for C (via Transition Invariants)
  Terminator – (*Cook, Podelski, Rybalchenko et al.*, since '05)
  CProver – (*Kroening, Sharygina, Tsitovich, Wintersteiger*, since '10)
- Termination Analysis for C (via Polynomial Orders)
  KITTeL – (*Falke, Kapur, Sinz*, since '11)
- Termination Analysis for Java (via Path Length, CLP backend)
  Julia – (*Spoto, Mesnard, Payet*, since '08)
  COSTA – (*Albert, Arenas, Codish, Genaim, Puebla, Zanardini*, since '08)

Automated Termination Analysis for Imperative Programs

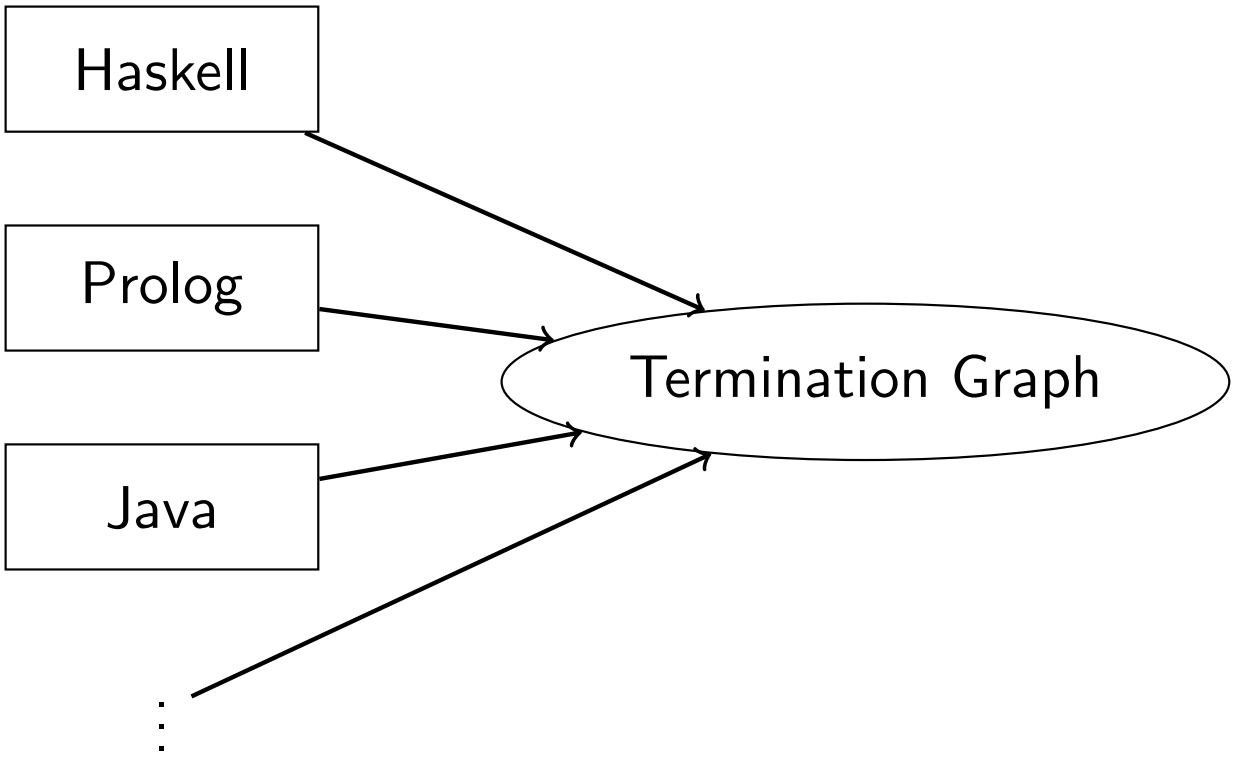
- Synthesis of Linear Ranking Functions
  (Colon & Sipma, '01), (Podelski & Rybalchenko, '04), ...
- Termination Analysis for C (via Transition Invariants)
  Terminator – (*Cook, Podelski, Rybalchenko et al.*, since '05)
  CProver – (*Kroening, Sharygina, Tsitovich, Wintersteiger*, since '10)
- Termination Analysis for C (via Polynomial Orders)
  KITTeL – (*Falke, Kapur, Sinz*, since '11)
- Termination Analysis for Java (via Path Length, CLP backend)
  Julia – (*Spoto, Mesnard, Payet*, since '08)
  COSTA – (*Albert, Arenas, Codish, Genaim, Puebla, Zanardini*, since '08)
- ...

Rewriting-backed approach: Idea

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

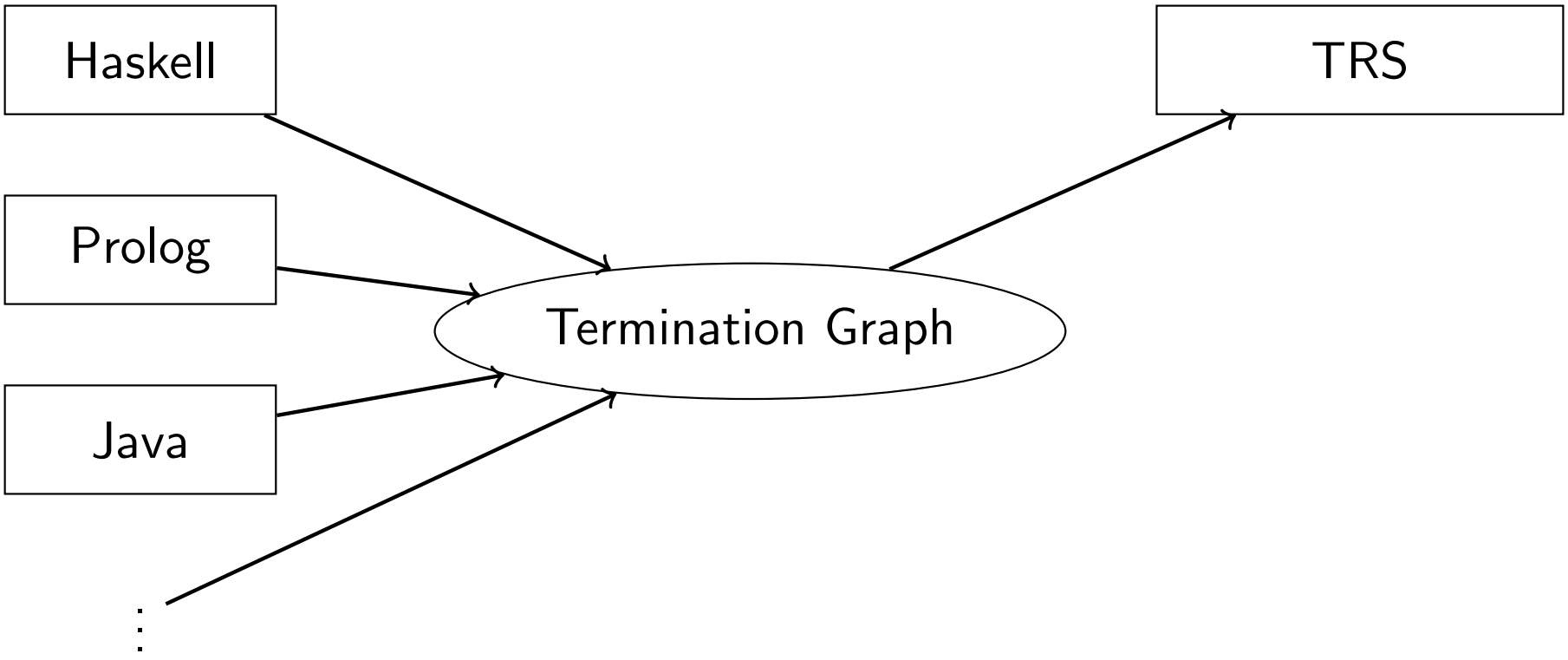
Rewriting-backed approach: Idea

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



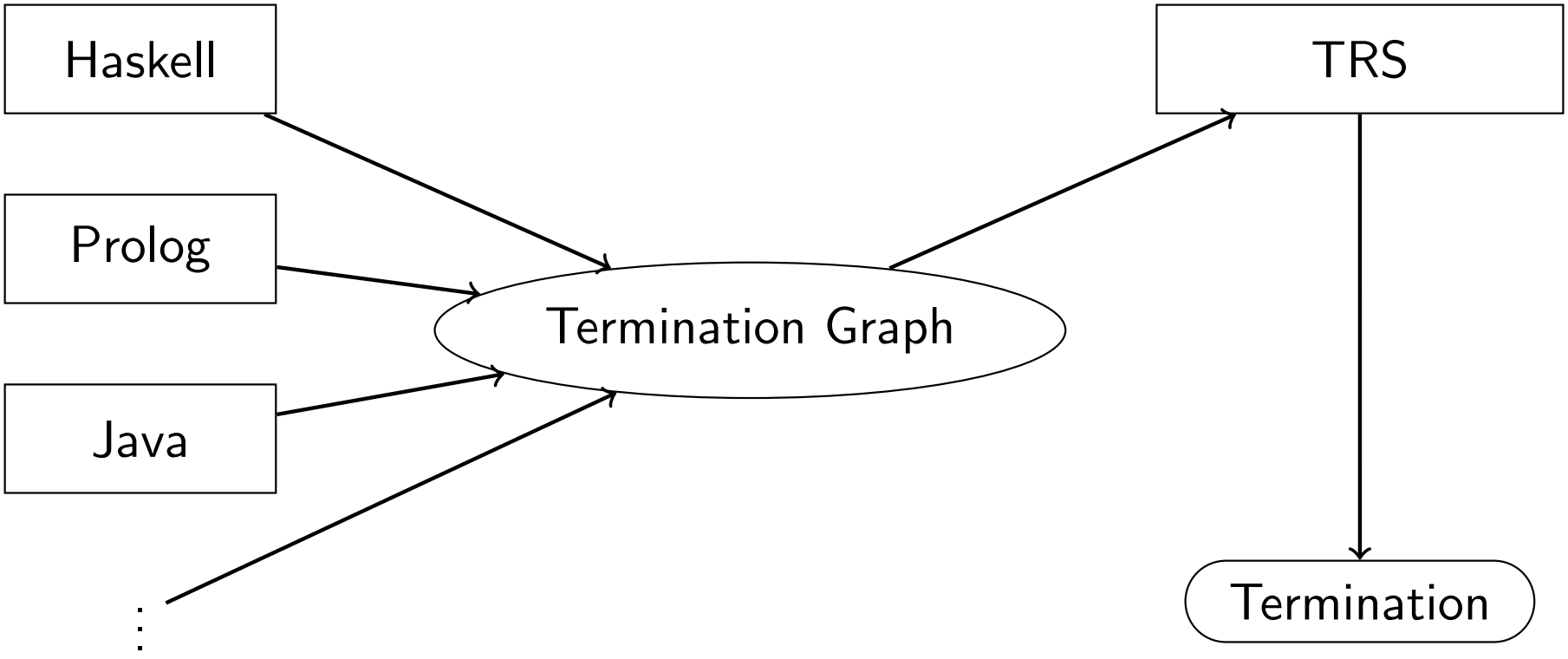
Rewriting-backed approach: Idea

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

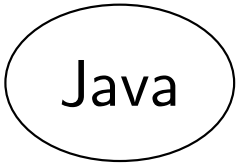


Rewriting-backed approach: Idea

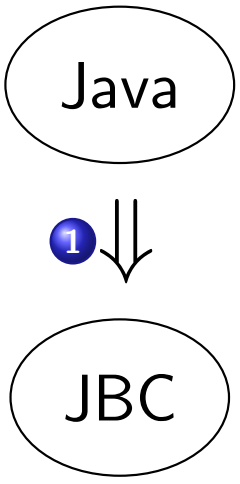
- Programming languages *hard* $\curvearrowright$ Simpler representation needed
- Termination Graphs: Simpler, contain all information
- Term Rewrite Systems (TRSs) generated from Termination Graph
- Prove TRS termination using existing provers



Rewriting-backed approach: Structure

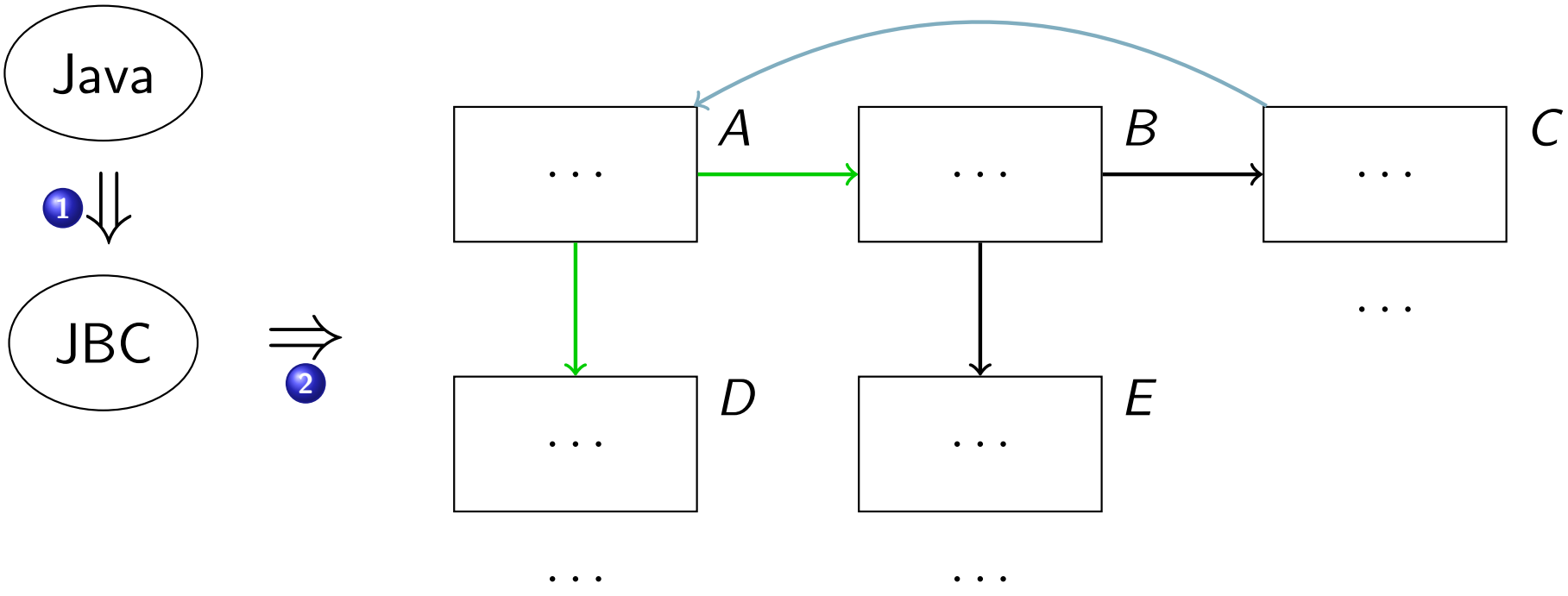


Rewriting-backed approach: Structure



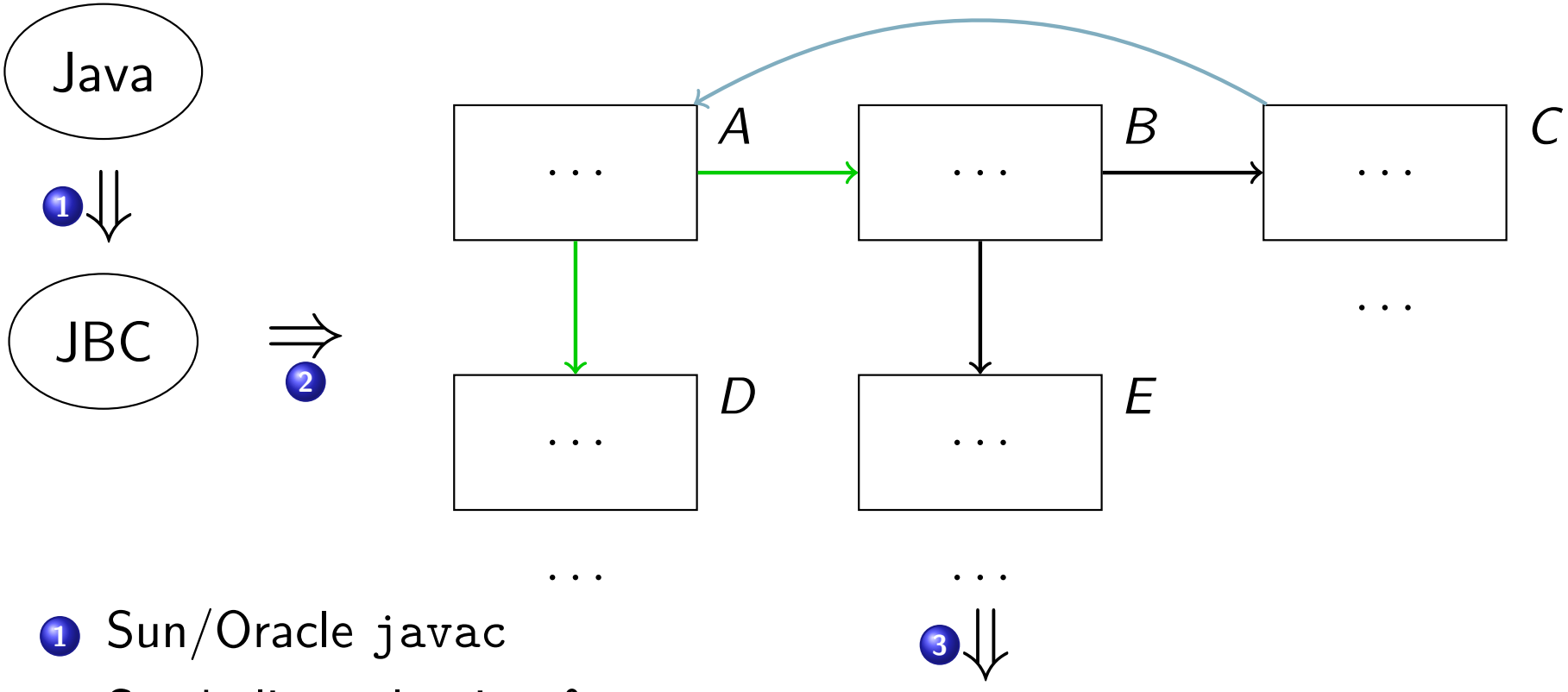
1. Sun/Oracle `javac`

Rewriting-backed approach: Structure



1. Sun/Oracle `javac`
2. Symbolic evaluation & Abstraction

Rewriting-backed approach: Structure



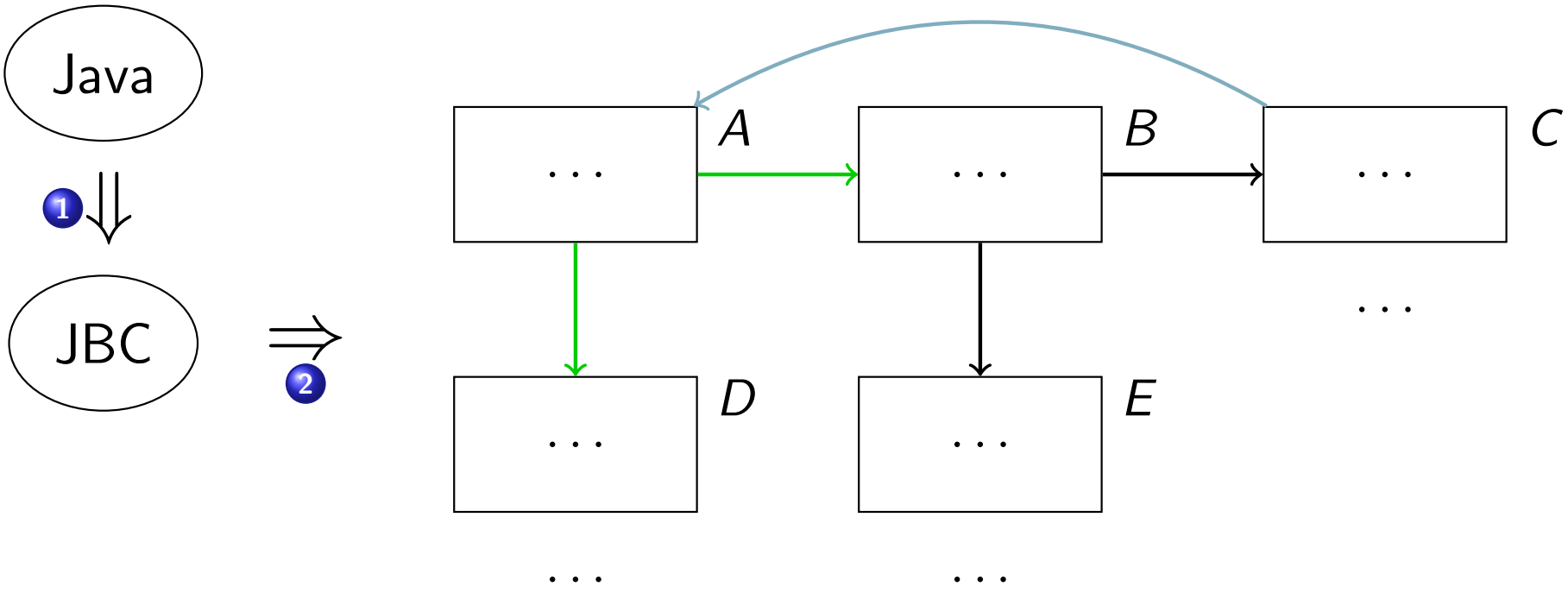
1. Sun/Oracle `javac`
2. Symbolic evaluation & Abstraction
3. Post-processing & Translation to TRS

$$f_A(\ldots) \longrightarrow f_B(\ldots) \qquad f_B(\ldots) \longrightarrow f_C(\ldots)$$

$$f_A(\ldots) \longrightarrow f_D(\ldots) \qquad f_B(\ldots) \longrightarrow f_E(\ldots)$$

$$f_C(\ldots) \longrightarrow f_A(\ldots)$$

Rewriting-backed approach: Structure



1. Sun/Oracle `javac`
2. Symbolic evaluation & Abstraction
3. Post-processing & Translation to TRS
4. Standard rewriting techniques

3

$f_A(\ldots) \longrightarrow f_B(\ldots)$ $\quad f_B(\ldots) \longrightarrow f_C(\ldots)$

$f_A(\ldots) \longrightarrow f_D(\ldots)$ $\quad f_B(\ldots) \longrightarrow f_E(\ldots)$

4 $\quad f_C(\ldots) \longrightarrow f_A(\ldots)$

Terminates

Rewriting-backed approach: Advantages

Handling of user-defined data structures:

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

Rewriting-backed 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

Rewriting-backed 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-backed 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 well-founded orders automatically

$\Rightarrow$ Complex orders for user-defined data structures possible

Rewriting-backed approach: Challenges

Handling of user-defined cyclic data structures:

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

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

Rewriting-backed 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-backed 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

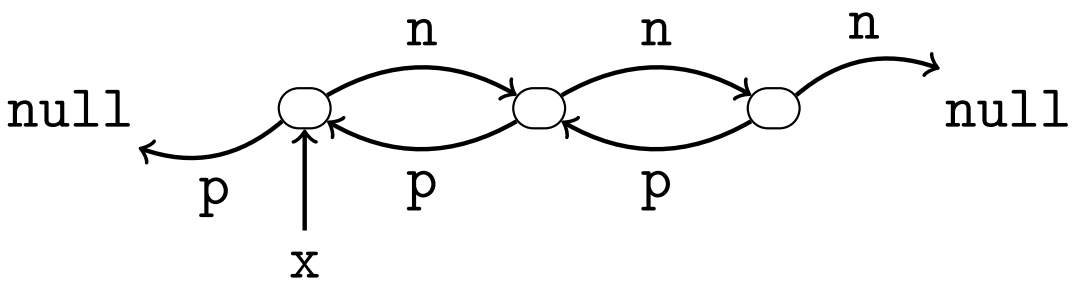
Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant

Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant:

```
class L1 {
  L1 p, n;
  static int length(L1 x) {
    int r = 1;
    while (x != null) {
      r++;
      x = x.n;
    }
    return r; }}
```

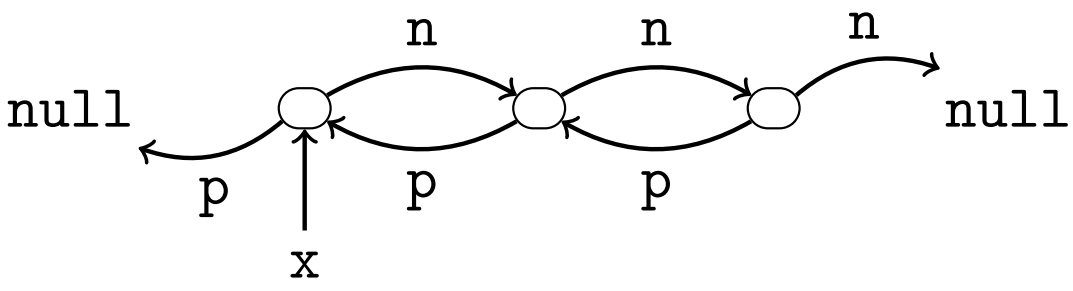


1. Prepare result `r`
2. Continue if `x` not `null`
3. Increment `r`
4. Go to next element

Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant:

```
class L1 {
  L1 p, n;
  static int length(L1 x) {
    int r = 1;
    while (x != null) {
      r++;
      x = x.n;
    }
    return r; }}
```

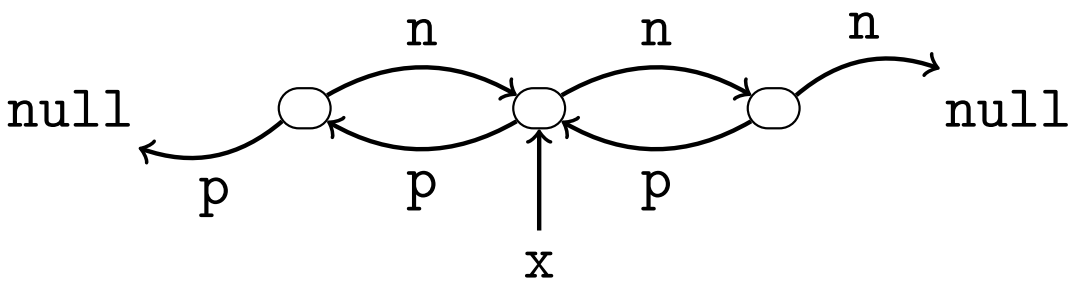


1. Prepare result `r`
2. Continue if `x` not `null`
3. Increment `r`
4. Go to next element

Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant:

```
class L1 {
  L1 p, n;
  static int length(L1 x) {
    int r = 1;
    while (x != null) {
      r++;
      x = x.n;
    }
    return r; }}
```

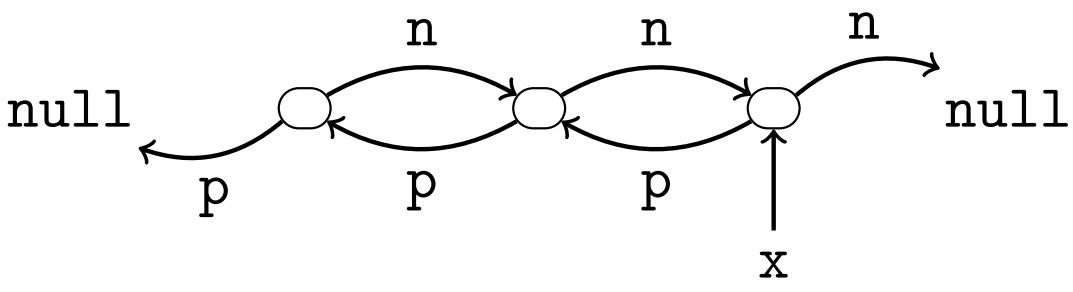


1. Prepare result `r`
2. Continue if `x` not `null`
3. Increment `r`
4. Go to next element

Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant:

```
class L1 {
  L1 p, n;
  static int length(L1 x) {
    int r = 1;
    while (x != null) {
      r++;
      x = x.n;
    }
    return r; }}
```

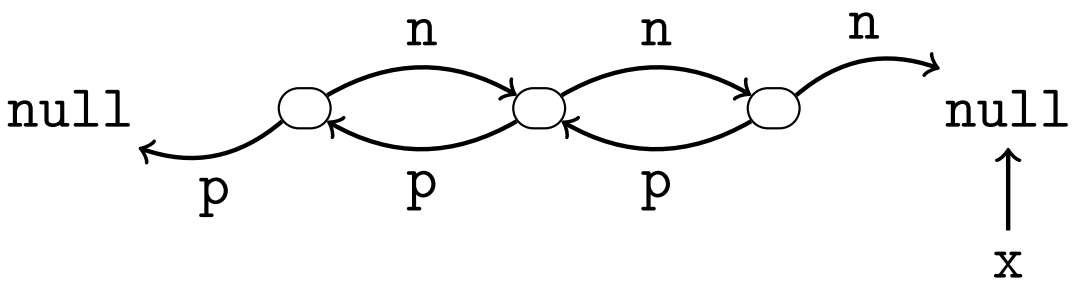


1. Prepare result `r`
2. Continue if `x` not `null`
3. Increment `r`
4. Go to next element

Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant:

```
class L1 {
  L1 p, n;
  static int length(L1 x) {
    int r = 1;
    while (x != null) {
      r++;
      x = x.n;
    }
    return r; }}
```



1. Prepare result `r`
2. Continue if `x` not `null`
3. Increment `r`
4. Go to next element

Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant

ii) Marking algorithms

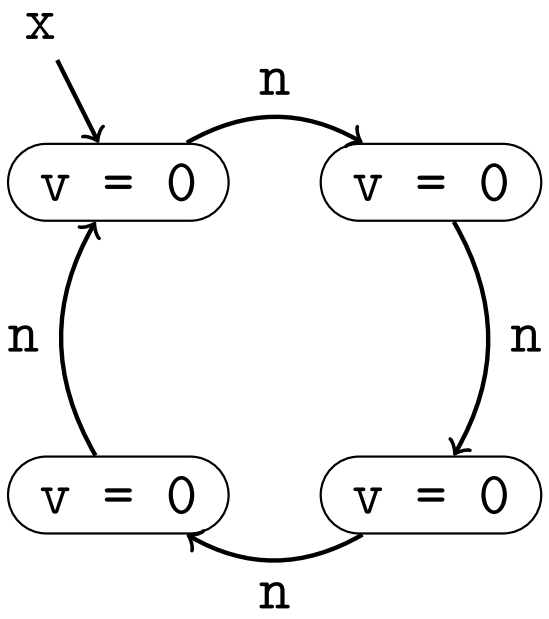
Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant

ii) Marking algorithms:

```
class L2 {
  int v;    L2 n;
  static void visit(L2 x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

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



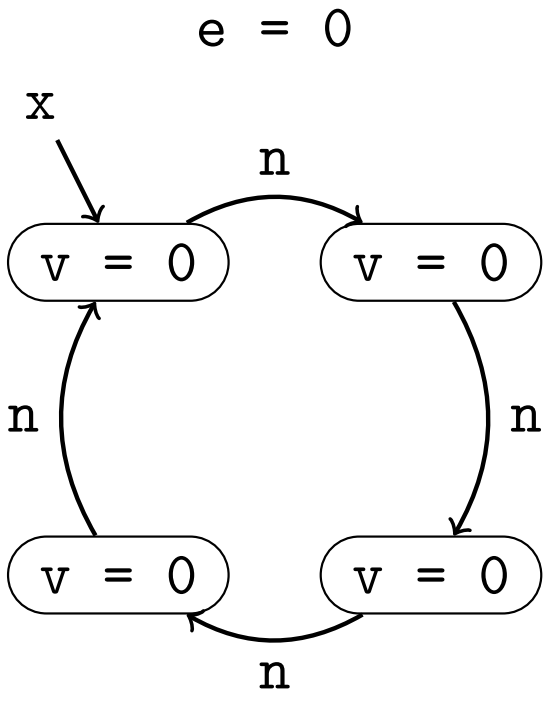
Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant

ii) Marking algorithms:

```
class L2 {
  int v;    L2 n;
  static void visit(L2 x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

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



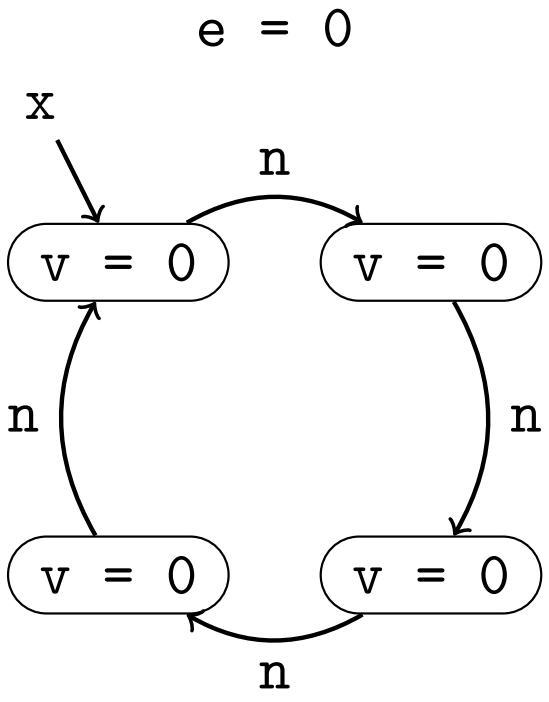
Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant

ii) Marking algorithms:

```
class L2 {
  int v;    L2 n;
  static void visit(L2 x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

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



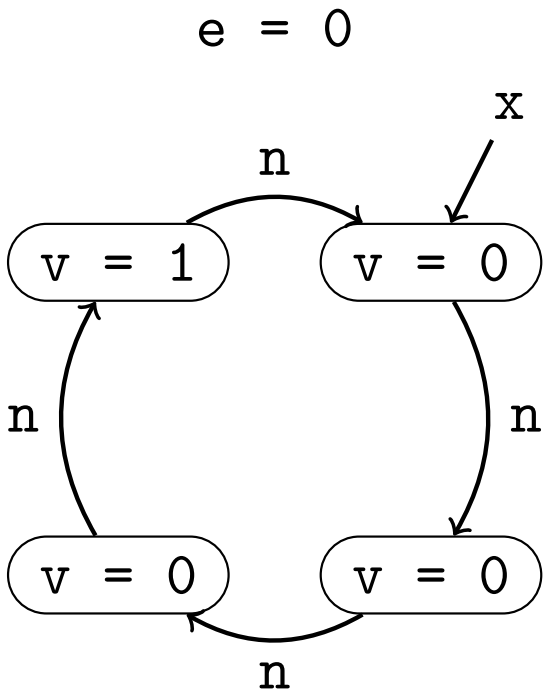
Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant

ii) Marking algorithms:

```
class L2 {
  int v;    L2 n;
  static void visit(L2 x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

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



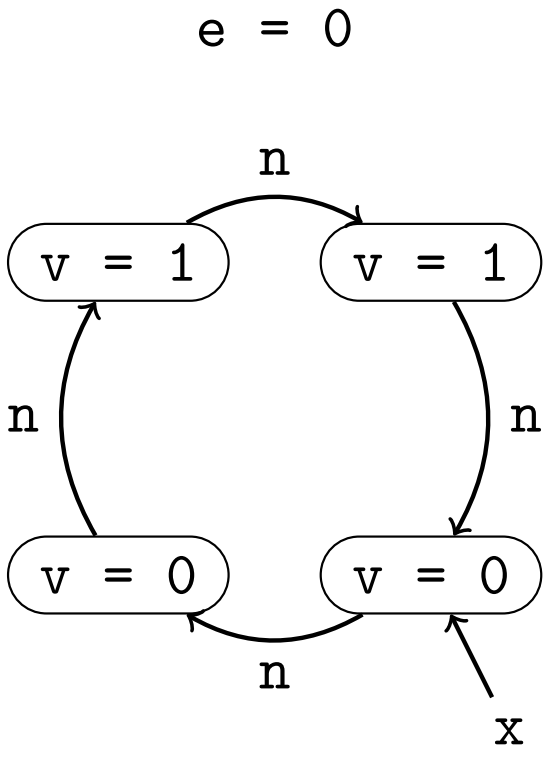
Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant

ii) Marking algorithms:

```
class L2 {
  int v;    L2 n;
  static void visit(L2 x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

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



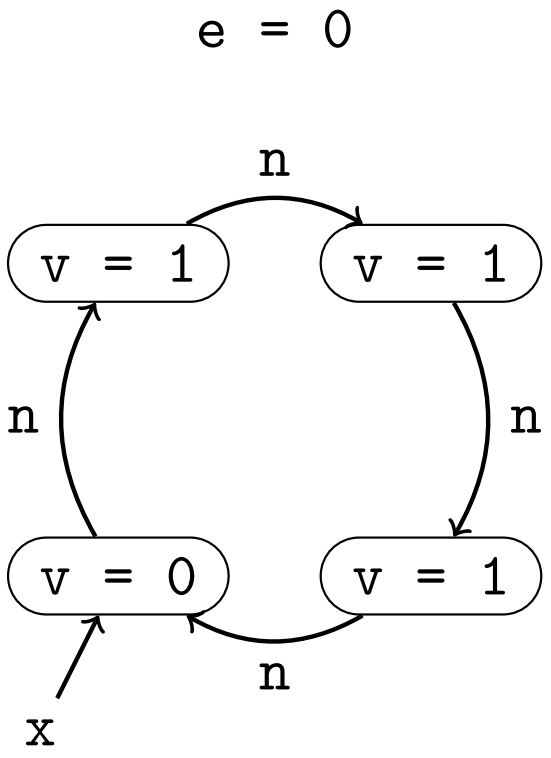
Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant

ii) Marking algorithms:

```
class L2 {
  int v;    L2 n;
  static void visit(L2 x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

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



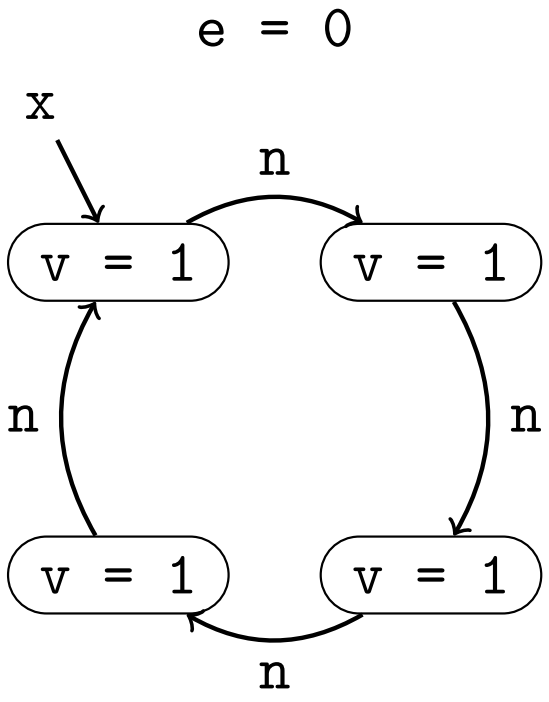
Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant

ii) Marking algorithms:

```
class L2 {
  int v;    L2 n;
  static void visit(L2 x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

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



Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant
ii) Marking algorithms
iii) Definite cyclicity used (via sentinel element)

Handled Classes of Algorithms on Cyclic Data

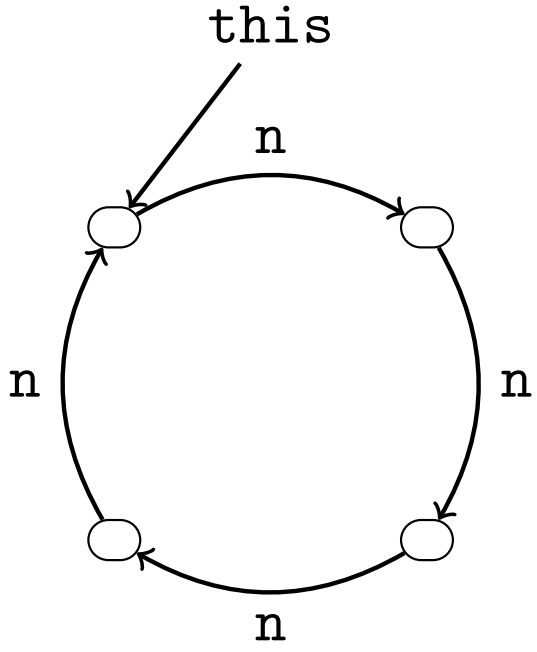
i) Cyclicity of data irrelevant

ii) Marking algorithms

iii) Definite cyclicity used (via sentinel element):

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

1. Keep first element in `this`
2. Continue if not yet reached
3. Go to next element



Handled Classes of Algorithms on Cyclic Data

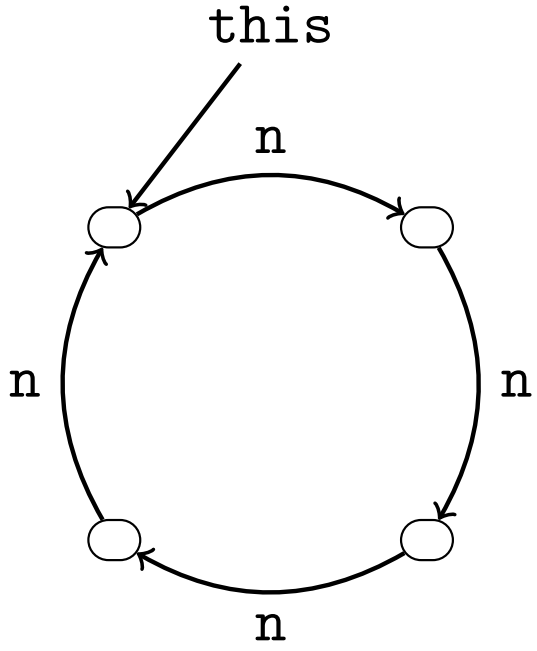
i) Cyclicity of data irrelevant

ii) Marking algorithms

iii) Definite cyclicity used (via sentinel element):

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

1. Keep first element in `this`
2. Continue if not yet reached
3. Go to next element



Handled Classes of Algorithms on Cyclic Data

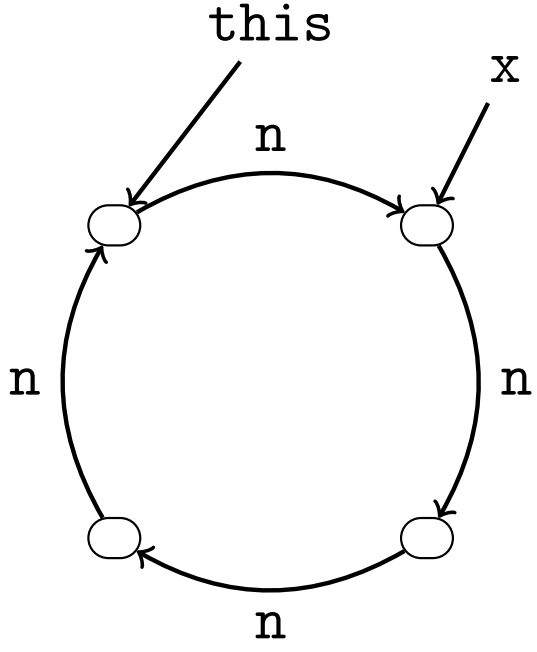
i) Cyclicity of data irrelevant

ii) Marking algorithms

iii) Definite cyclicity used (via sentinel element):

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

1. Keep first element in `this`
2. Continue if not yet reached
3. Go to next element



Handled Classes of Algorithms on Cyclic Data

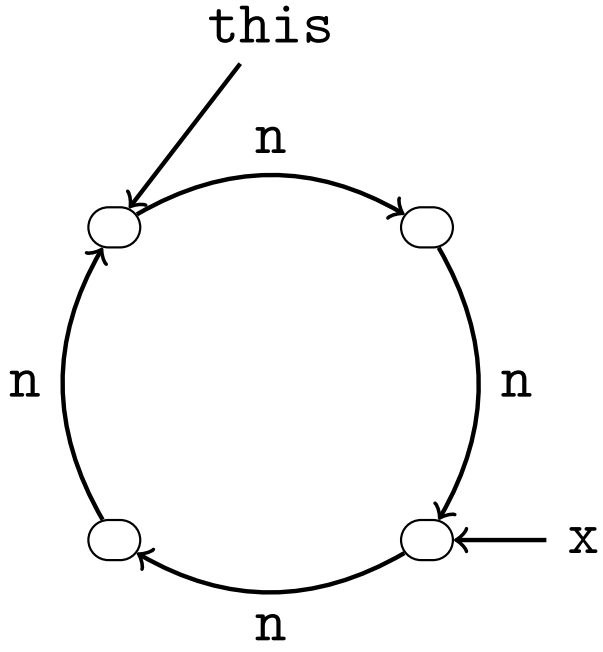
i) Cyclicity of data irrelevant

ii) Marking algorithms

iii) Definite cyclicity used (via sentinel element):

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

1. Keep first element in `this`
2. Continue if not yet reached
3. Go to next element



Handled Classes of Algorithms on Cyclic Data

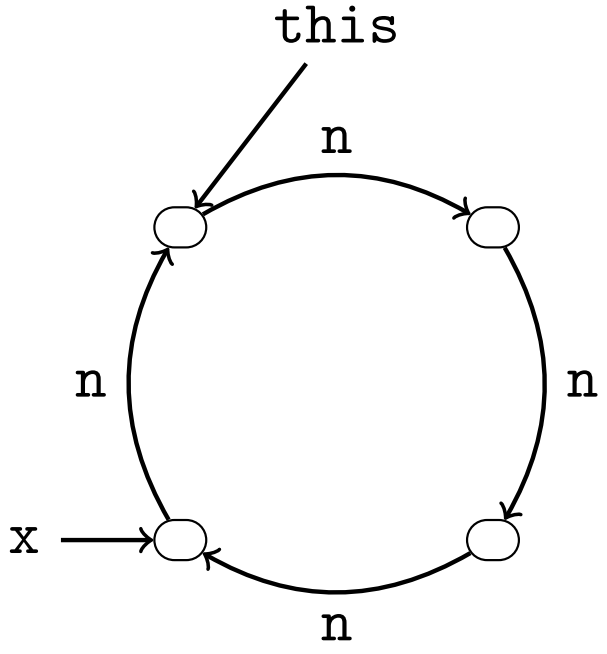
i) Cyclicity of data irrelevant

ii) Marking algorithms

iii) Definite cyclicity used (via sentinel element):

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

1. Keep first element in `this`
2. Continue if not yet reached
3. Go to next element



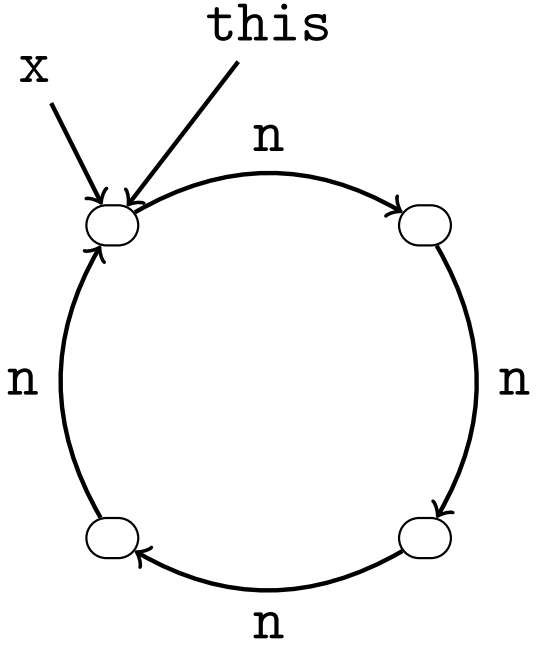
Handled Classes of Algorithms on Cyclic Data

i) Cyclicity of data irrelevant

ii) Marking algorithms

iii) Definite cyclicity used (via sentinel element):

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



1. Keep first element in `this`
2. Continue if not yet reached
3. Go to next element

iterate: the example

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

iterate: the example

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

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

Abstract Java virtual machine states

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

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

$$\frac{05 \mid \mathtt{t}:o_1,\mathtt{x}:o_2 \mid \varepsilon}{\begin{array}{ll} o_1:\mathrm{L}(\mathtt{n}=o_2) & o_2:\mathrm{L}(?) \\ o_1,o_2\circlearrowleft & o_1 =^{?} o_2 \\ o_1 \diagdown\!\!\diagup o_2 & o_2 \xrightarrow{\{\mathtt{n}\}}{}^{!} o_1 \end{array}}$$

Abstract Java virtual machine states

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

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

Stack frame:

- Next instruction

$$\frac{05 \mid \mathtt{t}:o_1,\mathtt{x}:o_2 \mid \varepsilon}{\begin{array}{ll} o_1:\mathrm{L}(\mathtt{n}=o_2) & o_2:\mathrm{L}(?) \\ o_1,o_2\circlearrowleft & o_1 =^{?} o_2 \\ o_1 \not\perp o_2 & o_2 \xrightarrow{\{\mathtt{n}\}}{}^{!} o_1 \end{array}}$$

Abstract Java virtual machine states

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

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

Stack frame:

- Next instruction
- Local variables
- Operand stack

$$\frac{05 \mid t : o_1, x : o_2 \mid \varepsilon}{\begin{array}{ll} o_1 : \mathrm{L}(\mathtt{n} = o_2) & o_2 : \mathrm{L}(?) \\ o_1, o_2 \circlearrowleft & o_1 =^{?} o_2 \\ o_1 \diagdown\!\!\diagup o_2 & o_2 \xrightarrow{\{\mathtt{n}\}}{}^{!} o_1 \end{array}}$$

Abstract Java virtual machine states

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

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

Stack frame:

- Next instruction
- Local variables
- Operand stack

Heap information:

$$\frac{05 \mid \mathtt{t}:o_1,\mathtt{x}:o_2 \mid \varepsilon}{\begin{array}{ll} o_1:\mathrm{L}(\mathtt{n}=o_2) & o_2:\mathrm{L}(?) \\ o_1,o_2\circlearrowleft & o_1 =^{?} o_2 \\ o_1 \diagdown\!\!\diagup o_2 & o_2 \xrightarrow{\{\mathtt{n}\}}{}^{!} o_1 \end{array}}$$

Abstract Java virtual machine states

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

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

Stack frame:

- Next instruction
- Local variables
- Operand stack

$$\frac{05 \mid \mathtt{t}:o_1,\mathtt{x}:o_2 \mid \varepsilon}{\begin{array}{ll} o_1:\mathrm{L}(\mathtt{n}=o_2) & o_2:\mathrm{L}(?) \\ o_1,o_2\circlearrowleft & o_1 =^{?} o_2 \\ o_1 \diagdown\!\!\diagup o_2 & o_2 \xrightarrow{\{\mathtt{n}\}}{}^{!} o_1 \end{array}}$$

Heap information:

- At o_1 is known L object

Abstract Java virtual machine states

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

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

Stack frame:

- Next instruction
- Local variables
- Operand stack

$$\begin{array}{l|l|l} 05 & \mathtt{t}:o_1,\mathtt{x}:o_2 & \varepsilon \end{array}$$

$$\begin{array}{ll} o_1:\mathtt{L}(\mathtt{n}=o_2) & o_2:\mathtt{L}(?) \\ o_1,o_2\circlearrowleft & o_1 =^{?} o_2 \\ o_1 \diagdown\!\!\diagup o_2 & o_2 \xrightarrow{\{\mathtt{n}\}}{}^{!} o_1 \end{array}$$

Heap information:

- At o_1 is known L object
- At o_2 is unknown L object or null

Abstract Java virtual machine states

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

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

Stack frame:

- Next instruction
- Local variables
- Operand stack

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

Heap information:

- At o_1 is known L object
- At o_2 is unknown L object or null
- Integers: $i_1 : \mathbb{Z}$, $i_2 : [2, \infty)$

Abstract Java virtual machine states

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

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

Stack frame:

- Next instruction
- Local variables
- Operand stack

$$\frac{05 \mid \mathtt{t}:o_1,\mathtt{x}:o_2 \mid \varepsilon}{\begin{array}{ll} o_1:\mathtt{L}(\mathtt{n}=o_2) & o_2:\mathtt{L}(?) \\ o_1,o_2\circlearrowleft & o_1 =^{?} o_2 \\ o_1 \diagdown\!\!\diagup o_2 & o_2 \xrightarrow{\{\mathtt{n}\}}{}^{!} o_1 \end{array}}$$

Heap information:

- At o_1 is known `L` object
- At o_2 is unknown `L` object or `null`
- Integers: $i_1 : \mathbb{Z}$, $i_2 : [2, \infty)$

Only explicit sharing

Abstract Java virtual machine states

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

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

Stack frame:

- Next instruction
- Local variables
- Operand stack

$$\frac{05 \mid \mathtt{t}:o_1,\mathtt{x}:o_2 \mid \varepsilon}{o_1:\mathtt{L}(\mathtt{n}=o_2) \quad o_2:\mathtt{L}(?) \\ o_1,o_2\circlearrowleft \quad o_1 =^{?} o_2 \\ o_1 \backslash\!/ o_2 \quad o_2 \xrightarrow{\{\mathtt{n}\}}{}^{!} o_1}$$

Heap information:

- At o_1 is known `L` object
- At o_2 is unknown `L` object or `null`
- Integers: $i_1 : \mathbb{Z}$, $i_2 : [2, \infty)$

Heap predicates: **Only explicit sharing**

- References o_1, o_2 may be cyclic

Abstract Java virtual machine states

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

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

Stack frame:

- Next instruction
- Local variables
- Operand stack

$$\frac{05 \mid \mathtt{t}:o_1,\mathtt{x}:o_2 \mid \varepsilon}{o_1{:}\mathtt{L}(\mathtt{n}=o_2) \quad o_2{:}\mathtt{L}(?) \quad o_1,o_2\circlearrowleft \quad o_1 =^{?} o_2 \quad o_1 \diagdown\!\!\diagup o_2 \quad o_2 \xrightarrow{\{\mathtt{n}\}}{}^{!} o_1}$$

Heap information:

- At o_1 is known L object
- At o_2 is unknown L object or null
- Integers: $i_1 : \mathbb{Z}$, $i_2 : [2, \infty)$

Heap predicates: **Only explicit sharing**

- References o_1, o_2 may be cyclic
- References o_1, o_2 may be equal

Abstract Java virtual machine states

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

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

Stack frame:

- Next instruction
- Local variables
- Operand stack

$$05 \mid \mathtt{t}:o_1,\mathtt{x}:o_2 \mid \varepsilon$$

$o_1{:}\mathtt{L}(\mathtt{n}=o_2)$ $\quad o_2{:}\mathtt{L}(?)$

$o_1, o_2 \circlearrowleft$ $\quad o_1 =^{?} o_2$

$o_1 \not\downarrow\!\!\downarrow o_2$ $\quad o_2 \xrightarrow{\{\mathtt{n}\}}{}^{!} o_1$

Heap information:

- At o_1 is known L object
- At o_2 is unknown L object or null
- Integers: $i_1 : \mathbb{Z}$, $i_2 : [2, \infty)$

Heap predicates: **Only explicit sharing**

- References o_1, o_2 may be cyclic
- References o_1, o_2 may be equal
- References o_1, o_2 may share

Abstract Java virtual machine states

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

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

Stack frame:

- Next instruction
- Local variables
- Operand stack

$$\frac{05 \mid \mathtt{t}:o_1,\mathtt{x}:o_2 \mid \varepsilon}{o_1{:}\mathtt{L}(\mathtt{n}=o_2) \quad o_2{:}\mathtt{L}(?) \quad o_1,o_2\circlearrowleft \quad o_1 =^{?} o_2 \quad o_1 \diagdown\!\!\diagup o_2 \quad o_2 \xrightarrow{\{\mathtt{n}\}}{}^{!} o_1}$$

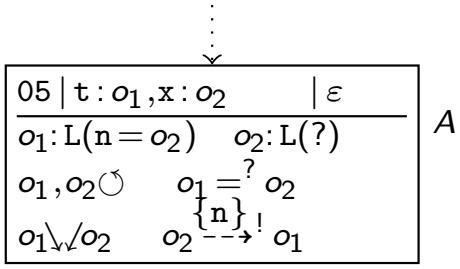
Heap information:

- At o_1 is known `L` object
- At o_2 is unknown `L` object or `null`
- Integers: $i_1 : \mathbb{Z}$, $i_2 : [2, \infty)$

Heap predicates: **Only explicit sharing**

- References o_1, o_2 may be cyclic
- References o_1, o_2 may be equal
- References o_1, o_2 may share
- Reference o_2 definitely reaches o_1 when following the field `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
```

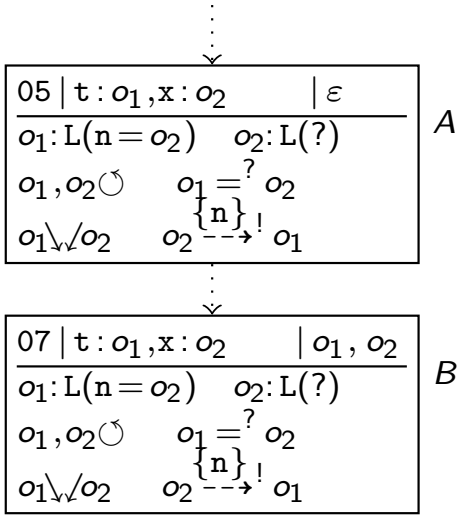


State *A*:

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

```
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*:

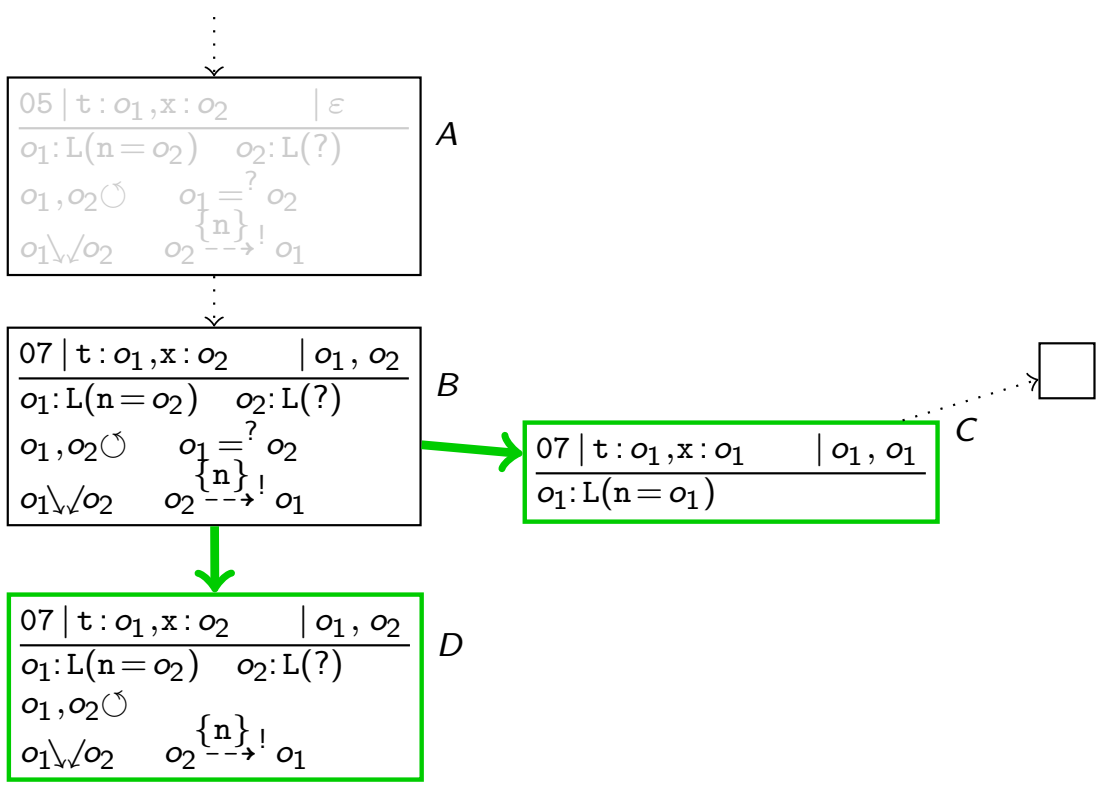
- `t` some definitely cyclic list
- `x` second element in list

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
```



State *A*:

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

State *B*:

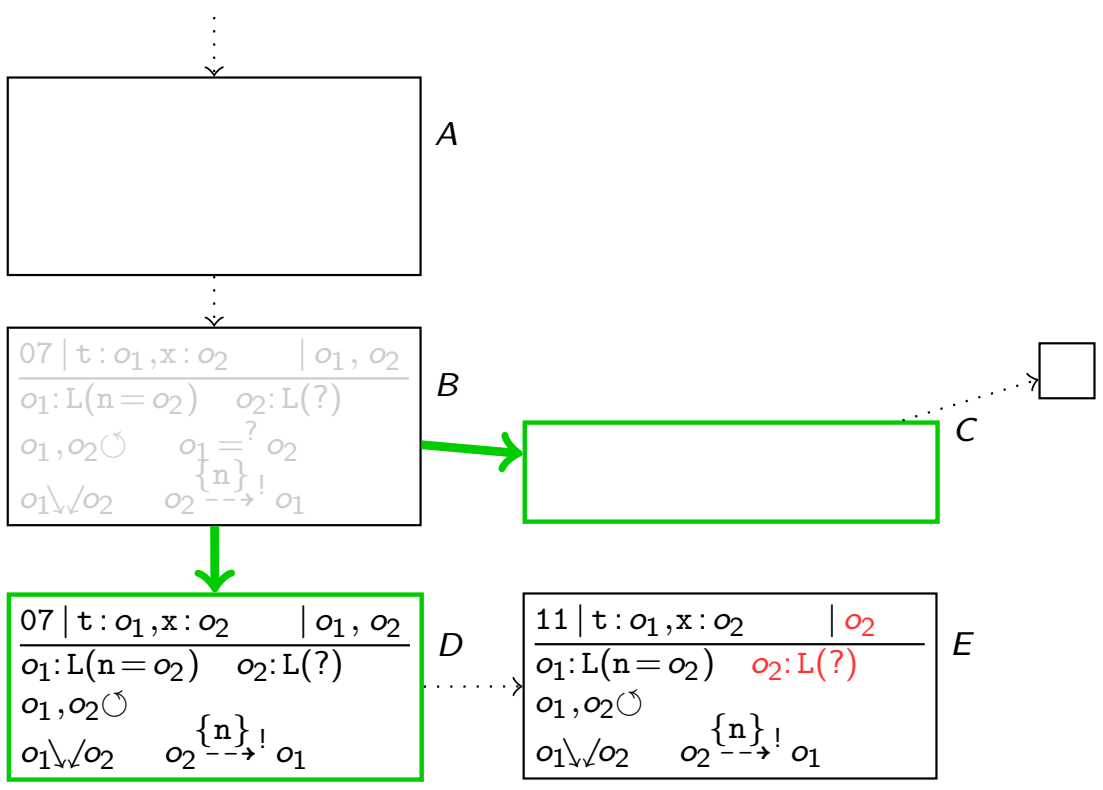
- First equals second element?

⇒ Refinement

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

```
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
```

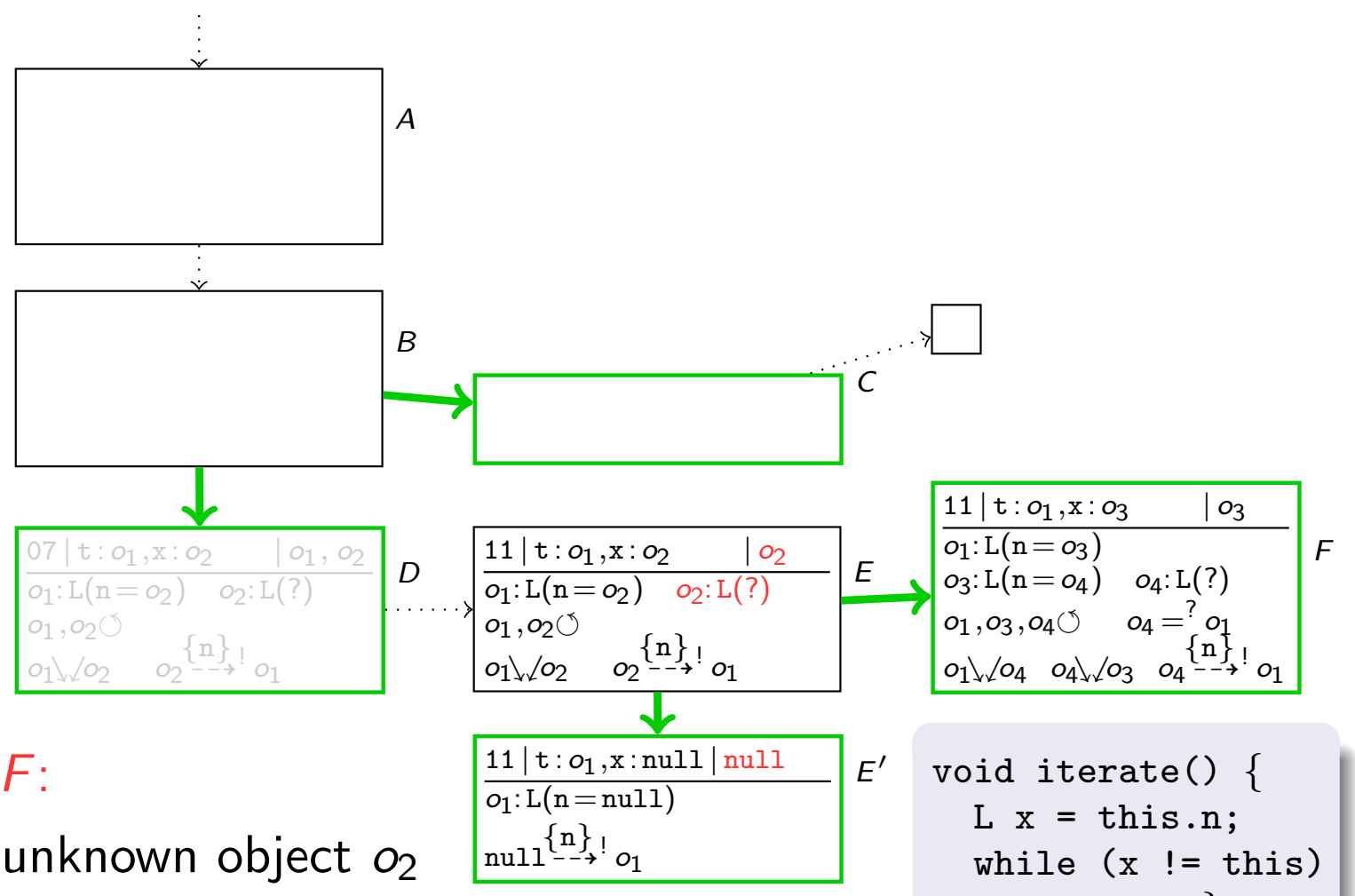


State E:

- 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
```

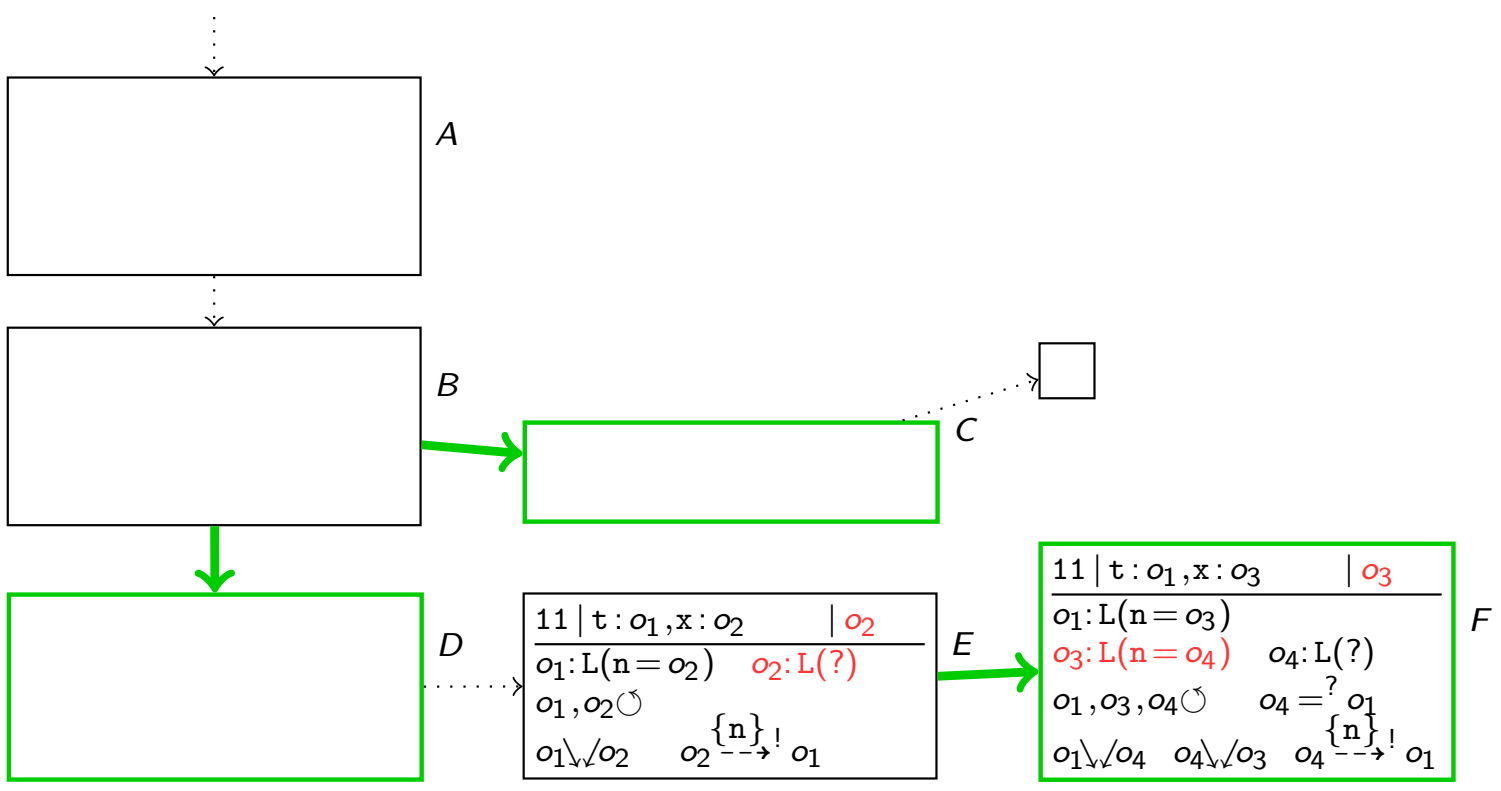


States E, F:

- Access to unknown object o_2
  $\Rightarrow$ Refinement
- In E': Case $o_2 = \texttt{null}$ not possible (implies o_2 not reaching 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
```



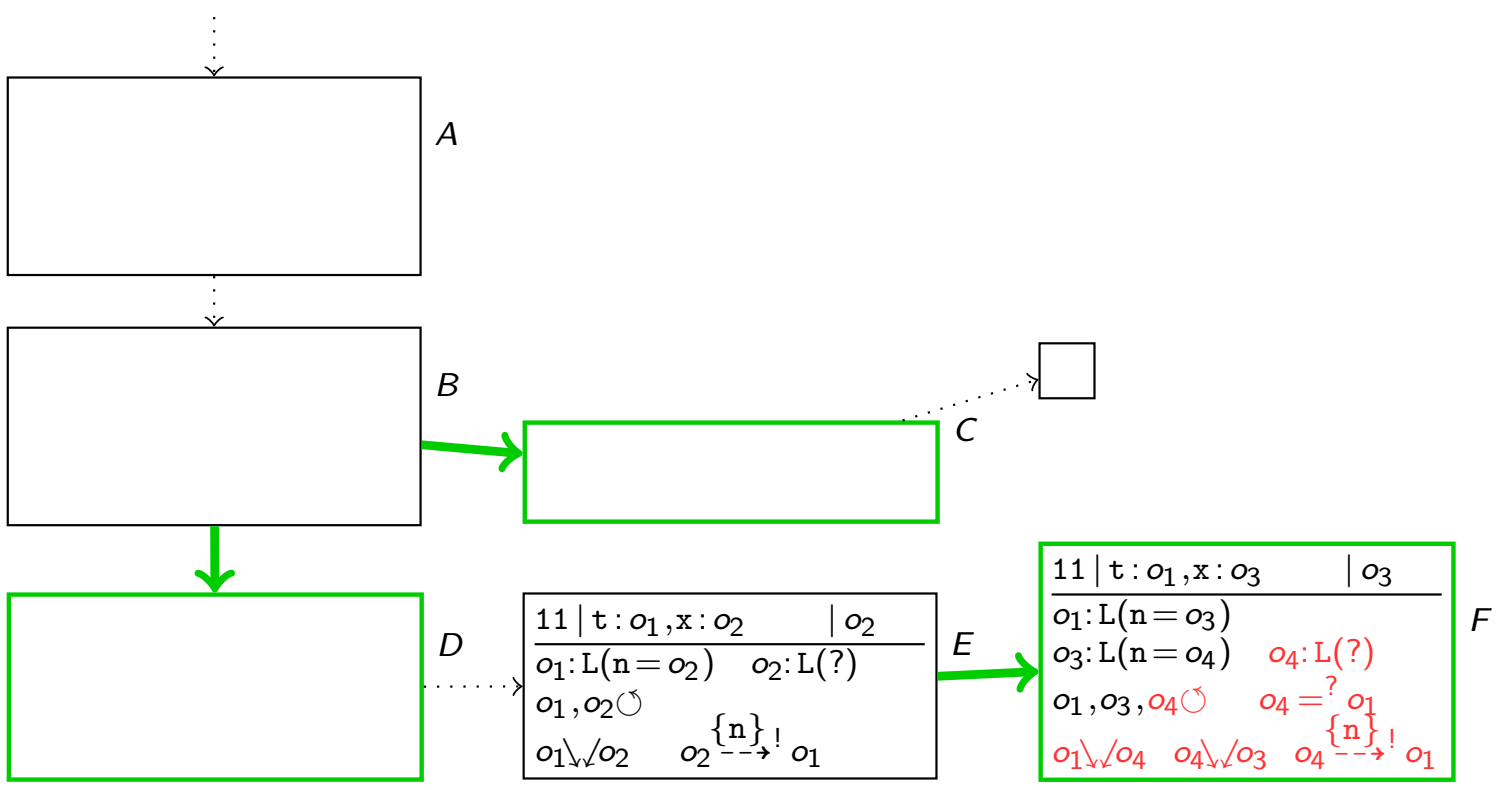
States E, F:

- Access to unknown object o_2
  $\Rightarrow$ Refinement

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

- In E': Case $o_2 = \texttt{null}$ not possible (implies o_2 not reaching o_1)
- In F: o_2 renamed to o_3, pointing to L-object with successor o_4:

```
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
```



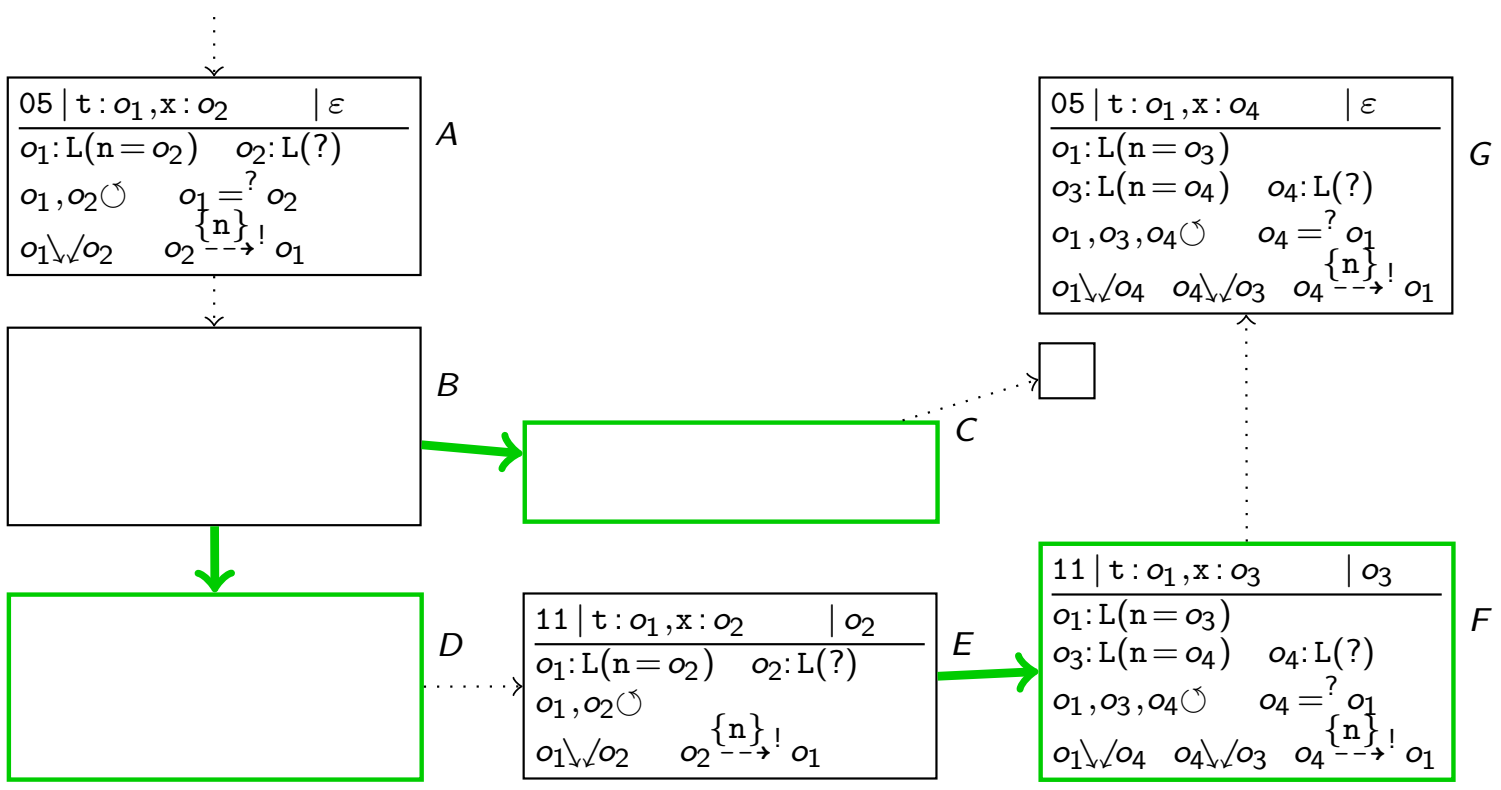
States E, F:

- Access to unknown object o_2
  $\Rightarrow$ Refinement

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

- In E': Case $o_2 = \texttt{null}$ not possible (implies o_2 not reaching o_1)
- In F: o_2 renamed to o_3, pointing to L-object with successor o_4:
  - o_4 possibly cyclic
  - o_4 possibly equal to o_1 and may reach o_1, o_3
  - o_4 definitely reaches o_1 via field 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; }
```

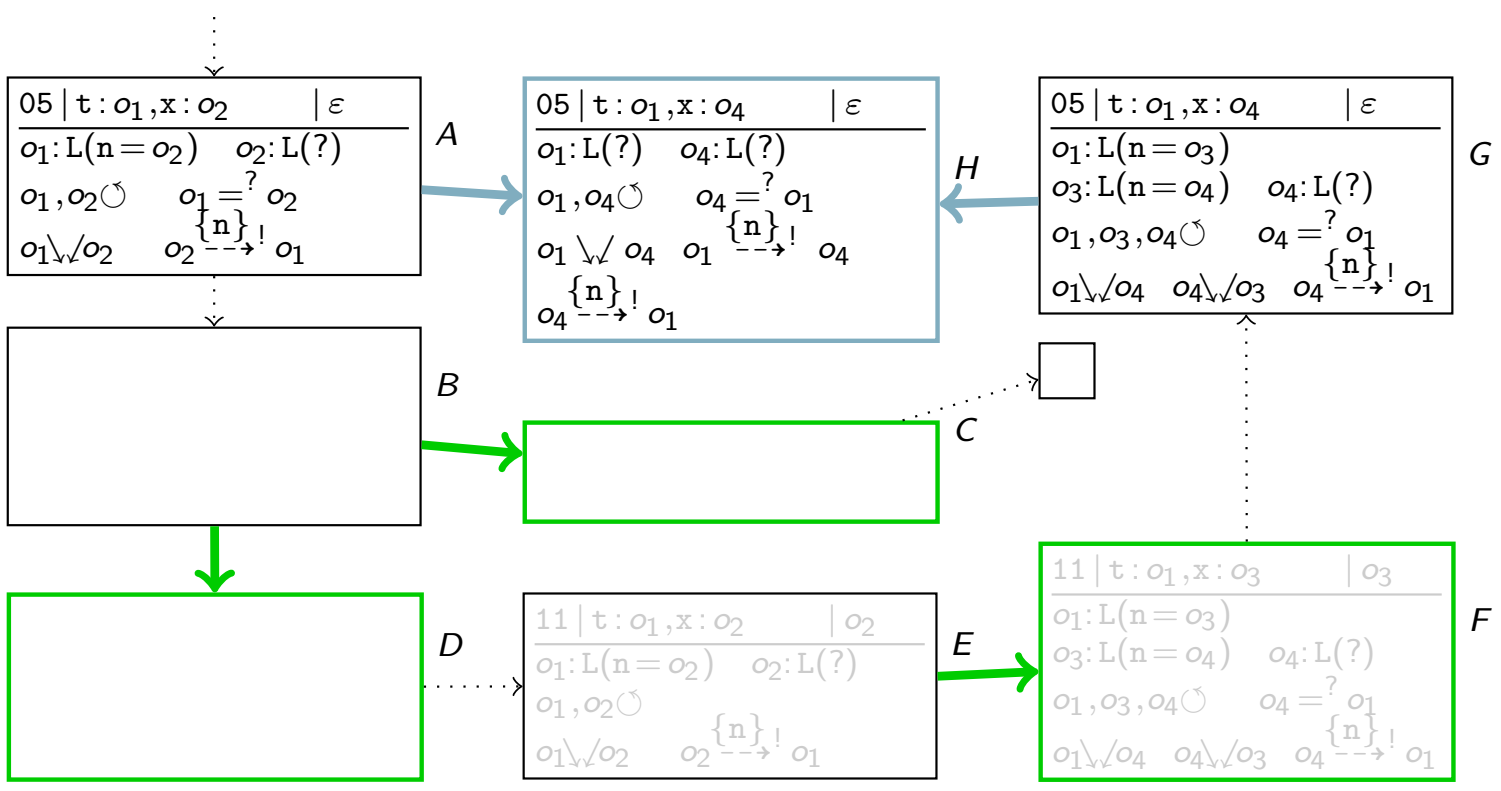
States G, H:

- Same program position as A $\Rightarrow$ Generalize

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

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

```
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 $\Rightarrow$ Generalize

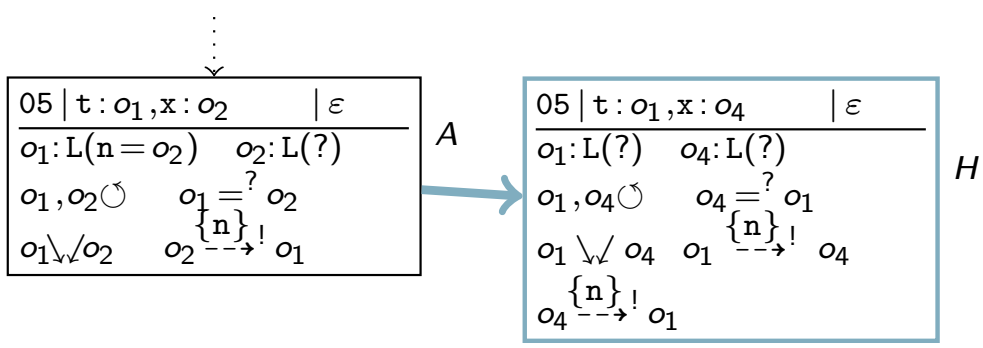
  In A: $\texttt{this} = o_1 \xrightarrow{\texttt{n}} o_2 = \texttt{x}$

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

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

```
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 $\Rightarrow$ Generalize

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

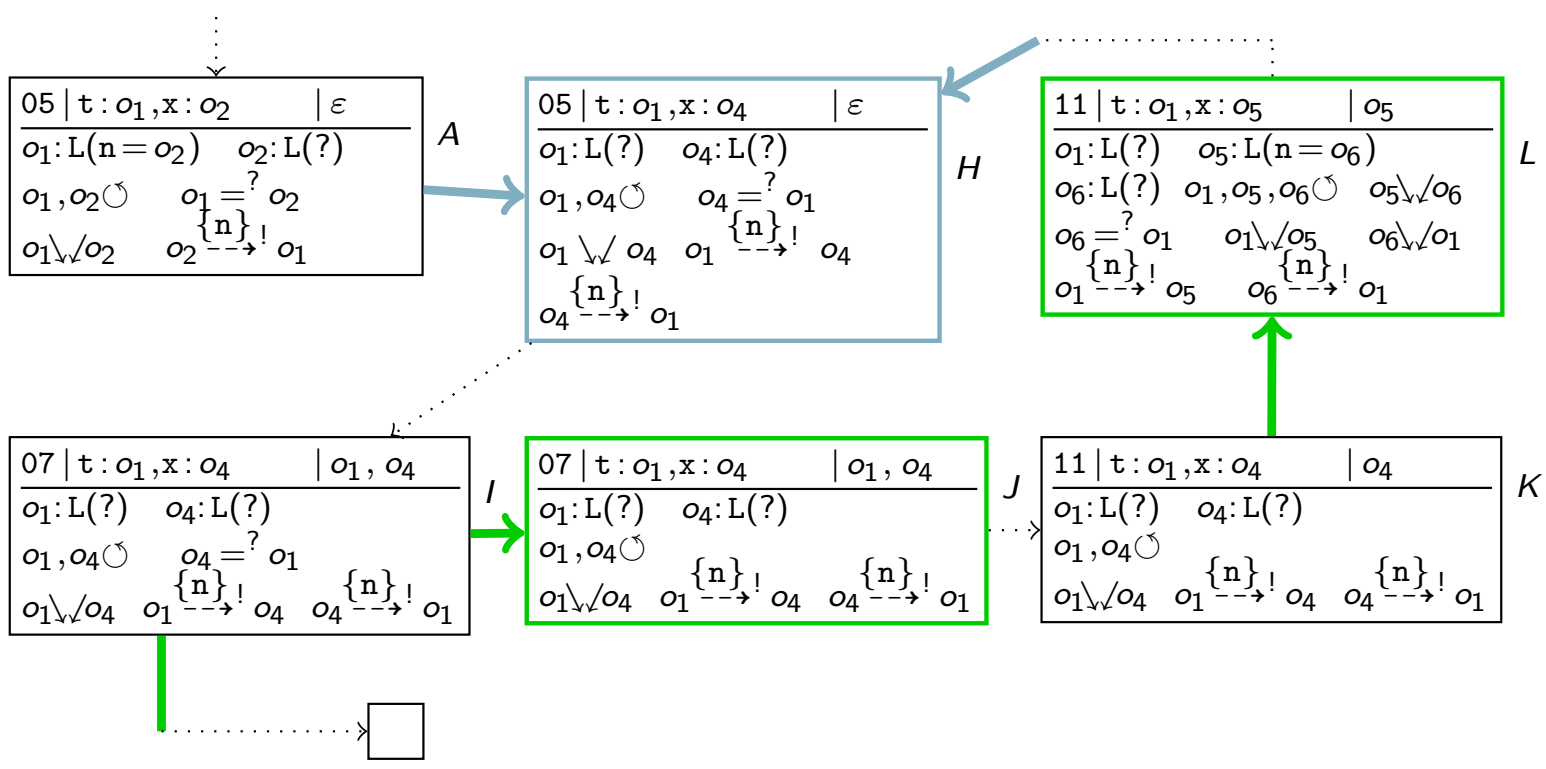
  In G: $\texttt{this} = o_1 \xrightarrow{\texttt{n}} o_3 \xrightarrow{\texttt{n}} o_4 = \texttt{x}$

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

- 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 $\Rightarrow$ Generalize

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

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

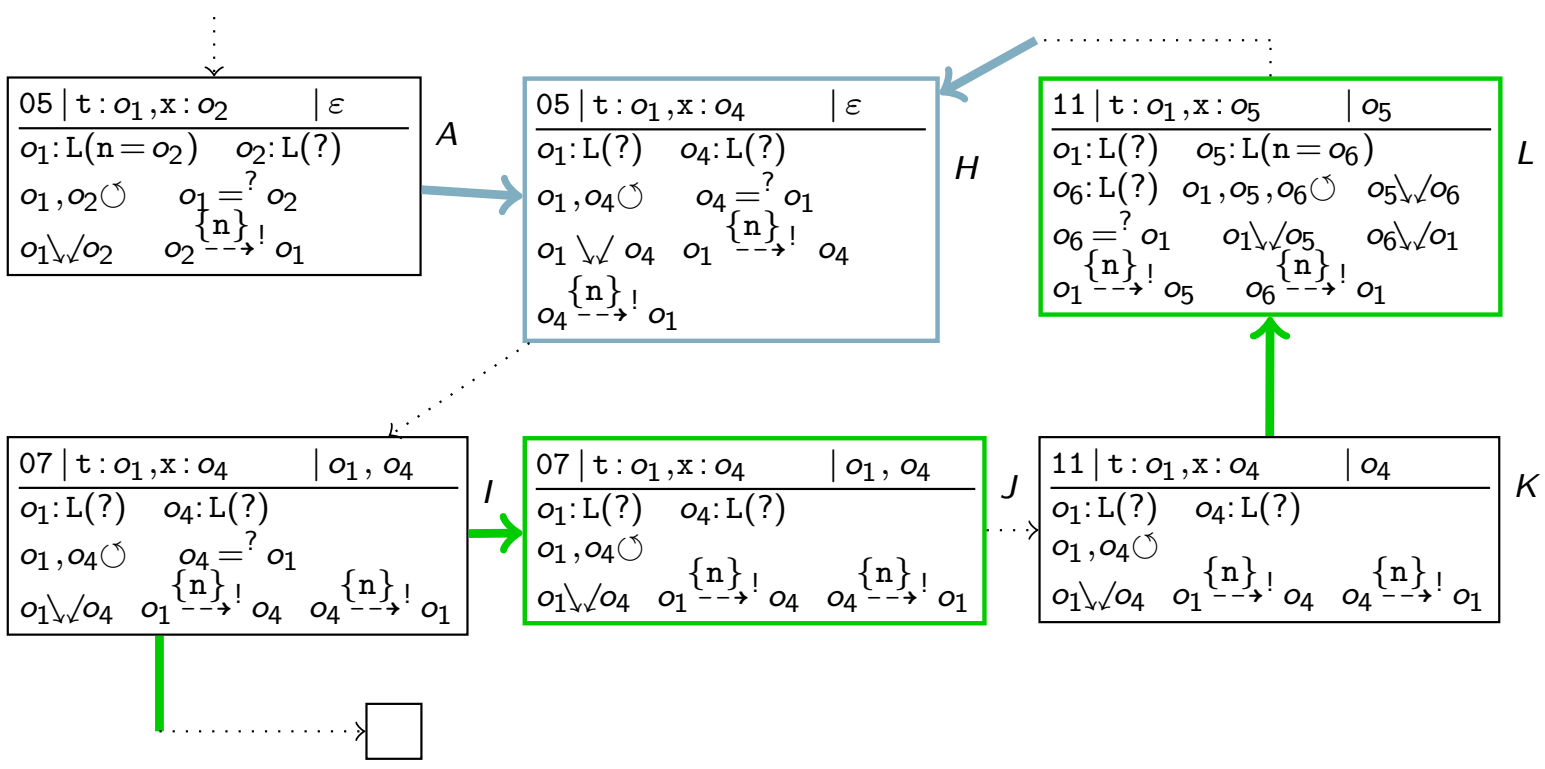
  $\Rightarrow$ In H: Abstract to $\texttt{this} = o_1 \xrightarrow{\{\texttt{n}\}!} o_4 = \texttt{x}$

- 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
```

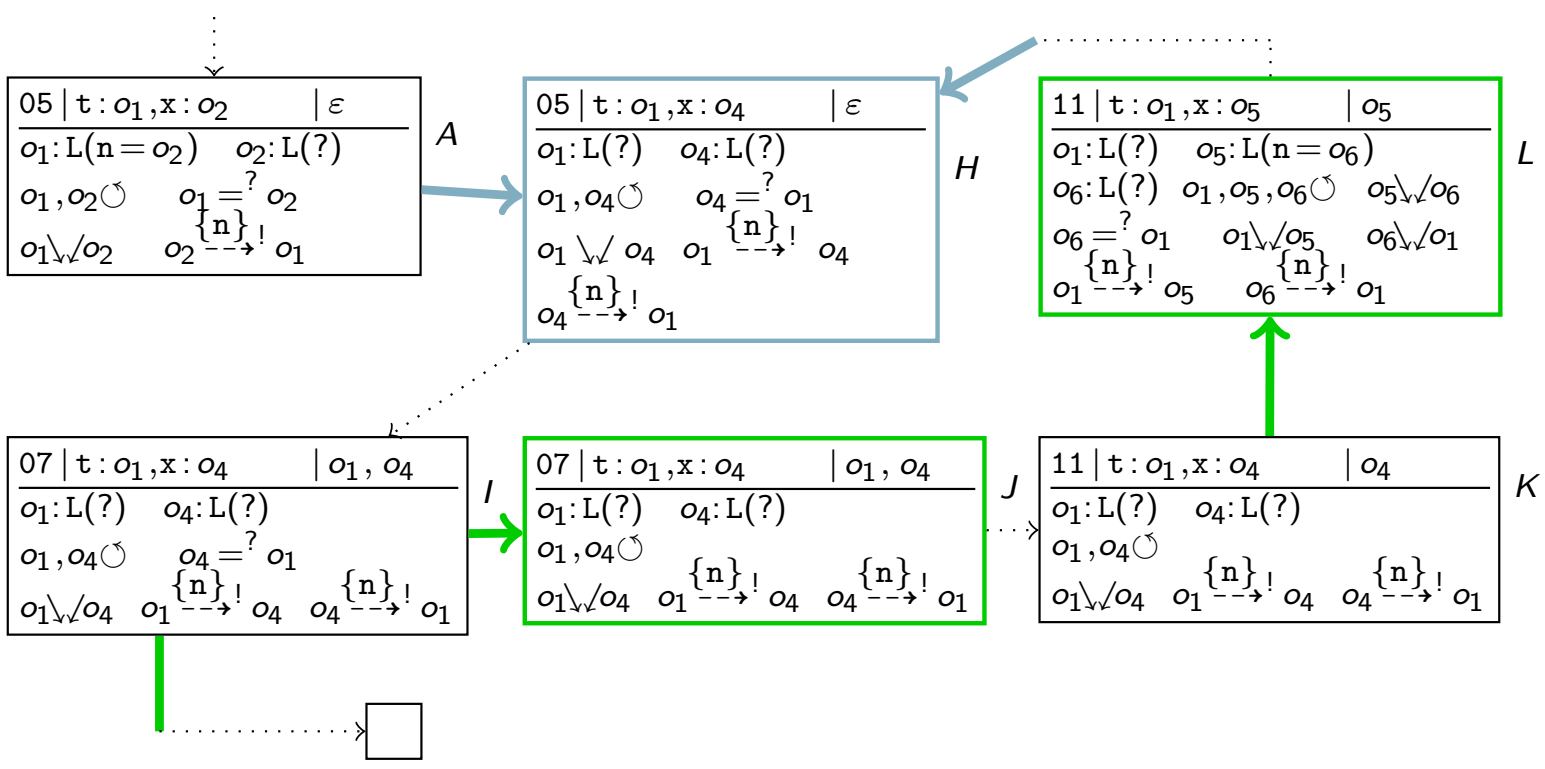


Proving termination with $R = o \xrightarrow{F}{}^{!} o'$:

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
```

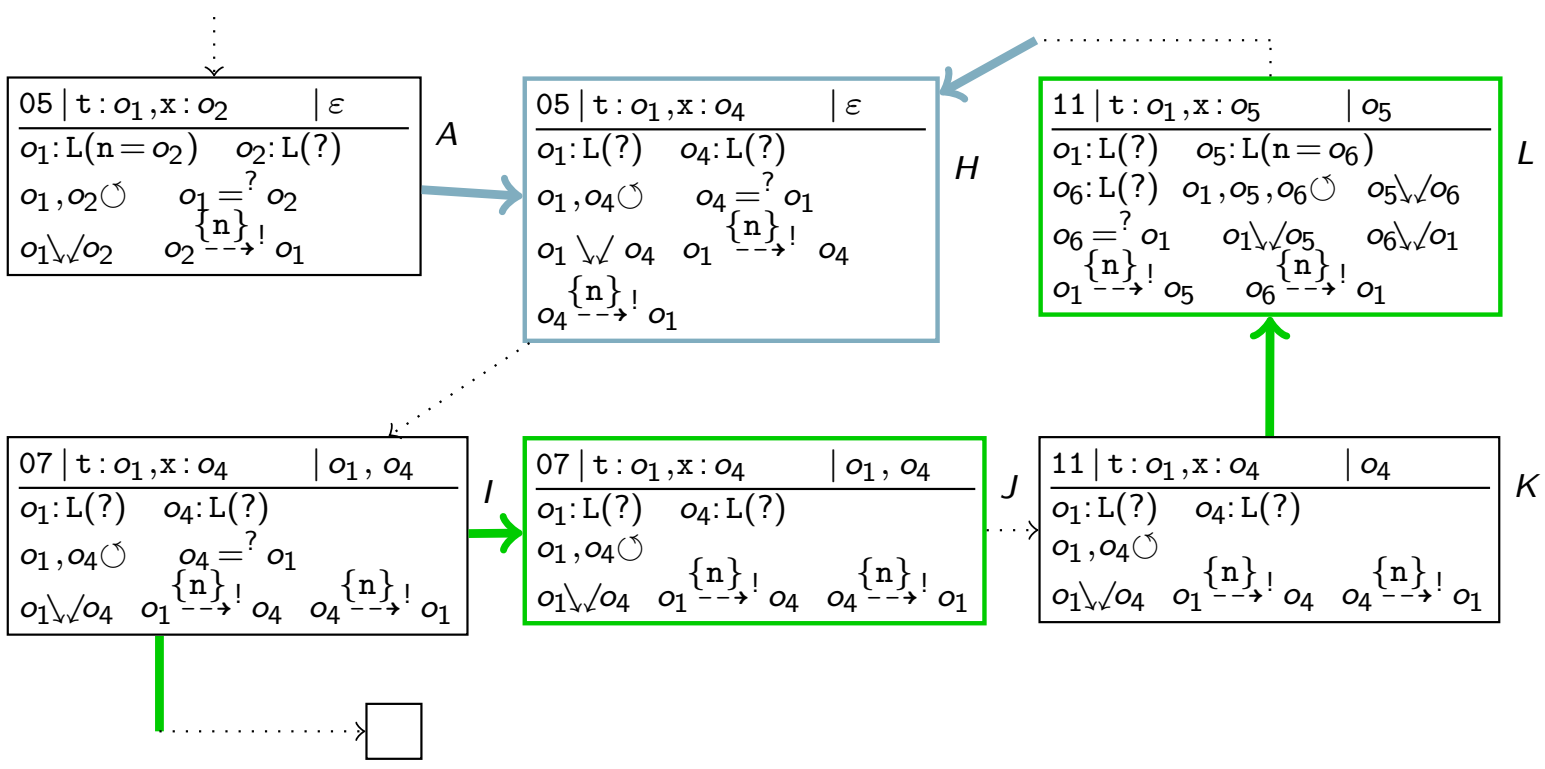


Proving termination with $R = o \xrightarrow{F}{}^{!} o'$:

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
```

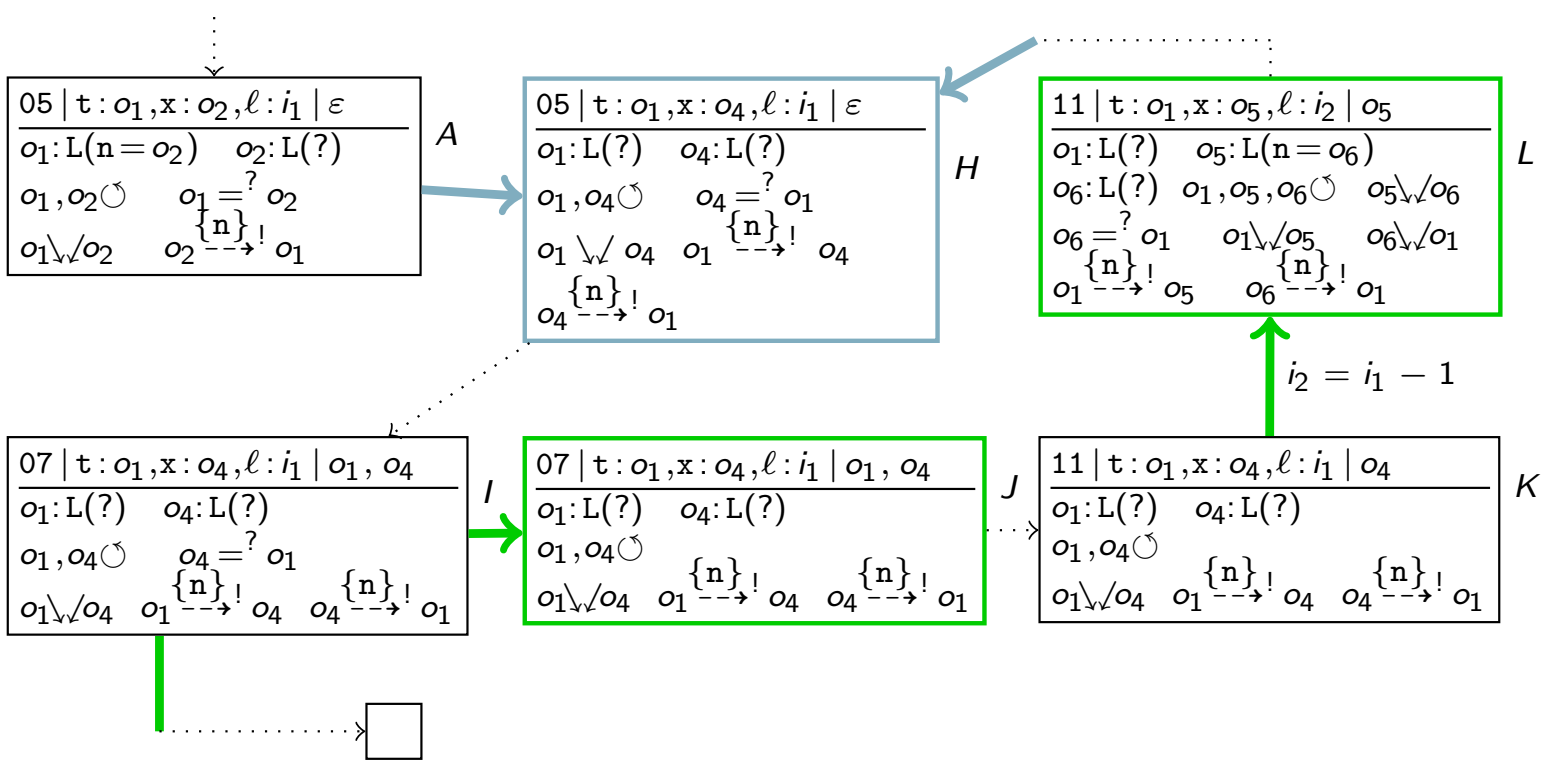


Proving termination with $R = o \xrightarrow{F}{}^{!} o'$:

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 $R' = \tilde{o} \xrightarrow{F}{}^{!} o'$)

```
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
```

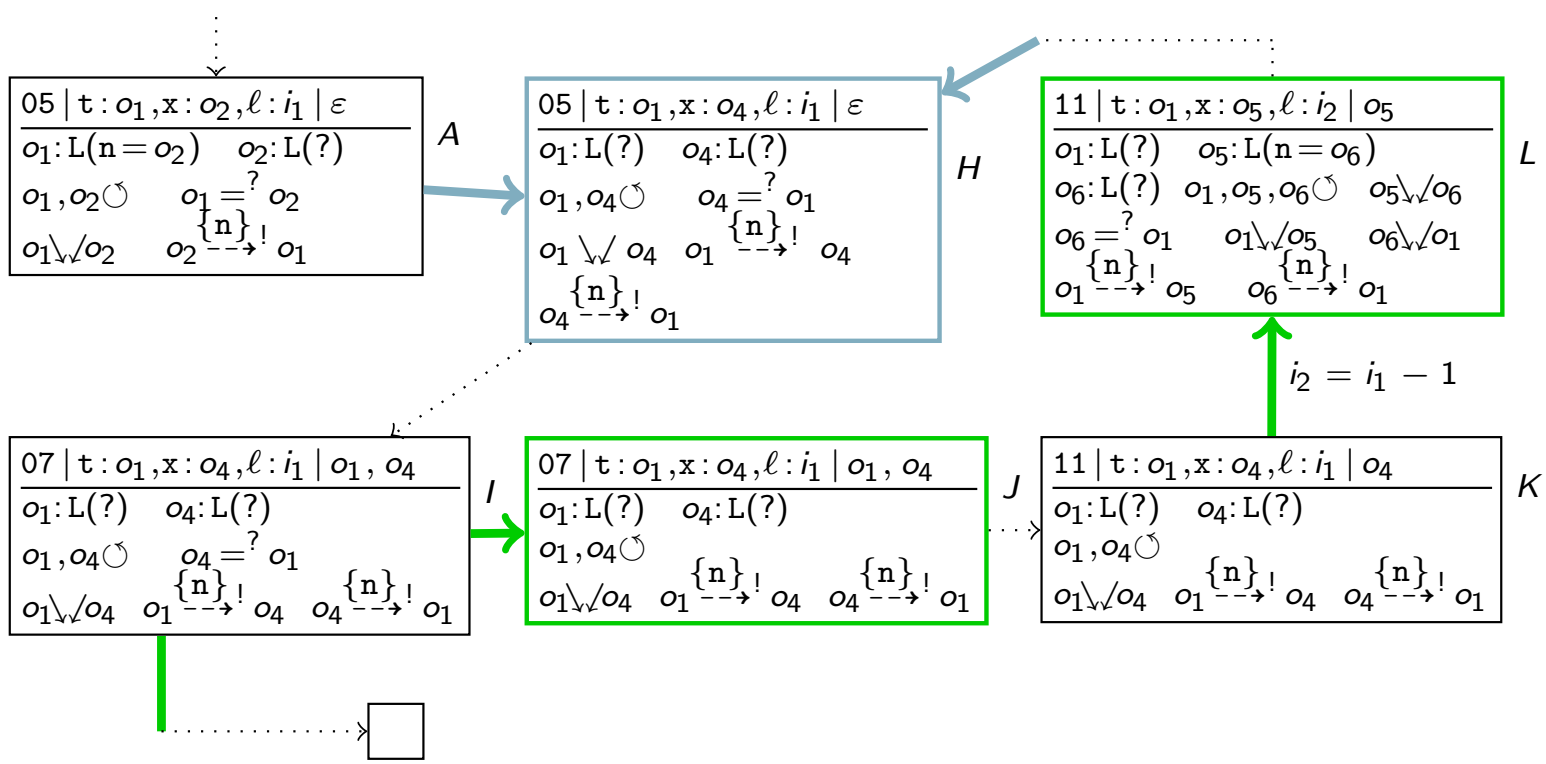


Proving termination with $R = o \xrightarrow{F}{}^{!} o'$:

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 $R' = \tilde{o} \xrightarrow{F}{}^{!} o'$)
3. Add variable for lengths to graphs (here: only done for ℓ $o_4 \xrightarrow{\{n\}}{}^{!} o_1$ as ℓ)

```
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 $R = o \xrightarrow{F}{}^{!} o'$:

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 $R' = \tilde{o} \xrightarrow{F}{}^{!} o'$)
3. Add variable for lengths to graphs (here: only done for $\ell_{o_4 \xrightarrow{\{n\}}{}^{!} o_1}$ as ℓ)

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

Automatically created TRS:

$$\mathsf{f}(\ldots, \ell) \to \mathsf{f}(\ldots, \ell - 1) \quad | \ \ell > 0$$

Other AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof [VIT'10] w.r.t. JINJA (Klein & Nipkow '06)

Other AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof [VIT'10] w.r.t. JINJA (Klein & Nipkow '06)
- Built-in, implicit analyses for nullness, aliasing, sharing, cyclicity

Other AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof [VIT'10] w.r.t. JINJA (Klein & Nipkow '06)
- Built-in, implicit analyses for nullness, aliasing, sharing, cyclicity
- Termination analysis for algorithms
  - on integers [RTA'10]
  - on acyclic user-defined data structures (trees, DAGs, ...) [RTA'10]

Other AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof [VIT'10] w.r.t. JINJA (Klein & Nipkow '06)
- Built-in, implicit analyses for nullness, aliasing, sharing, cyclicity
- Termination analysis for algorithms
  - on integers [RTA'10]
  - on acyclic user-defined data structures (trees, DAGs, ...) [RTA'10]
  - using recursion [RTA'11]

Other AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof [VIT'10] w.r.t. JINJA (Klein & Nipkow '06)
- Built-in, implicit analyses for nullness, aliasing, sharing, cyclicity
- Termination analysis for algorithms
  - on integers [RTA'10]
  - on acyclic user-defined data structures (trees, DAGs, ...) [RTA'10]
  - using recursion [RTA'11]
  - on arrays [FoVeOOS'11]

Other AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof [VIT'10] w.r.t. JINJA (Klein & Nipkow '06)
- Built-in, implicit analyses for nullness, aliasing, sharing, cyclicity
- Termination analysis for algorithms
  - on integers [RTA'10]
  - on acyclic user-defined data structures (trees, DAGs, ...) [RTA'10]
  - using recursion [RTA'11]
  - on arrays [FoVeOOS'11]
  - on cyclic data [CAV'12]:
    - by detecting (and ignoring) irrelevant cyclicity
    - by automatically finding and counting markers
    - by using definite reachability information

Other AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof [VIT'10] w.r.t. JINJA (Klein & Nipkow '06)
- Built-in, implicit analyses for nullness, aliasing, sharing, cyclicity
- Termination analysis for algorithms
  - on integers [RTA'10]
  - on acyclic user-defined data structures (trees, DAGs, ...) [RTA'10]
  - using recursion [RTA'11]
  - on arrays [FoVeOOS'11]
  - on cyclic data [CAV'12]:
    - by detecting (and ignoring) irrelevant cyclicity
    - by automatically finding and counting markers
    - by using definite reachability information
- *Non*-termination analysis [FoVeOOS'11]

Automated Termination Proofs for Java Programs with Cyclic Data

- Evaluated on collection of 387 programs:
  - Termination Problem Data Base
  - Standard libraries from `java.util` (JDK)

	Term	**NonT**	**Fail**	**t** (s)	**Term**	**NonT**	**Fail**	**t** (s)
AProVE '12	267	81	39	9.5	51	0	9	15.8
AProVE '11	225	81	81	11.4	23	0	37	18.3
Julia	191	22	174	4.7	32	0	28	8.2
COSTA	160	0	227	11.0	29	0	31	30.4
	all examples				`LinkedList` + `HashMap`			

Automated Termination Proofs for Java Programs with Cyclic Data

- Evaluated on collection of 387 programs:
  - Termination Problem Data Base
  - Standard libraries from `java.util` (JDK)

	Term	**NonT**	**Fail**	**t** (s)	**Term**	**NonT**	**Fail**	**t** (s)
AProVE '12	267	81	39	9.5	51	0	9	15.8
AProVE '11	225	81	81	11.4	23	0	37	18.3
Julia	191	22	174	4.7	32	0	28	8.2
COSTA	160	0	227	11.0	29	0	31	30.4
	all examples				`LinkedList` + `HashMap`			

- Won Termination Competition 2012

Automated Termination Proofs for Java Programs with Cyclic Data

- Evaluated on collection of 387 programs:
  - Termination Problem Data Base
  - Standard libraries from `java.util` (JDK)

	Term	**NonT**	**Fail**	**t** (s)	**Term**	**NonT**	**Fail**	**t** (s)
AProVE '12	267	81	39	9.5	51	0	9	15.8
AProVE '11	225	81	81	11.4	23	0	37	18.3
Julia	191	22	174	4.7	32	0	28	8.2
COSTA	160	0	227	11.0	29	0	31	30.4
	all examples				`LinkedList` + `HashMap`			

- Won Termination Competition 2012
- Termination graphs facilitate and simplify complex analysis

Automated Termination Proofs for Java Programs with Cyclic Data

- Evaluated on collection of 387 programs:
  - Termination Problem Data Base
  - Standard libraries from `java.util` (JDK)

	Term	**NonT**	**Fail**	**t** (s)	**Term**	**NonT**	**Fail**	**t** (s)
AProVE '12	267	81	39	9.5	51	0	9	15.8
AProVE '11	225	81	81	11.4	23	0	37	18.3
Julia	191	22	174	4.7	32	0	28	8.2
COSTA	160	0	227	11.0	29	0	31	30.4
	all examples				`LinkedList` + `HashMap`			

- Won Termination Competition 2012
- Termination graphs facilitate and simplify complex analysis
- http://aprove.informatik.rwth-aachen.de/eval/JBC-Cyclic