Proving Termination of Heap-Manipulating Java Programs

Marc Brockschmidt

LuFG Informatik 2, RWTH Aachen University, Germany

March 2013

Automated Termination Analysis

Imperative Programs:

Automated Termination Analysis

Imperative Programs:

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

Automated Termination Analysis

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

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)
  ...

Declarative Programs:

Automated Termination Analysis

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)
  ...

Declarative Programs:

- Logic Programming (since the 70s)

Automated Termination Analysis

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)
  ...

Declarative Programs:

- Logic Programming (since the 70s)
- Term Rewriting (since the 70s)

Automated Termination Analysis

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)
  ...

Declarative Programs:

- Logic Programming (since the 70s)
- Term Rewriting (since the 70s)

Transformation of Imperative to Declarative Programs:

Automated Termination Analysis

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)
  ...

Declarative Programs:

- Logic Programming (since the 70s)
- Term Rewriting (since the 70s)

Transformation of Imperative to Declarative Programs:

- Termination Analysis for C (via Polynomial Orders)
  KITTeL – (*Falke, Kapur, Sinz*, since '11)

Automated Termination Analysis

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)
  ...

Declarative Programs:

- Logic Programming (since the 70s)
- Term Rewriting (since the 70s)

Transformation of Imperative to Declarative Programs:

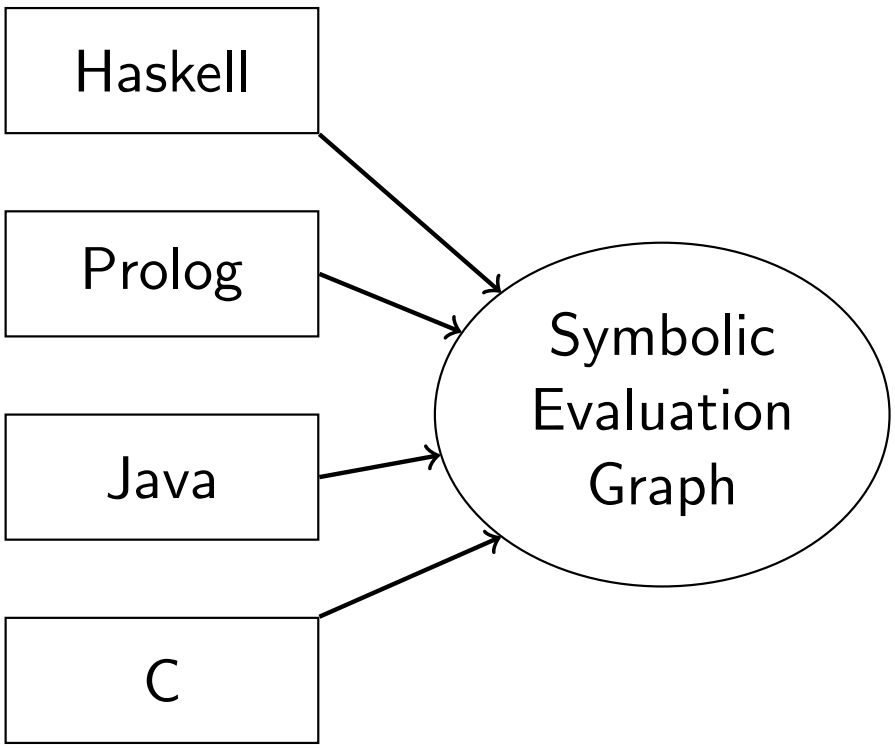
- Termination Analysis for C (via Polynomial Orders)
  KITTeL – (*Falke, Kapur, Sinz*, since '11)
- Termination Analysis for Java (via Path Length, CLP backend)
  Julia – (*Spoto, Mesnard, Payet*, since '08)
  COSTA – (*Albert, Arenas, Codish, Genaim, Puebla, Zanardini*, since '08)

Rewriting-backed approach: Idea

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

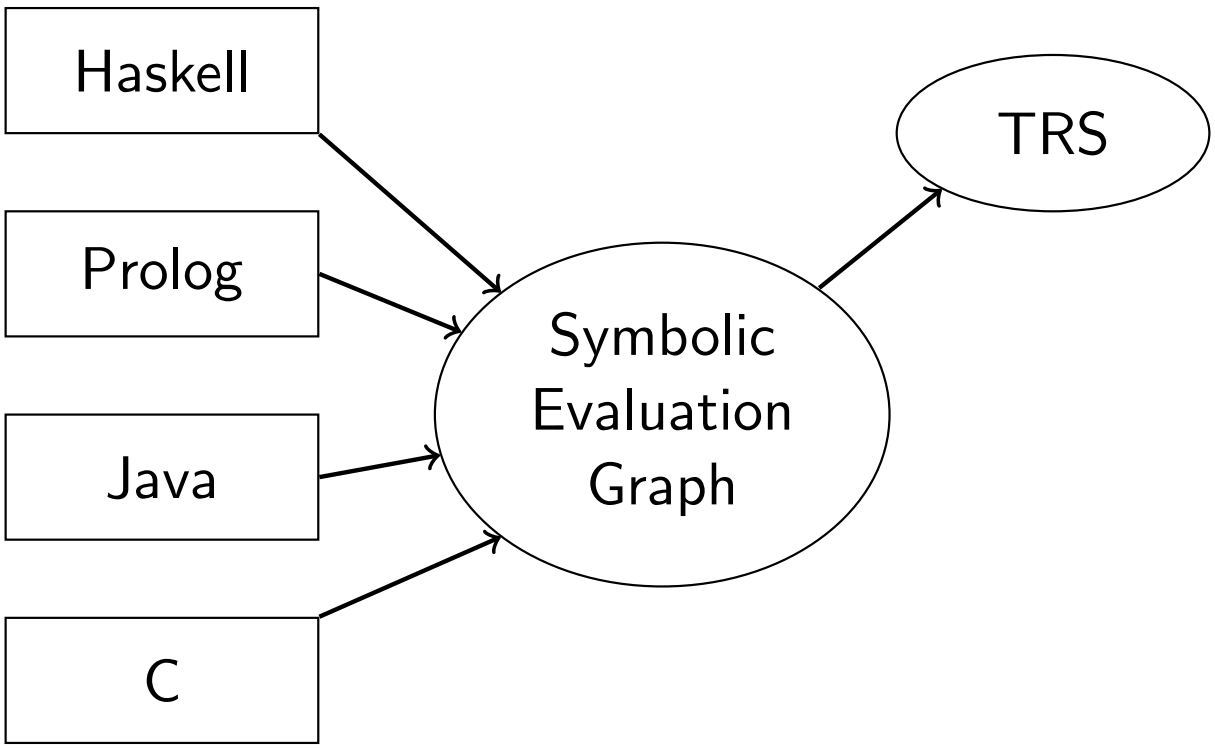
Rewriting-backed approach: Idea

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



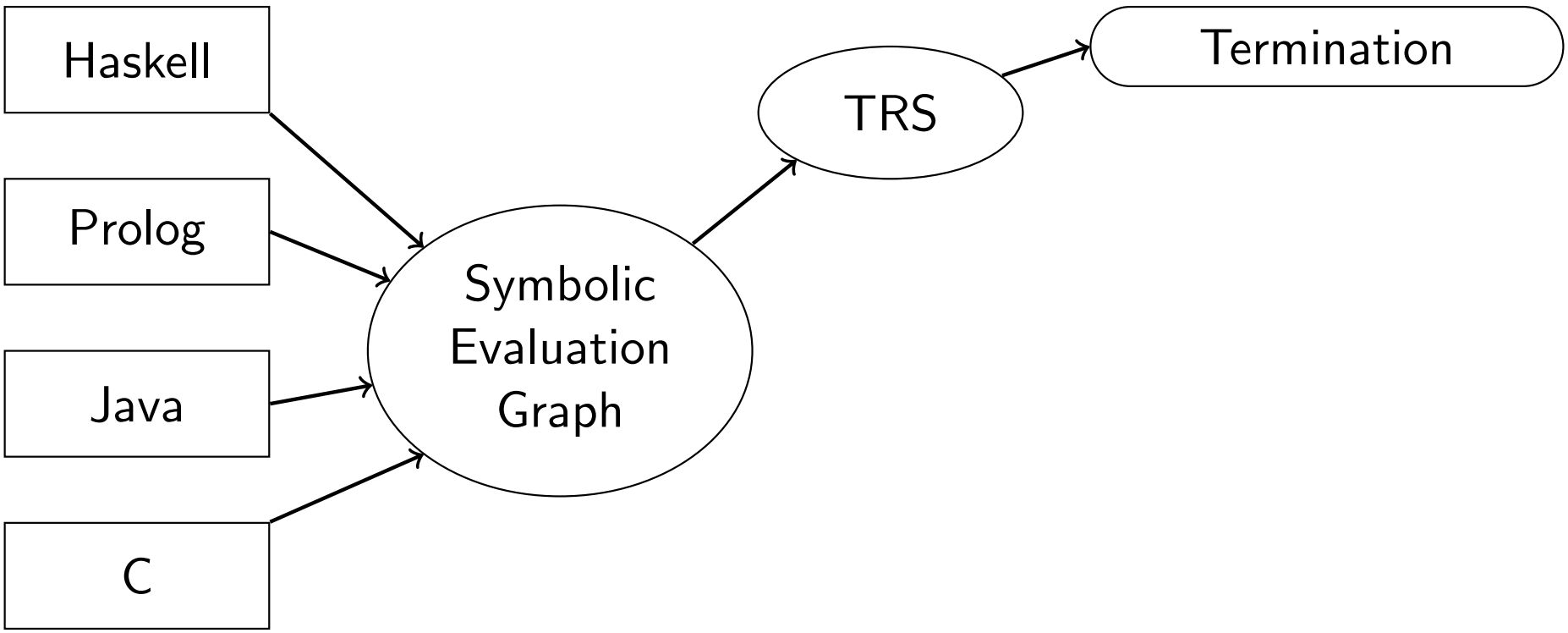
Rewriting-backed approach: Idea

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



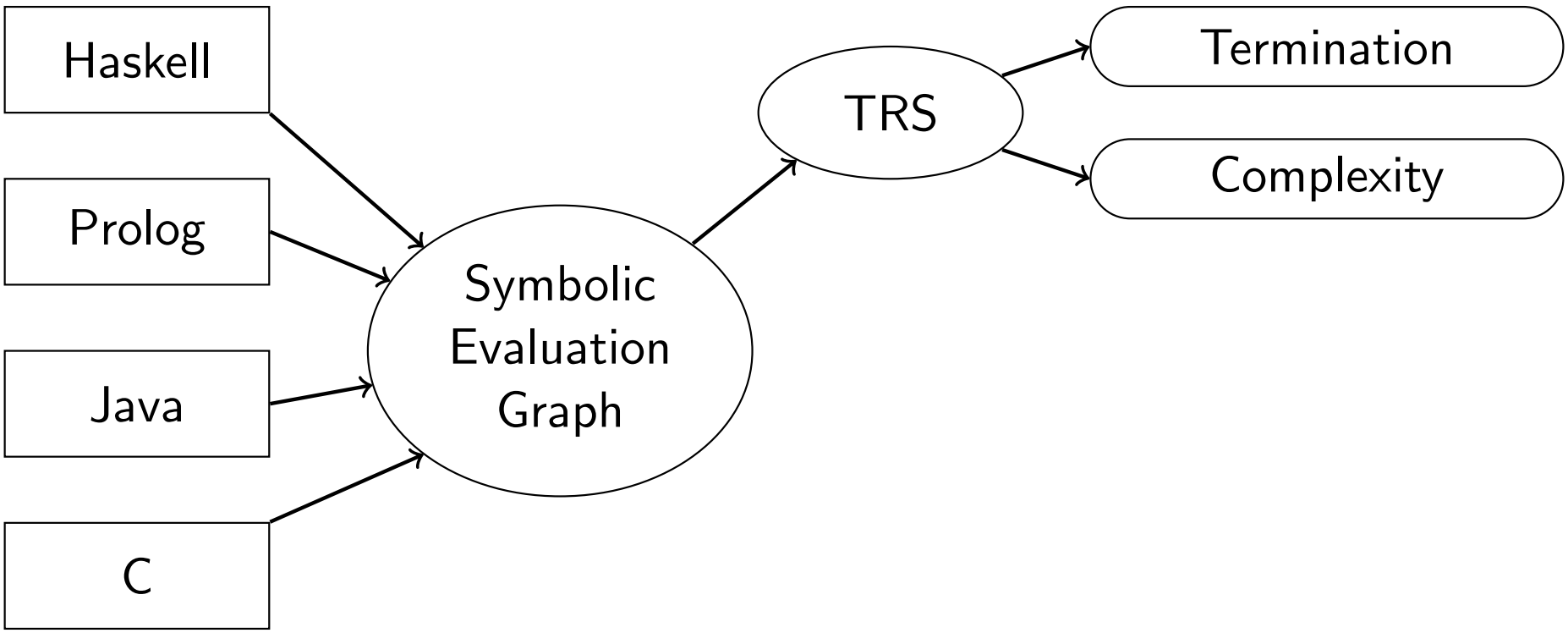
Rewriting-backed approach: Idea

- Programming languages *hard* ↷ Simpler representation needed
- Symbolic Evaluation Graphs: Simpler, contain all information
- Term Rewrite Systems (TRSs) generated from Symbolic Evaluation
- Prove TRS termination using existing provers



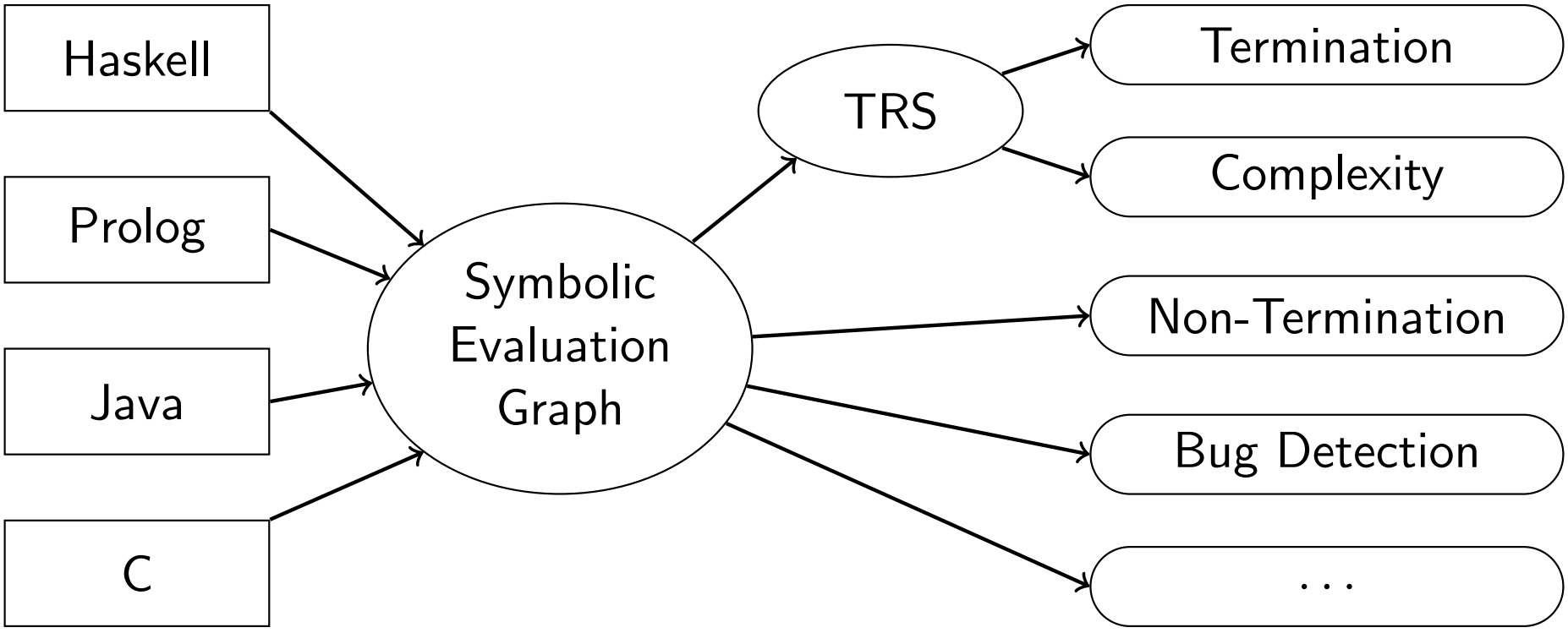
Rewriting-backed approach: Idea

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



Rewriting-backed approach: Idea

- Programming languages *hard* ↷ Simpler representation needed
- Symbolic Evaluation Graphs: Simpler, contain all information
- Term Rewrite Systems (TRSs) generated from Symbolic Evaluation
- Prove TRS termination using existing provers



Rewriting-backed approach: Advantages

Natural handling of user-defined data structures:

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

Rewriting-backed approach: Advantages

Natural 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

Natural 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

Natural 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

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

Rewriting-backed approach: Challenges

Rewriting-backed approach: Challenges

Terms cannot fully represent the heap:

1. Side-effects via *sharing*
2. No measure of *distances*
3. No representation for *cyclic* structures

Rewriting-backed approach: Challenges

Terms cannot fully represent the heap:

1. Side-effects via *sharing*
2. No measure of *distances*
3. No representation for *cyclic* structures

Solutions:

1. Overapproximate
2. Handle in symbolic evaluation
3. Post-process: Represent distances via counters

Rewriting-backed approach: Challenges

Terms cannot fully represent the heap:

1. Side-effects via *sharing*
2. No measure of *distances*
3. No representation for *cyclic* structures

Solutions:

1. Overapproximate
2. Handle in symbolic evaluation
3. Post-process: Represent ... via counters

Overview

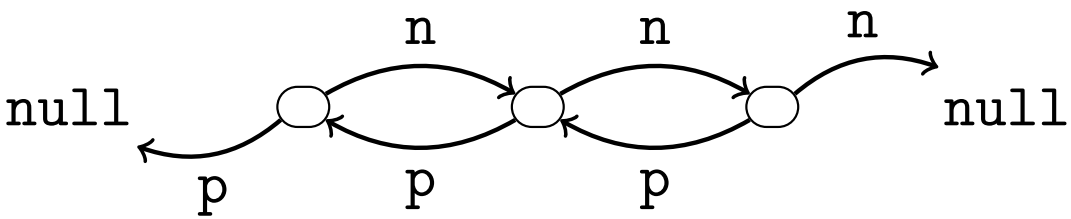
1. Introduction
2. Building Symbolic Evaluation Graphs
3. Generating TRSs from Symbolic Evaluation Graphs
4. Post-processing Symbolic Evaluation Graphs
5. Conclusion

length: the example

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```

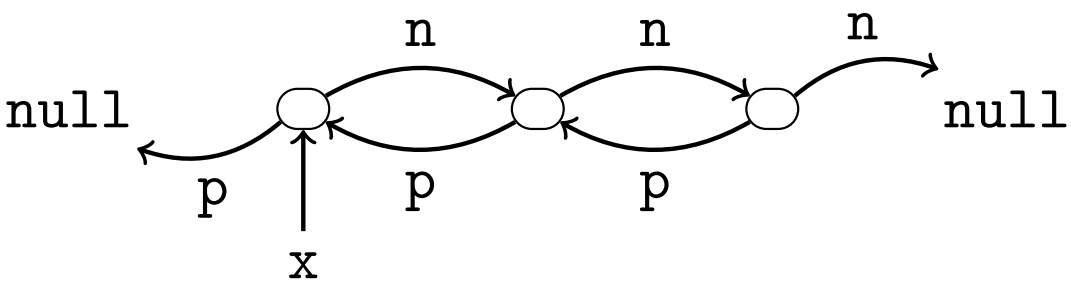
length: the example

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```



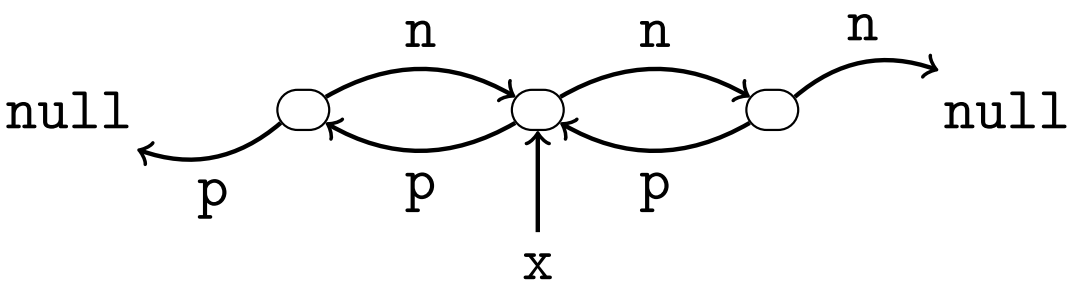
length: the example

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```



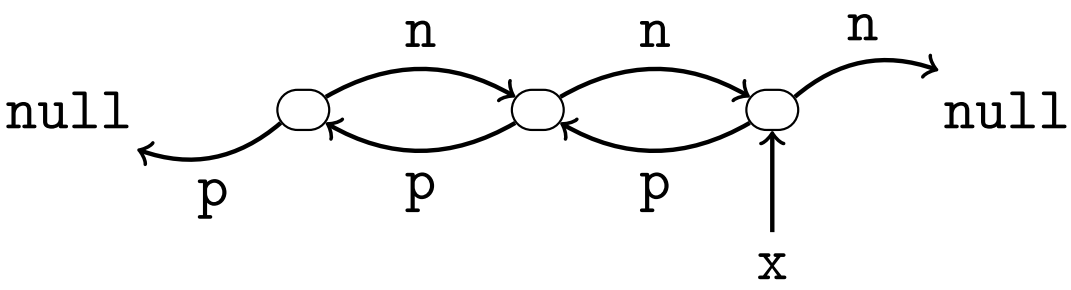
length: the example

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```



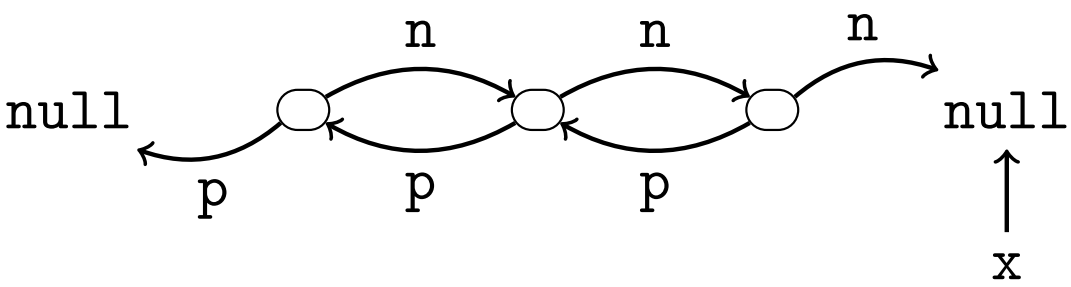
length: the example

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```



length: the example

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```



length: the example

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```

```
00: iconst_0      #load 0
01: istore_1      #store to r
02: aload_0       #load x
03: ifnull 17     #jump if x null
06: aload_0       #load x
07: getfield n    #get n from x
10: astore_0      #store to x
11: iinc 1, 1     #increment r
14: goto 2
17: iload_1       #load r
18: ireturn       #return r
```

The abstract domain: symbolic states

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```

```
00: iconst_0      #load 0
01: istore_1      #store to r
02: aload_0       #load x
03: ifnull 17     #jump if x null
06: aload_0       #load x
07: getfield n    #get n from x
10: astore_0      #store to x
11: iinc 1, 1     #increment r
14: goto 2
17: iload_1       #load r
18: ireturn       #return r
```

The abstract domain: symbolic states

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```

```
00: iconst_0      #load 0
01: istore_1      #store to r
02: aload_0       #load x
03: ifnull 17     #jump if x null
06: aload_0       #load x
07: getfield n    #get n from x
10: astore_0      #store to x
11: iinc 1, 1     #increment r
14: goto 2
17: iload_1       #load r
18: ireturn       #return r
```

Stack frame:

00	x : o_1	ε
o_1 : L(?)		$o_1 \circlearrowleft_{\{p,n\}}$

- Next program instruction

The abstract domain: symbolic states

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```

```
00: iconst_0      #load 0
01: istore_1      #store to r
02: aload_0       #load x
03: ifnull 17     #jump if x null
06: aload_0       #load x
07: getfield n    #get n from x
10: astore_0      #store to x
11: iinc 1, 1     #increment r
14: goto 2
17: iload_1       #load r
18: ireturn       #return r
```

00	x : o_1	ε
o_1 : L(?)	o_1 ↻ {p,n}	

Stack frame:

- Next program instruction
- Local variables
- Operand stack

The abstract domain: symbolic states

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```

```
00: iconst_0      #load 0
01: istore_1      #store to r
02: aload_0       #load x
03: ifnull 17     #jump if x null
06: aload_0       #load x
07: getfield n    #get n from x
10: astore_0      #store to x
11: iinc 1, 1     #increment r
14: goto 2
17: iload_1       #load r
18: ireturn       #return r
```

00 | x : o_1 | ε
o_1 : L(?) $o_1\circlearrowleft_{\{p,n\}}$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Heap information:

The abstract domain: symbolic states

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```

```
00: iconst_0      #load 0
01: istore_1      #store to r
02: aload_0       #load x
03: ifnull 17     #jump if x null
06: aload_0       #load x
07: getfield n    #get n from x
10: astore_0      #store to x
11: iinc 1, 1     #increment r
14: goto 2
17: iload_1       #load r
18: ireturn       #return r
```

00 | x : o_1 | ε
o_1 : L(?) $o_1\circlearrowleft_{\{p,n\}}$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Heap information:

- o_1 is L object or `null`

The abstract domain: symbolic states

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```

```
00: iconst_0      #load 0
01: istore_1      #store to r
02: aload_0       #load x
03: ifnull 17     #jump if x null
06: aload_0       #load x
07: getfield n    #get n from x
10: astore_0      #store to x
11: iinc 1, 1     #increment r
14: goto 2
17: iload_1       #load r
18: ireturn       #return r
```

00 | x : o_1 | ε
o_1 : L(?) $o_1\circlearrowleft_{\{p,n\}}$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Heap information:

- o_1 is L object or null
- Known L object: $o_2 : \mathrm{L}(\mathrm{n} = o_3)$

The abstract domain: symbolic states

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```

```
00: iconst_0      #load 0
01: istore_1      #store to r
02: aload_0       #load x
03: ifnull 17     #jump if x null
06: aload_0       #load x
07: getfield n    #get n from x
10: astore_0      #store to x
11: iinc 1, 1     #increment r
14: goto 2
17: iload_1       #load r
18: ireturn       #return r
```

00 | x : o_1 | ε
o_1 : L(?) $o_1\circlearrowleft_{\{p,n\}}$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Heap information:

- o_1 is L object or null
- Known L object: $o_2 : \mathrm{L}(\mathrm{n} = o_3)$
- Unknown integer: $i_1 : \mathbb{Z}$

The abstract domain: symbolic states

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```

```
00: iconst_0      #load 0
01: istore_1      #store to r
02: aload_0       #load x
03: ifnull 17     #jump if x null
06: aload_0       #load x
07: getfield n    #get n from x
10: astore_0      #store to x
11: iinc 1, 1     #increment r
14: goto 2
17: iload_1       #load r
18: ireturn       #return r
```

00 | $x : o_1$ | ε
$o_1 : L(?)$ $\quad o_1 \circlearrowleft_{\{p,n\}}$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Heap information:

- o_1 is L object or null
- Known L object: $o_2 : L(n = o_3)$
- Unknown integer: $i_1 : \mathbb{Z}$

Only explicit sharing

The abstract domain: symbolic states

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```

```
00: iconst_0      #load 0
01: istore_1      #store to r
02: aload_0       #load x
03: ifnull 17     #jump if x null
06: aload_0       #load x
07: getfield n    #get n from x
10: astore_0      #store to x
11: iinc 1, 1     #increment r
14: goto 2
17: iload_1       #load r
18: ireturn       #return r
```

00 | x : o_1 | ε
o_1 : L(?) $o_1 \circlearrowleft_{\{p,n\}}$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Heap information:

- o_1 is L object or null
- Known L object: $o_2 : \mathrm{L}(\mathrm{n} = o_3)$
- Unknown integer: $i_1 : \mathbb{Z}$

Heap predicates: **Only explicit sharing**

- Two references may be equal: $o_1 =^? o_2$

The abstract domain: symbolic states

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```

```
00: iconst_0      #load 0
01: istore_1      #store to r
02: aload_0       #load x
03: ifnull 17     #jump if x null
06: aload_0       #load x
07: getfield n    #get n from x
10: astore_0      #store to x
11: iinc 1, 1     #increment r
14: goto 2
17: iload_1       #load r
18: ireturn       #return r
```

00 | x : o_1 | ε
o_1 : L(?) $o_1 \circlearrowleft_{\{p,n\}}$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

Heap information:

- o_1 is L object or null
- Known L object: $o_2 : \mathrm{L}(\mathrm{n} = o_3)$
- Unknown integer: $i_1 : \mathbb{Z}$

Heap predicates: **Only explicit sharing**

- Two references may be equal: $o_1 =^? o_2$
- Two references may share: $o_1 \diagdown\!\!\diagup o_2$

The abstract domain: symbolic states

```
class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r; }}
```

```
00: iconst_0      #load 0
01: istore_1      #store to r
02: aload_0       #load x
03: ifnull 17     #jump if x null
06: aload_0       #load x
07: getfield n    #get n from x
10: astore_0      #store to x
11: iinc 1, 1     #increment r
14: goto 2
17: iload_1       #load r
18: ireturn       #return r
```

00 | x : o_1 | ε
o_1 : L(?) $\quad o_1 \circlearrowleft_{\{p,n\}}$

Stack frame:

- Next program instruction
- Local variables
- Operand stack

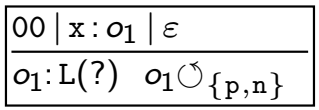
Heap information:

- o_1 is L object or null
- Known L object: $o_2 : \mathrm{L}(\mathrm{n} = o_3)$
- Unknown integer: $i_1 : \mathbb{Z}$

Heap predicates: **Only explicit sharing**

- Two references may be equal: $o_1 =^? o_2$
- Two references may share: $o_1 \diagdown\!\!\diagup o_2$
- Reference might have cycles containing all fields F: $o_1 \circlearrowleft_F$

```
00: iconst_0
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```



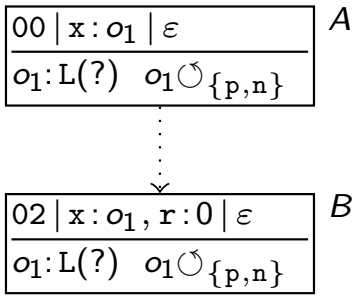
A

State A:

- x some list, might contain cycles using p and n

```
int length(L x) {
  int r = 0;
  while (x != null) {
    x = x.n; r++;   }
  return r;      }
```

```
00: iconst_0
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```



State *A*:

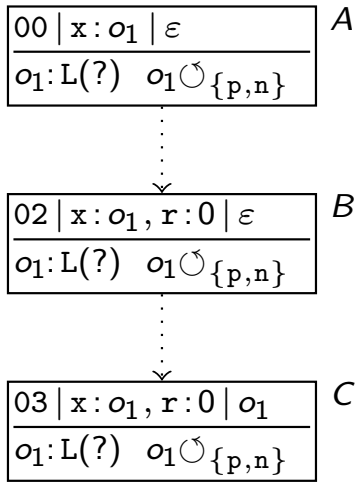
- `x` some list, might contain cycles using `p` and `n`

State *B*:

- Initialized variable `r` to 0

```
int length(L x) {
  int r = 0;
  while (x != null) {
    x = x.n; r++;   }
  return r;     }
```

```
00: iconst_0
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```



State *A*:

- `x` some list, might contain cycles using `p` and `n`

State *B*:

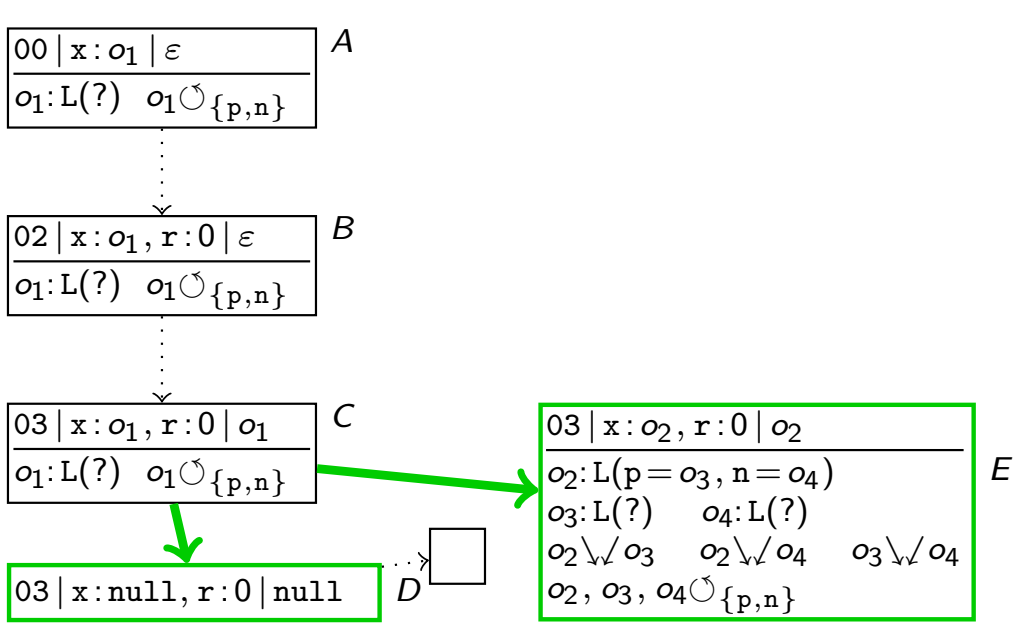
- Initialized variable `r` to 0

State *C*:

- `x` (o_1) null? We do not know!

```
int length(L x) {
  int r = 0;
  while (x != null) {
    x = x.n; r++;   }
  return r;     }
```

```
00: iconst_0
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```



State *A*:

- `x` some list, might contain cycles using `p` and `n`

State *B*:

- Initialized variable `r` to 0

States *C*, *D*, *E*:

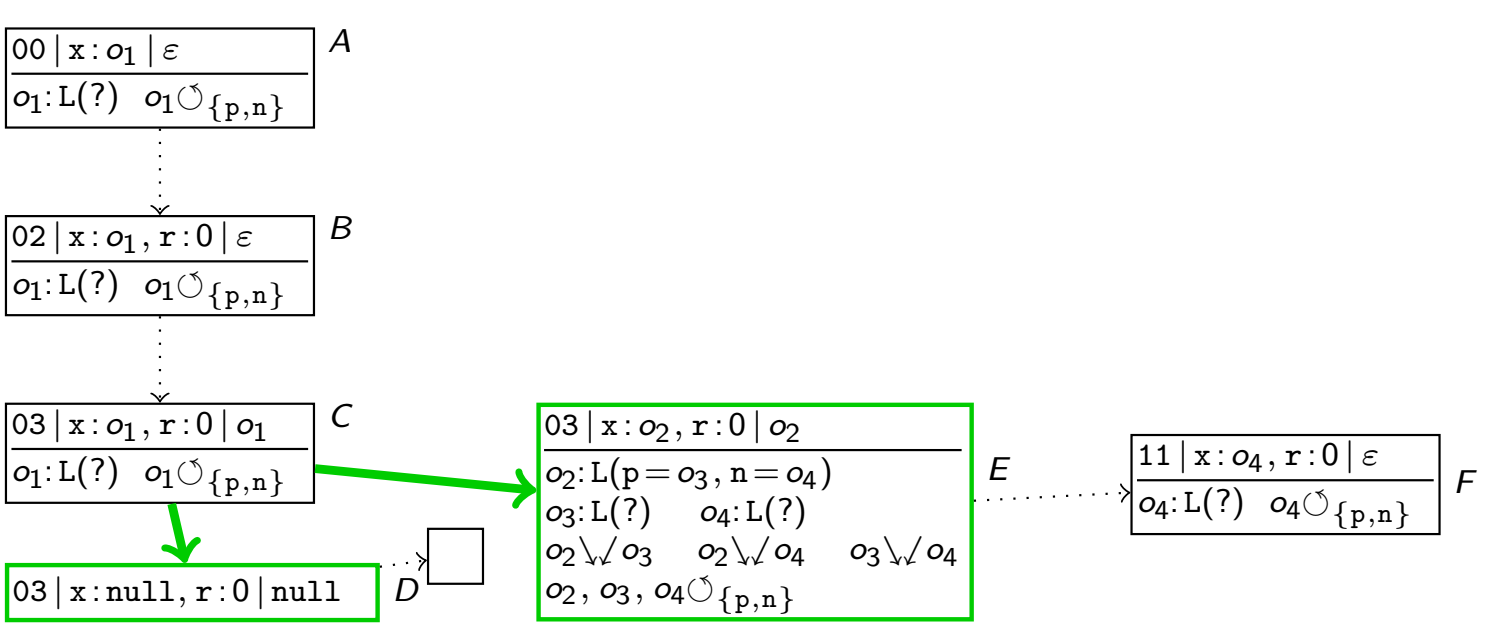
- `x` (o_1) null? We do not know!

⇒ Refinement

- In *D*: o_1 is null (↷ program ends)
- In *E*: o_1 replaced by o_2, which exists and has fields:
  - Field values can share (↷ add ↘↙)
  - Field values can be cyclic again (↷ add ↻)

```
int length(L x) {
  int r = 0;
  while (x != null) {
    x = x.n; r++;   }
  return r;     }
```

```
00: iconst_0
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```

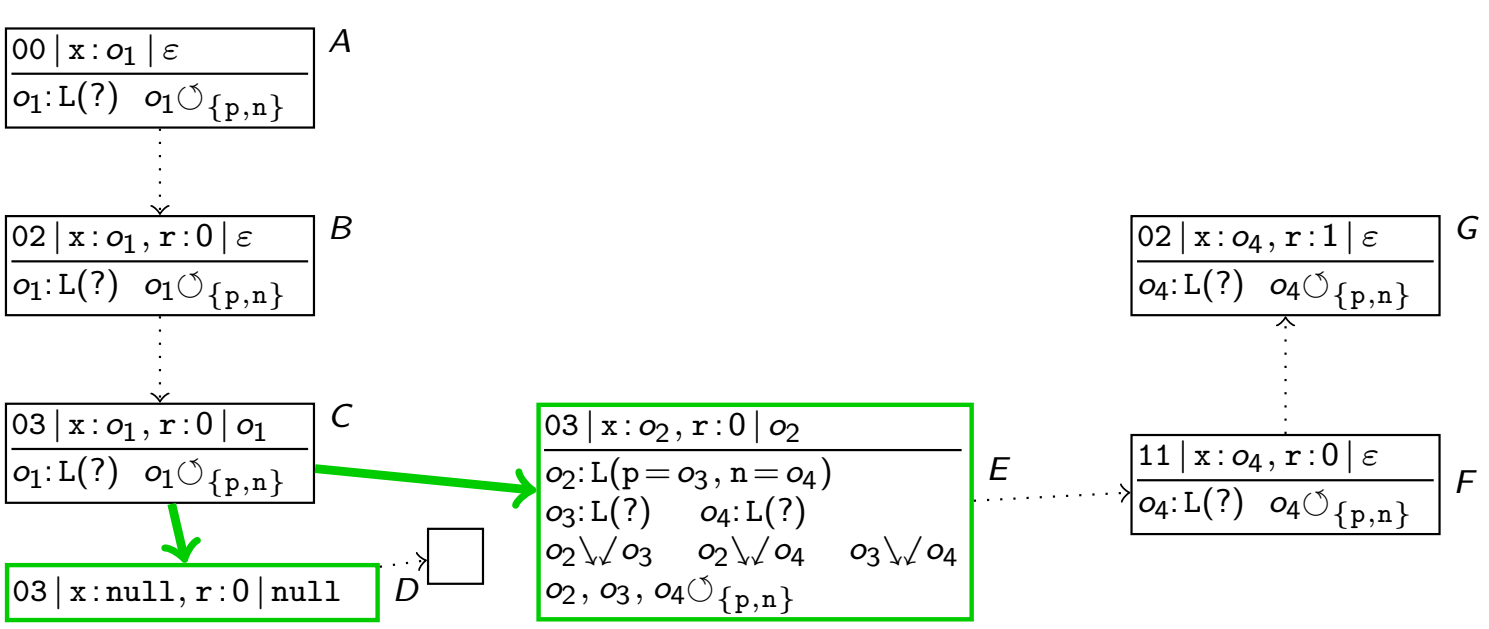


State *F*:

- Stored `x.n` to `x` (allowing for GC)

```
int length(L x) {
  int r = 0;
  while (x != null) {
    x = x.n; r++;   }
  return r;       }
```

```
00: iconst_0
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```



State *F*:

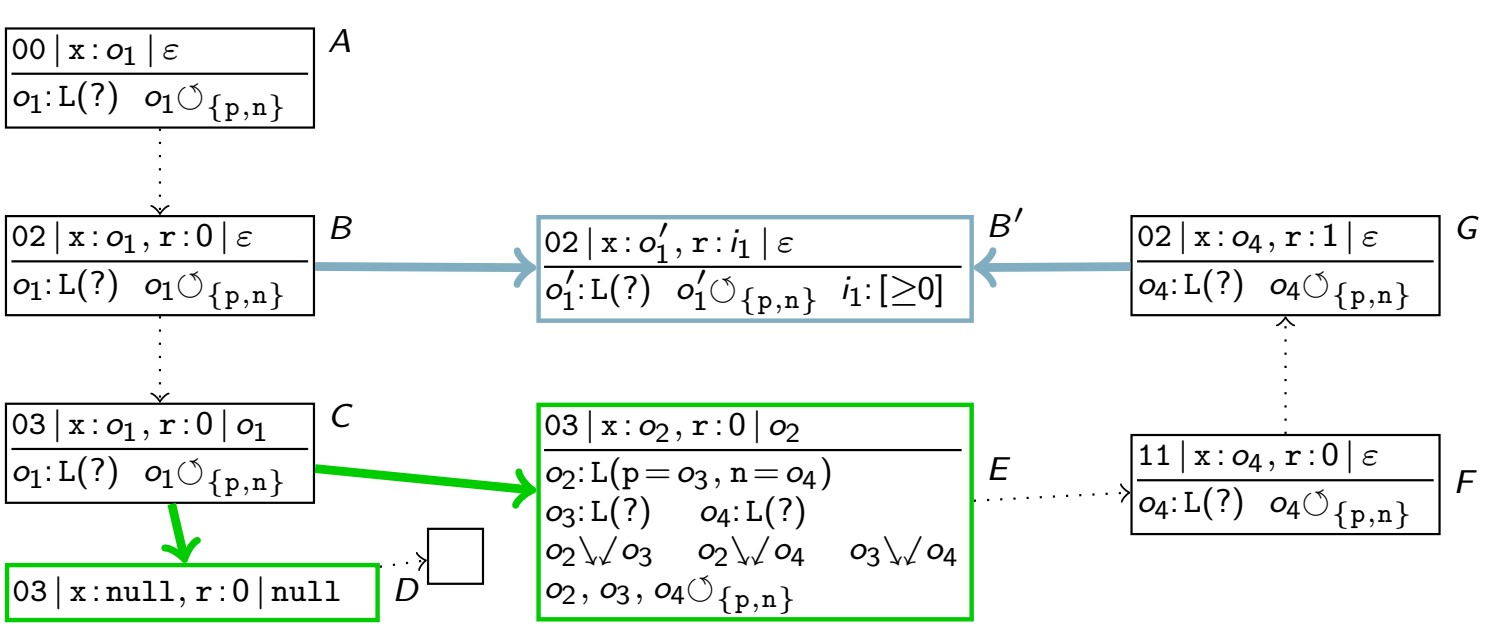
- Stored `x.n` to `x` (allowing for GC)

State *G*:

- Incremented `r`, back to position 02 (as *B*)

```
int length(L x) {
  int r = 0;
  while (x != null) {
    x = x.n; r++;   }
  return r;     }
```

```
00: iconst_0
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```



State F:

- Stored `x.n` to `x` (allowing for GC)

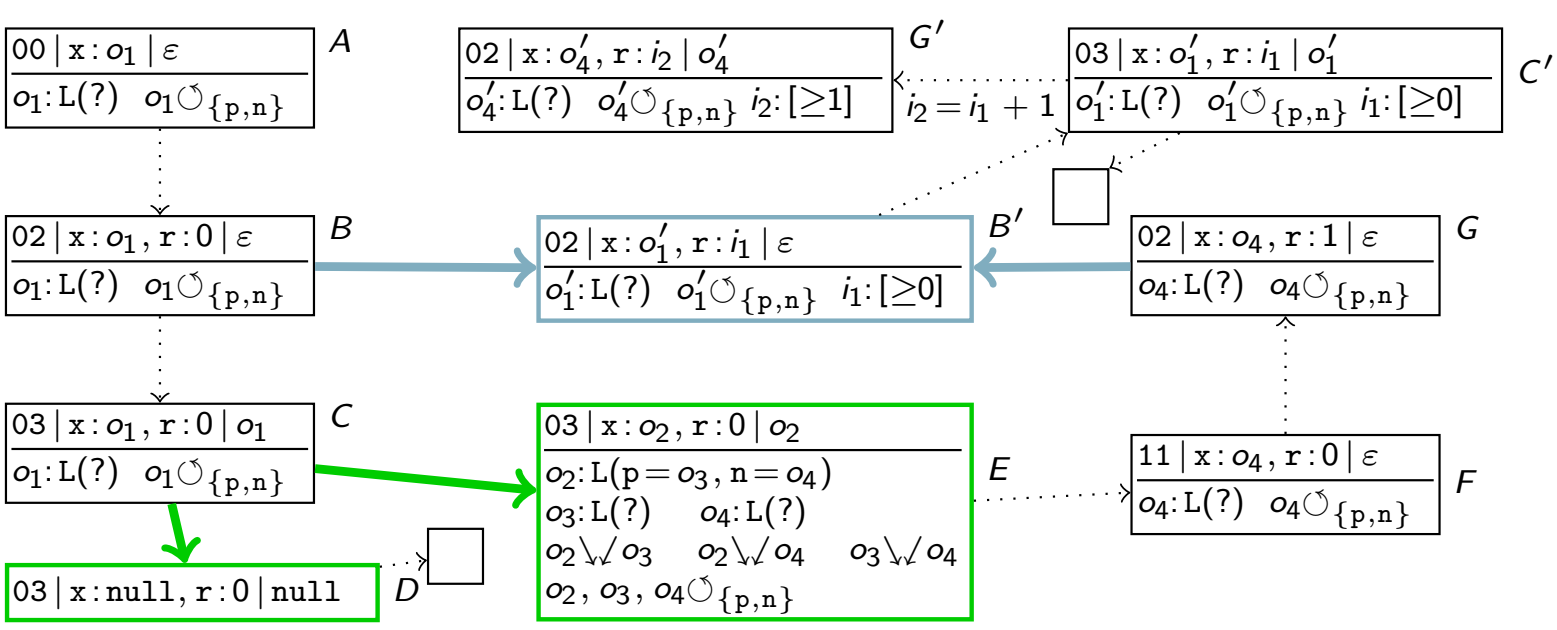
States G, B':

- Incremented `r`, back to position 02 (as B)

$\Rightarrow$ Generalization: "Merge" states B, G

```
int length(L x) {
  int r = 0;
  while (x != null) {
    x = x.n; r++;   }
  return r;     }
```

```
00: iconst_0
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```



State F:

- Stored `x.n` to `x` (allowing for GC)

States G, B':

- Incremented `r`, back to position 02 (as B)

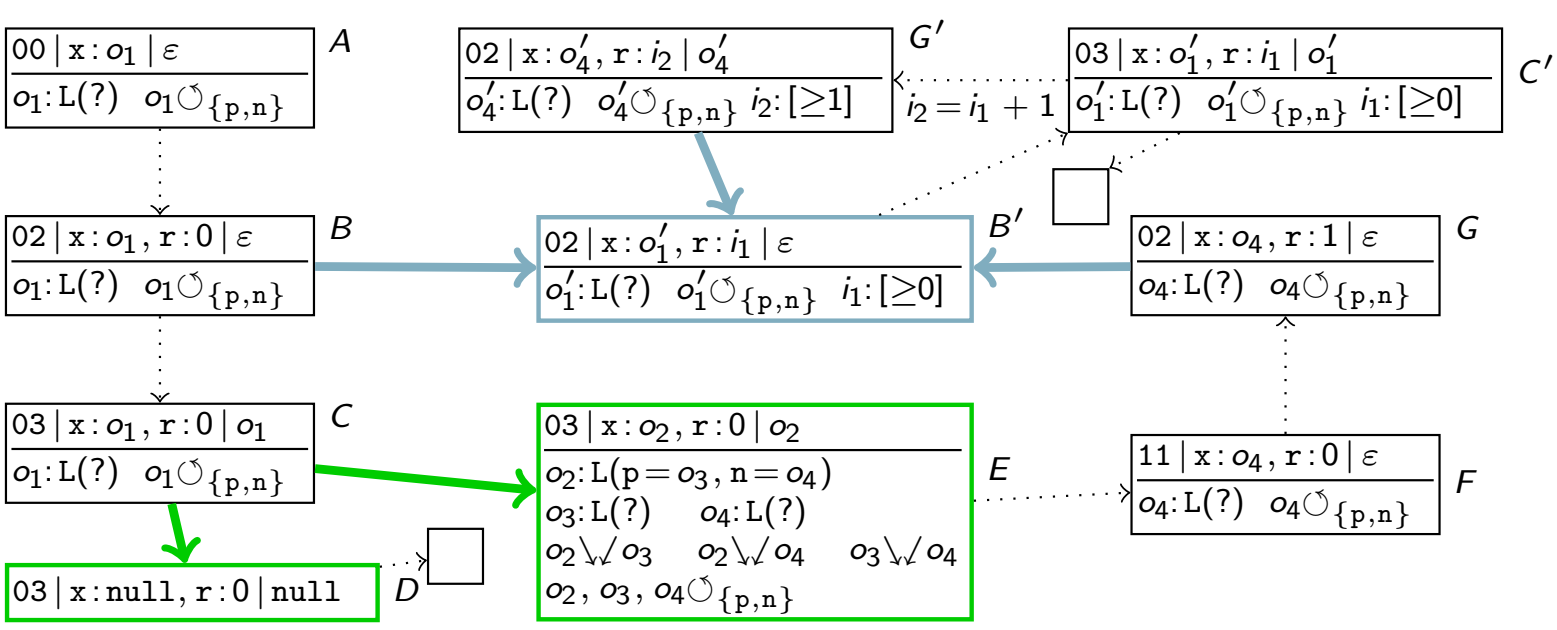
$\Rightarrow$ Generalization: "Merge" states B, G

States C', G':

- Repetition of C, G

```
int length(L x) {
  int r = 0;
  while (x != null) {
    x = x.n; r++;   }
  return r;       }
```

```
00: iconst_0
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```



State F:

- Stored `x.n` to `x` (allowing for GC)

States G, B':

- Incremented `r`, back to position 02 (as B)

$\Rightarrow$ Generalization: "Merge" states B, G

States C', G':

- Repetition of C, G

```
int length(L x) {
  int r = 0;
  while (x != null) {
    x = x.n; r++;   }
  return r;       }
```

Orientation: Term Rewriting

- Generalized Functional Programming

Orientation: Term Rewriting

- Generalized Functional Programming
- Rules $\mathcal{R}$ define rewrite relation:

$$\mathsf{app}(\mathsf{Cons}(x, xs), ys) \to \mathsf{Cons}(x, \mathsf{app}(xs, ys)) \quad (1)$$

$$\mathsf{app}(\mathsf{Nil}, ys) \to ys \quad (2)$$

- Rewriting of term t with rule $l \to r$:
  1. Find subterm s of t
  2. Find variable instantiation σ with $\sigma(l) = s$
  3. Result t' is t with s replaced by $\sigma(r)$

Orientation: Term Rewriting

- Generalized Functional Programming
- Rules $\mathcal{R}$ define rewrite relation:

$$\mathsf{app}(\mathsf{Cons}(x, xs), ys) \to \mathsf{Cons}(x, \mathsf{app}(xs, ys)) \quad (1)$$

$$\mathsf{app}(\mathsf{Nil}, ys) \to ys \quad (2)$$

- Rewriting of term t with rule $l \to r$:
  1. Find subterm s of t
  2. Find variable instantiation σ with $\sigma(l) = s$
  3. Result t' is t with s replaced by $\sigma(r)$

$$\underline{\mathsf{app}(\mathsf{Cons}(1, \mathsf{Nil}), \mathsf{Cons}(2, \mathsf{Nil}))} \quad \text{with } (1), x = 1, xs = \mathsf{Nil}, ys = \mathsf{Cons}(2, \mathsf{Nil})$$

$$\to \mathsf{Cons}(1, \underline{\mathsf{app}(\mathsf{Nil}, \mathsf{Cons}(2, \mathsf{Nil}))}) \quad \text{with } (2), ys = \mathsf{Cons}(2, \mathsf{Nil})$$

$$\to \mathsf{Cons}(1, \mathsf{Cons}(2, \mathsf{Nil}))$$

Transforming values to terms

$$\frac{03 \mid \mathtt{x} : o_2, \mathtt{r} : 0 \mid o_2}{\begin{array}{l} o_2 : \mathrm{L}(\mathtt{p} = o_3, \mathtt{n} = o_4) \\ o_3 : \mathrm{L}(?) \quad o_4 : \mathrm{L}(?) \\ o_2 \between o_3 \quad o_2 \between o_4 \quad o_3 \between o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \end{array}} \; E$$

Transforming values to terms

$$\frac{03 \mid \mathtt{x} : o_2, \mathtt{r} : 0 \mid o_2}{\begin{array}{l} o_2 : \mathrm{L}(\mathtt{p} = o_3, \mathtt{n} = o_4) \\ o_3 : \mathrm{L}(?) \quad o_4 : \mathrm{L}(?) \\ o_2 \between o_3 \quad o_2 \between o_4 \quad o_3 \between o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \end{array}} \; E$$

- Known integers transformed to themselves

Transforming values to terms

$$\frac{03 \mid \mathtt{x} : o_2, \mathtt{r} : 0 \mid o_2}{\begin{array}{l} o_2 : \mathrm{L}(\mathtt{p} = o_3, \mathtt{n} = o_4) \\ o_3 : \mathrm{L}(?) \quad o_4 : \mathrm{L}(?) \\ o_2 \diagdown\!\!\diagup o_3 \quad o_2 \diagdown\!\!\diagup o_4 \quad o_3 \diagdown\!\!\diagup o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \end{array}} \; E$$

- Known integers transformed to themselves
- Unknown values transformed to variables

$o_3, o_4 \quad 0$

Transforming values to terms

$$\frac{03 \mid \mathtt{x}:o_2, \mathtt{r}:0 \mid o_2}{\begin{array}{l} o_2{:}\,\mathtt{L}(\mathtt{p}=o_3, \mathtt{n}=o_4) \\ o_3{:}\,\mathtt{L}(?) \quad o_4{:}\,\mathtt{L}(?) \\ o_2 \not\perp o_3 \quad o_2 \not\perp o_4 \quad o_3 \not\perp o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \end{array}} \; E$$

- Known integers transformed to themselves
- Unknown values transformed to variables
- Data structures transformed to nested constructor terms:
  Class `Cl` with n fields $\curvearrowright$ symbol Cl of arity n

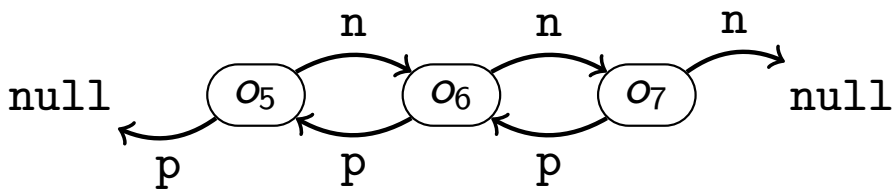
$$\overbrace{\mathsf{L}(o_3, o_4)}^{o_2}\ 0$$

Transforming values to terms

$$\frac{03 \mid \mathtt{x}:o_2, \mathtt{r}:0 \mid o_2}{\begin{array}{l} o_2{:}\,\mathrm{L}(\mathrm{p}=o_3, \mathrm{n}=o_4) \\ o_3{:}\,\mathrm{L}(?) \quad o_4{:}\,\mathrm{L}(?) \\ o_2 \not\downarrow o_3 \quad o_2 \not\downarrow o_4 \quad o_3 \not\downarrow o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathrm{p},\mathrm{n}\}} \end{array}} \; E$$

- Known integers transformed to themselves
- Unknown values transformed to variables
- Data structures transformed to nested constructor terms:
  Class `Cl` with n fields $\curvearrowright$ symbol Cl of arity n
- Encoding cycles: Special symbol $\bigcirc$ for repetition

$o_5{:}\,\mathrm{L}(\mathrm{p}=\mathtt{null}, \mathrm{n}=o_6)$
$o_6{:}\,\mathrm{L}(\mathrm{p}=o_5, \mathrm{n}=o_7)$
$o_7{:}\,\mathrm{L}(\mathrm{p}=o_6, \mathrm{n}=\mathtt{null})$



Encoding of o_5: $\mathrm{L}(\mathtt{null}, \mathrm{L}(\bigcirc, \mathrm{L}(\bigcirc, \mathtt{null})))$
Encoding of o_6: $\mathrm{L}(\mathrm{L}(\mathtt{null}, \bigcirc), \mathrm{L}(\bigcirc, \mathtt{null}))$

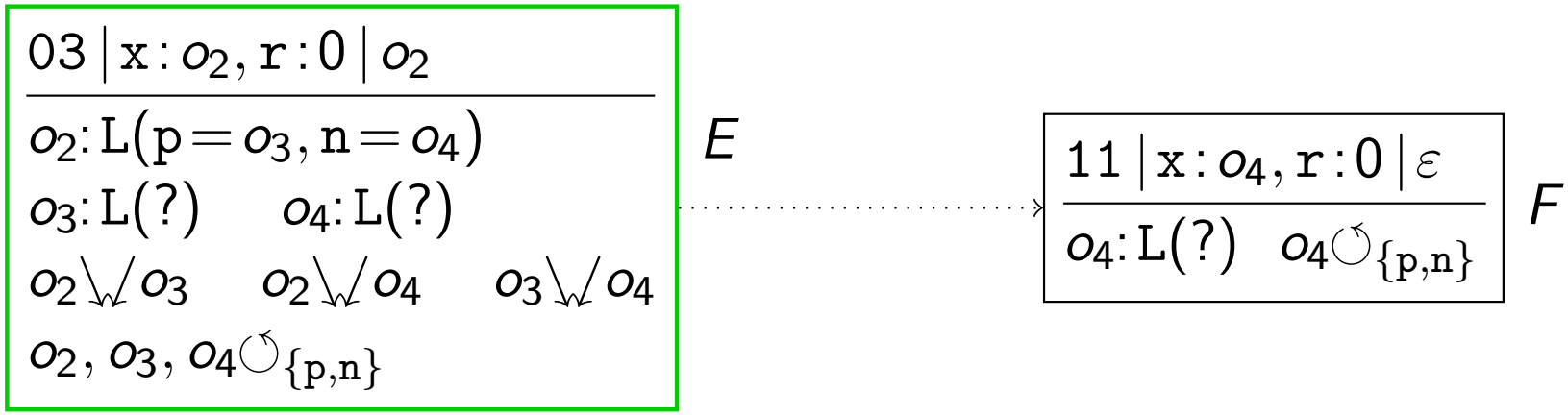
Transforming states to terms

$$\frac{03 \mid \mathtt{x} : o_2, \mathtt{r} : 0 \mid o_2}{\begin{array}{l} o_2 : \mathrm{L}(\mathtt{p} = o_3, \mathtt{n} = o_4) \\ o_3 : \mathrm{L}(?) \quad o_4 : \mathrm{L}(?) \\ o_2 \not\perp o_3 \quad o_2 \not\perp o_4 \quad o_3 \not\perp o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \end{array}} \; E$$

- State s term encoding:
  - Root symbol ($\equiv$ program position) f_s
  - All local variables, stack entries as arguments

$$\mathrm{f}_E(\overbrace{\mathrm{L}(o_3, o_4)}^{o_2}, 0, \overbrace{\mathrm{L}(o_3, o_4)}^{o_2})$$

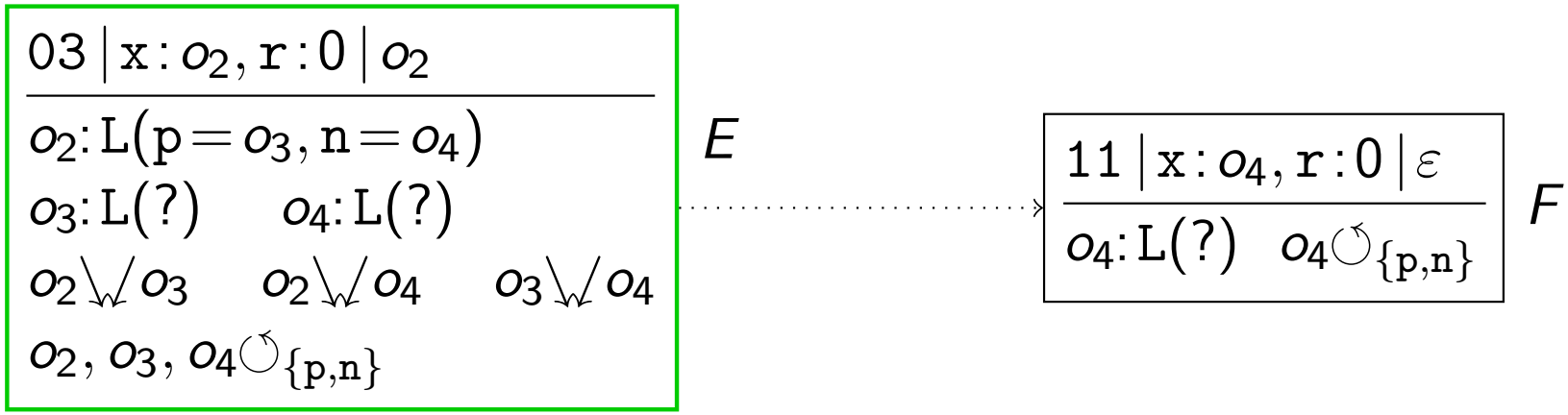
Transforming edges to rules



- State s term encoding:
  - Root symbol ($\equiv$ program position) f_s
  - All local variables, stack entries as arguments
- Evaluation edges: Encode states, put in $\rightarrow$

$$\mathsf{f}_E(\overbrace{\mathtt{L}(o_3, o_4)}^{o_2}, 0, \overbrace{\mathtt{L}(o_3, o_4)}^{o_2})$$

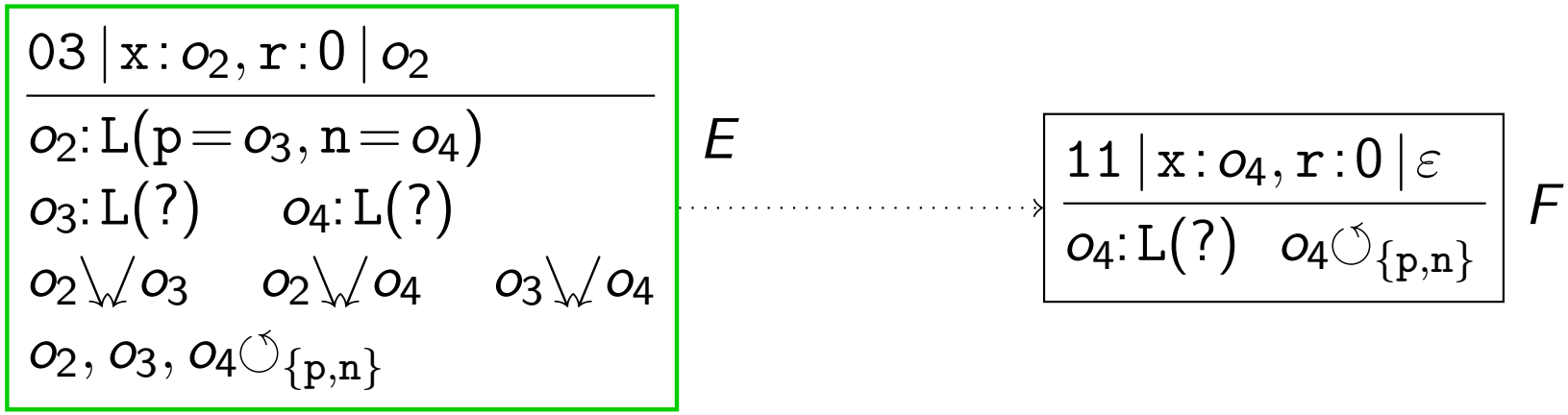
Transforming edges to rules



- State s term encoding:
  - Root symbol ($\equiv$ program position) f_s
  - All local variables, stack entries as arguments
- Evaluation edges: Encode states, put in $\to$
  - **Problem**: Cycle encoding changes $\curvearrowright$ free var on rhs
  - **Solution**: Filter: Only encode non-cyclic parts!

$$\mathsf{f}_E(\overbrace{\mathrm{L}(o_3, o_4)}^{o_2}, 0, \overbrace{\mathrm{L}(o_3, o_4)}^{o_2}) \quad \to \quad \mathsf{f}_F(o_4\ , 0)$$

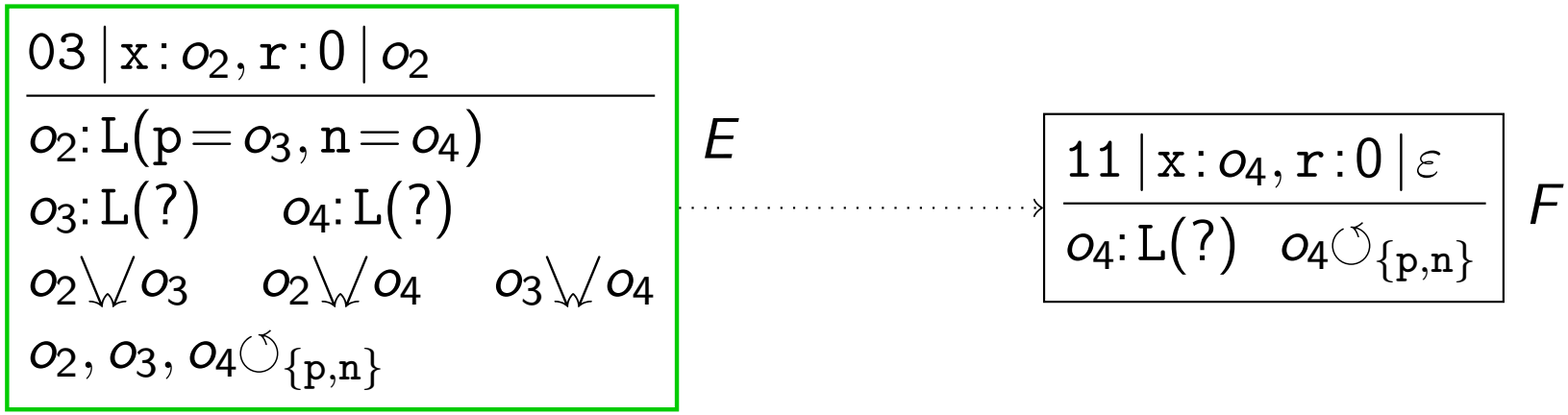
Transforming edges to rules



- State s term encoding:
  - Root symbol ($\equiv$ program position) f_s
  - All local variables, stack entries as arguments
- Evaluation edges: Encode states, put in $\rightarrow$
  - **Problem**: Cycle encoding changes $\curvearrowright$ free var on rhs
  - **Solution**: Filter: Only encode non-cyclic parts!
- Refinement edges: Encode target state twice, relabel

$$\mathsf{f}_E(\overbrace{\mathsf{L}(o_3, o_4)}^{o_2}, 0, \overbrace{\mathsf{L}(o_3, o_4)}^{o_2}) \quad \rightarrow \quad \mathsf{f}_F(o_4', 0)$$

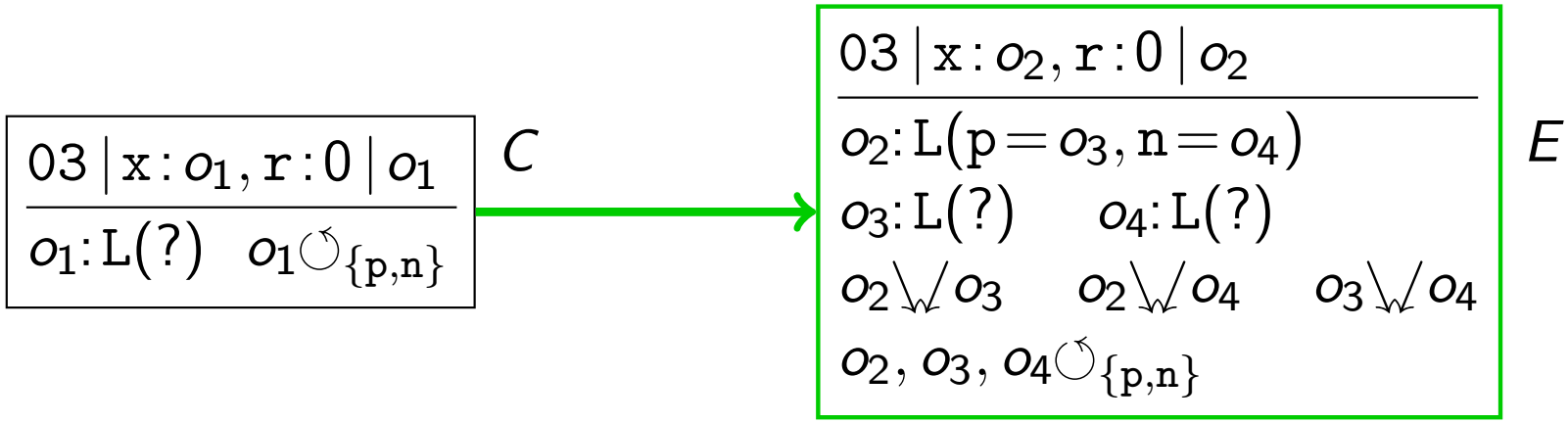
Transforming edges to rules



- State s term encoding:
  - Root symbol ($\equiv$ program position) f_s
  - All local variables, stack entries as arguments
- Evaluation edges: Encode states, put in $\rightarrow$
  - **Problem**: Cycle encoding changes $\curvearrowright$ free var on rhs
  - **Solution**: Filter: Only encode non-cyclic parts!
- Refinement edges: Encode target state twice, relabel
- Instantiation edges: Encode source state twice, relabel

$$\mathsf{f}_E(\overbrace{\mathrm{L}(\quad o_4)}^{o_2}, 0, \overbrace{\mathrm{L}(\quad o_4)}^{o_2}) \quad \rightarrow \quad \mathsf{f}_F(o_4\ , 0)$$

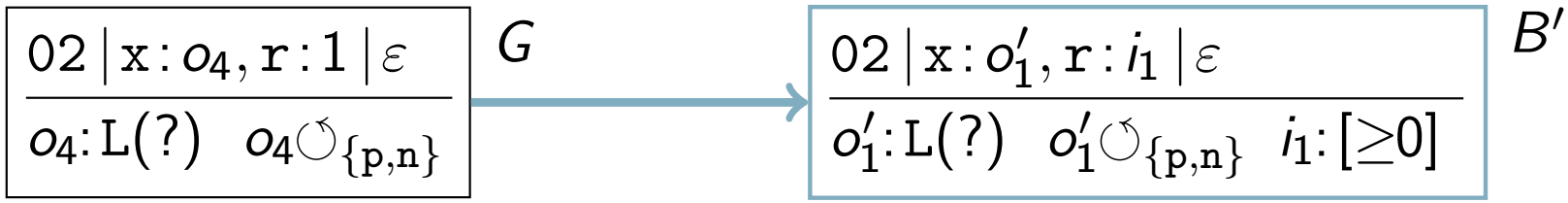
Transforming edges to rules



- State s term encoding:
  - Root symbol ($\equiv$ program position) f_s
  - All local variables, stack entries as arguments
- Evaluation edges: Encode states, put in $\rightarrow$
  - **Problem**: Cycle encoding changes $\curvearrowright$ free var on rhs
  - **Solution**: Filter: Only encode non-cyclic parts!
- Refinement edges: Encode target state twice, relabel
- Instantiation edges: Encode source state twice, relabel

$$\mathrm{f}_C(\mathrm{L}(o_4), 0, \mathrm{L}(o_4)) \quad \rightarrow \quad \mathrm{f}_E(\mathrm{L}(o_4), 0, \mathrm{L}(o_4))$$

Transforming edges to rules



- State s term encoding:
  - Root symbol ($\equiv$ program position) f_s
  - All local variables, stack entries as arguments
- Evaluation edges: Encode states, put in $\rightarrow$
  - **Problem**: Cycle encoding changes $\curvearrowright$ free var on rhs
  - **Solution**: Filter: Only encode non-cyclic parts!
- Refinement edges: Encode target state twice, relabel
- Instantiation edges: Encode source state twice, relabel

$$\mathsf{f}_G(o_4, 2) \;\rightarrow\; \mathsf{f}_{B'}(o_4, 2)$$

Orientation: Polynomial Orders

- Function symbols interpreted as polynomials over $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{N}^n, \ldots$

Orientation: Polynomial Orders

- Function symbols interpreted as polynomials over $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{N}^n, \ldots$
- Extension to terms: $[\![\mathsf{f}(t_1, \ldots, t_n)]\!] = [\![\mathsf{f}]\!]([\![t_1]\!], \ldots, [\![t_n]\!])$
- Termination proof: For $l \to r$ prove $\exists c. [\![l]\!] > [\![r]\!] \wedge [\![l]\!] \geq c$

Orientation: Polynomial Orders

- Function symbols interpreted as polynomials over $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{N}^n, \ldots$
- Extension to terms: $[\![\mathsf{f}(t_1, \ldots, t_n)]\!] = [\![\mathsf{f}]\!]([\![t_1]\!], \ldots, [\![t_n]\!])$
- Termination proof: For $l \to r$ prove $\exists c. [\![l]\!] > [\![r]\!] \wedge [\![l]\!] \geq c$

Rule: $\quad \mathsf{app}(\mathsf{Cons}(x, xs), ys) \to \mathsf{Cons}(x, \mathsf{app}(xs, ys))$

Choose $[\![\mathsf{app}]\!] = (x, y) \mapsto 1 + 2 \cdot x$, $[\![\mathsf{Cons}]\!] = (x, y) \mapsto 1 + y$,

Orientation: Polynomial Orders

- Function symbols interpreted as polynomials over $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{N}^n, \ldots$
- Extension to terms: $[\![\mathsf{f}(t_1, \ldots, t_n)]\!] = [\![\mathsf{f}]\!]([\![t_1]\!], \ldots, [\![t_n]\!])$
- Termination proof: For $l \to r$ prove $\exists c.[\![l]\!] > [\![r]\!] \wedge [\![l]\!] \geq c$

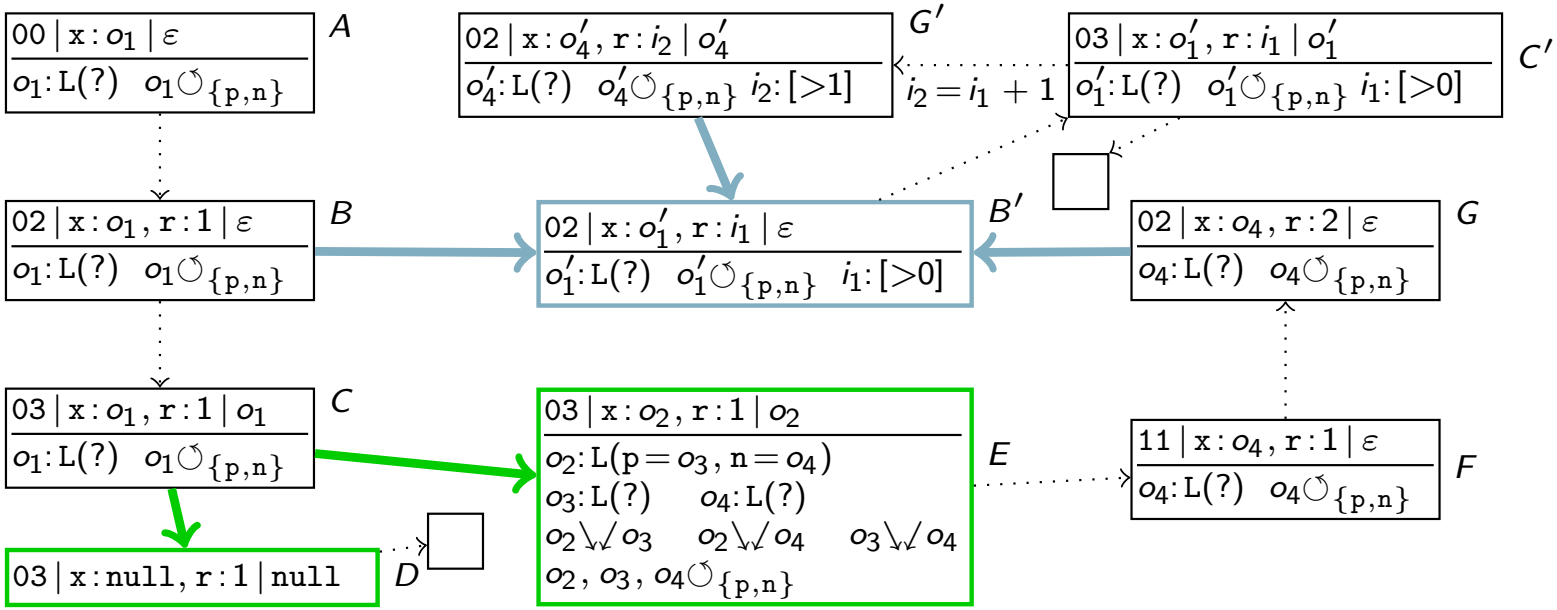
$$\text{Rule:} \quad \mathsf{app}(\mathsf{Cons}(x, xs), ys) \to \mathsf{Cons}(x, \mathsf{app}(xs, ys))$$

$$\text{Interpretation:} \quad 1 + 2 + 2 \cdot xs > 1 + 1 + 2 \cdot xs$$

Choose $[\![\mathsf{app}]\!] = (x, y) \mapsto 1 + 2 \cdot x$, $[\![\mathsf{Cons}]\!] = (x, y) \mapsto 1 + y$,

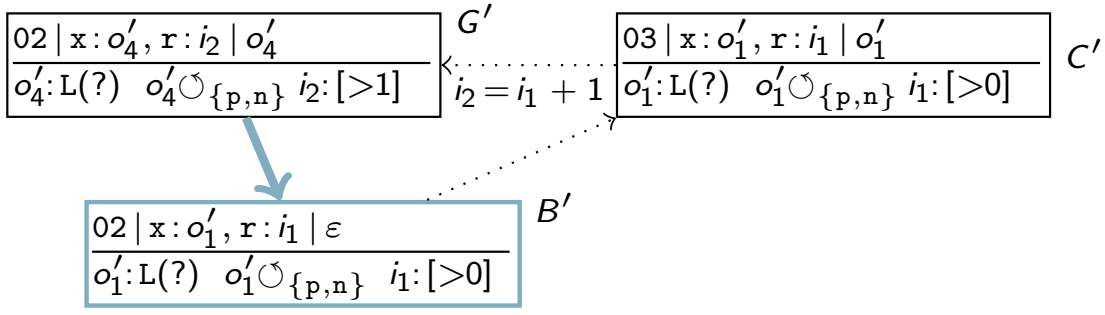
The example TRS

```
00: iconst_1
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```



The example TRS

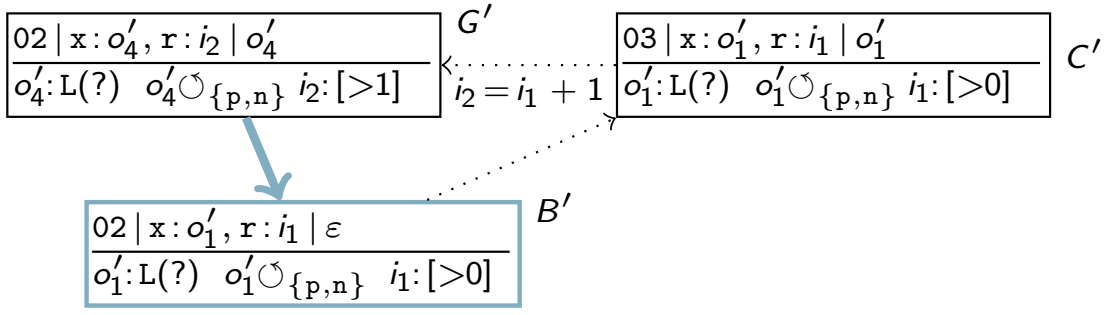
```
00: iconst_1
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```



1. Only consider SCCs!

The example TRS

```
00: iconst_1
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```

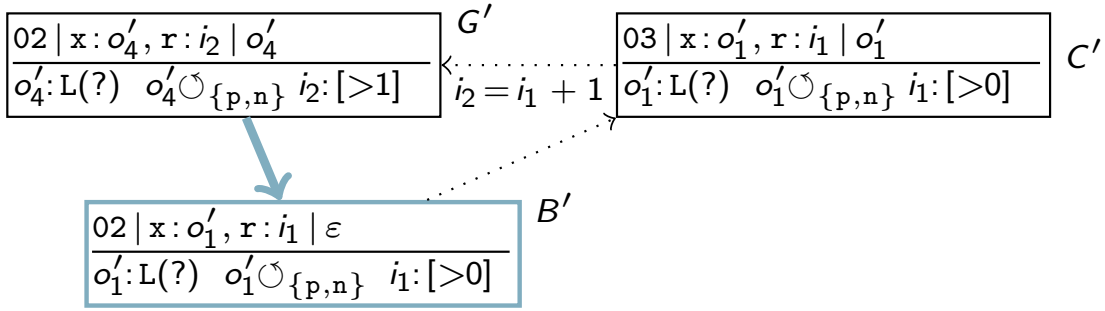


1. Only consider SCCs!
2. Transform all edges as before, simplify:

$$\mathsf{f}_{B'}(\mathrm{L}(o'_4), i_1) \quad \to \quad \mathsf{f}_{B'}(o'_4, i_1 + 1)$$

The example TRS

```
00: iconst_1
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```



1. Only consider SCCs!
2. Transform all edges as before, simplify:

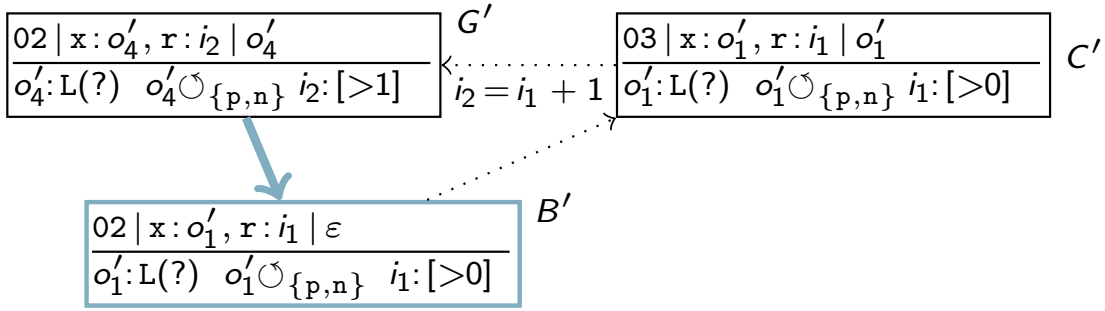
$$\mathsf{f}_{B'}(\mathsf{L}(o_4'), i_1) \quad \to \quad \mathsf{f}_{B'}(o_4', i_1 + 1)$$

3. Termination trivially proven with

$$[\![\mathsf{f}_{B'}]\!] = (x_1, x_2) \mapsto x_1$$
$$[\![\mathsf{L}]\!] = (x_1) \mapsto x_1 + 1$$

The example TRS

```
00: iconst_1
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```



1. Only consider SCCs!
2. Transform all edges as before, simplify:

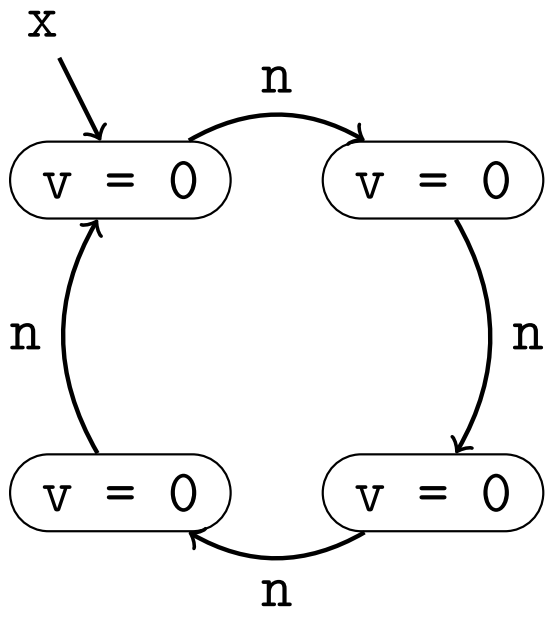
$$\begin{array}{rcl} \mathsf{f}_{B'}(\mathsf{L}(o_4'), i_1) & \to & \mathsf{f}_{B'}(o_4', i_1 + 1) \\ o_4' + 1 & > & o_4' \end{array}$$

3. Termination trivially proven with

$$[\![\mathsf{f}_{B'}]\!] = (x_1, x_2) \mapsto x_1$$
$$[\![\mathsf{L}]\!] = (x_1) \mapsto x_1 + 1$$

visit: the example

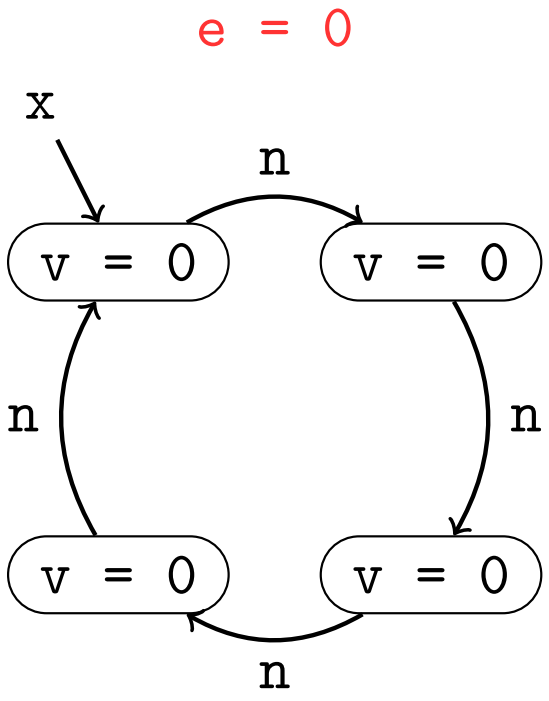
```
class L {
  int v;    L n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```



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

visit: the example

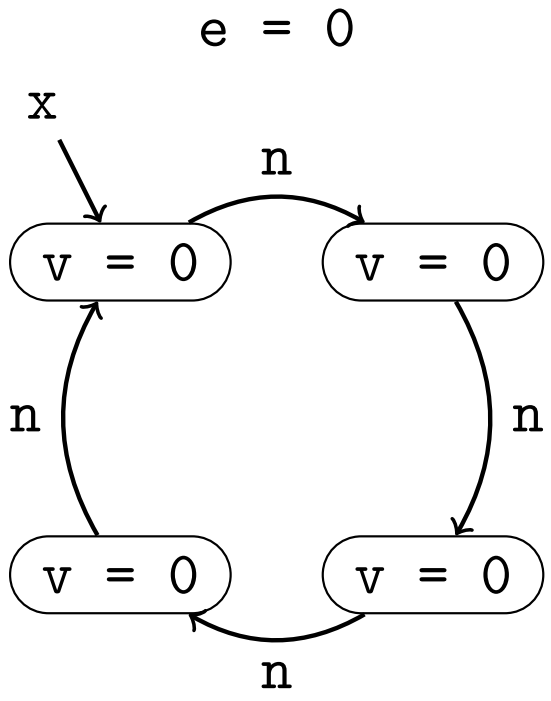
```
class L {
  int v;    L n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```



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

visit: the example

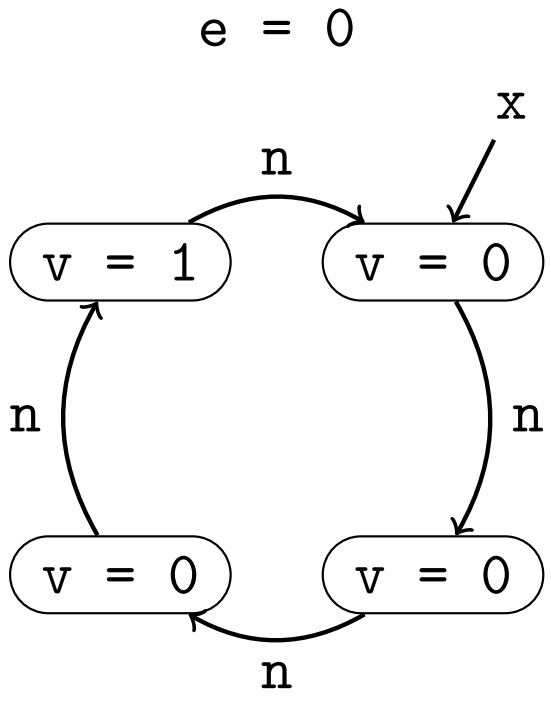
```
class L {
  int v;    L n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```



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

visit: the example

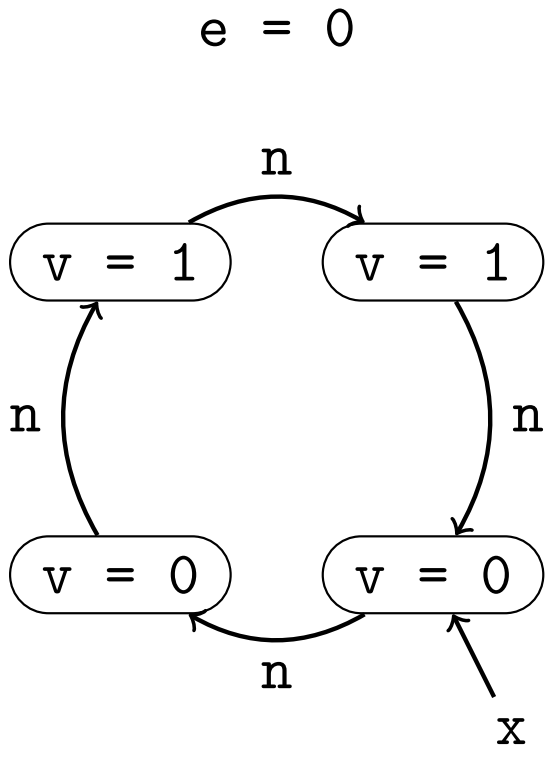
```
class L {
  int v;    L n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```



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

visit: the example

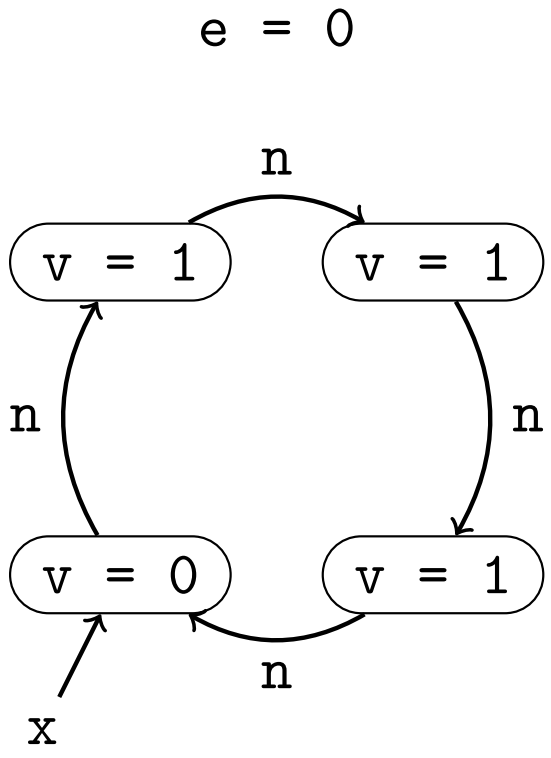
```
class L {
  int v;    L n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```



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

visit: the example

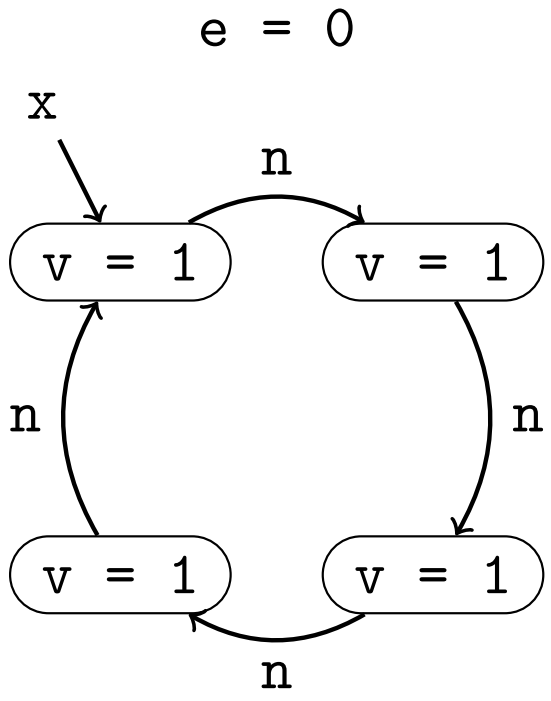
```
class L {
  int v;    L n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```



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

visit: the example

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

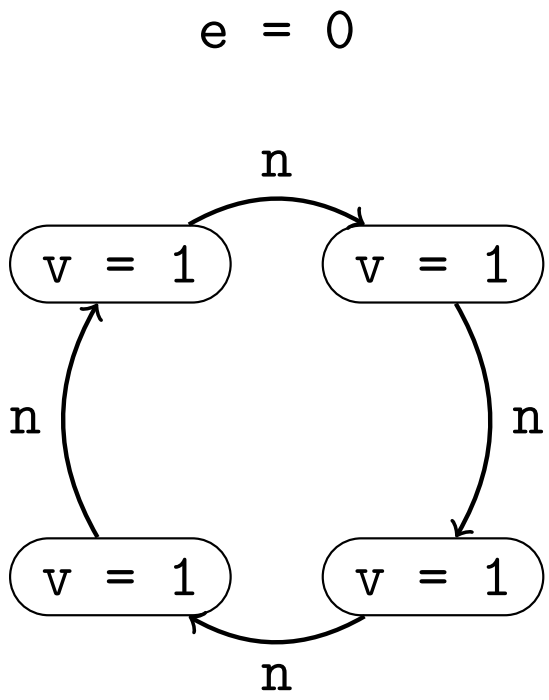


1. Store first v
2. Continue if object unvisited
3. Change v
4. Go to next element

visit: the example

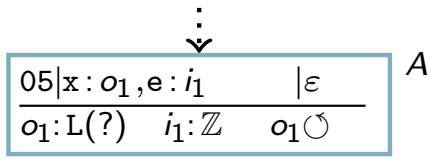
```
class L {
  int v;    L n;
  static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
      x.v = e + 1;
      x = x.n; }}}
```

```
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
```



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

```
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
```



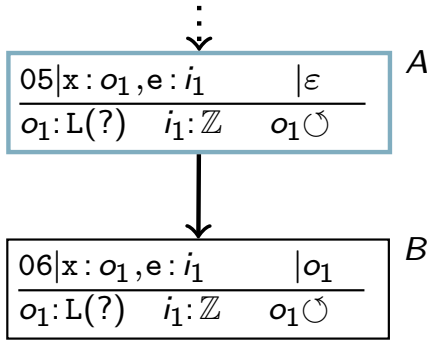
A

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
```

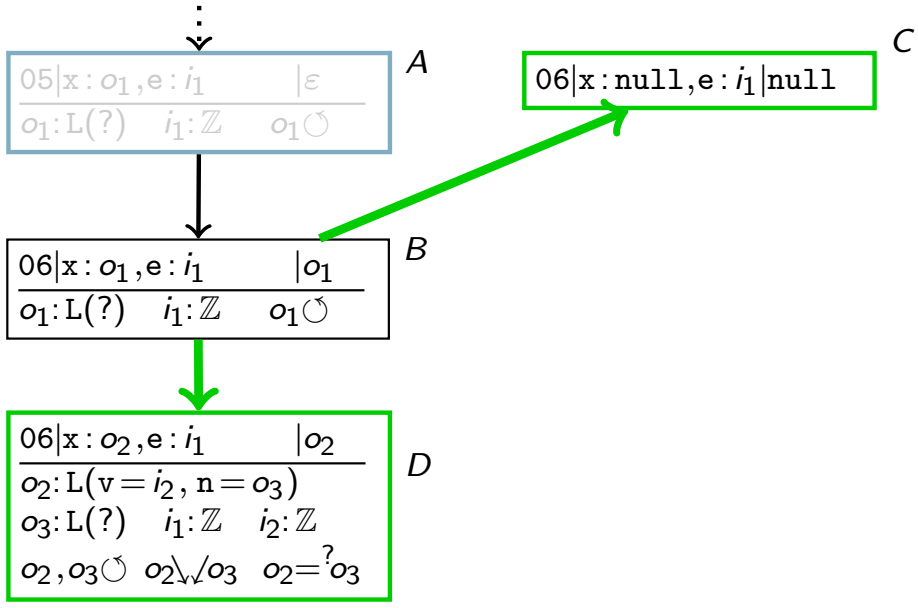


State *B*:

- *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
```

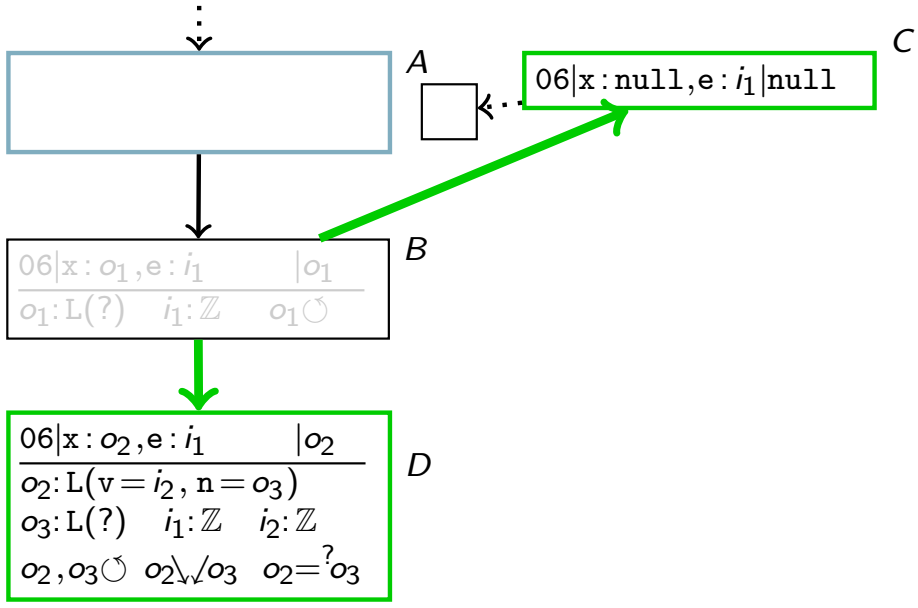



```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

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 share with 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
```

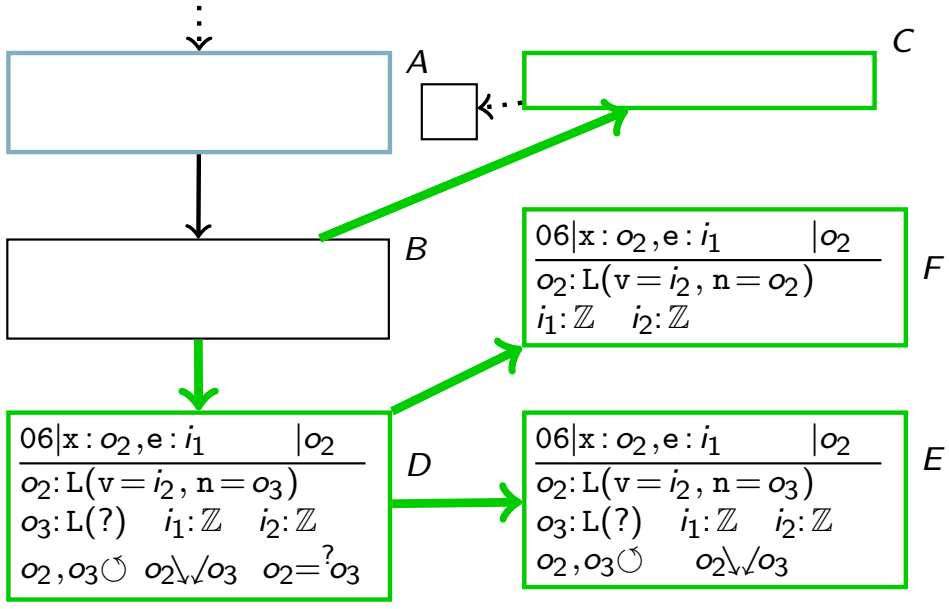



```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

States B, C, D:

- *Evaluation* between A and B
- Need field of o_1 $\Rightarrow$ 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 share with 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
```

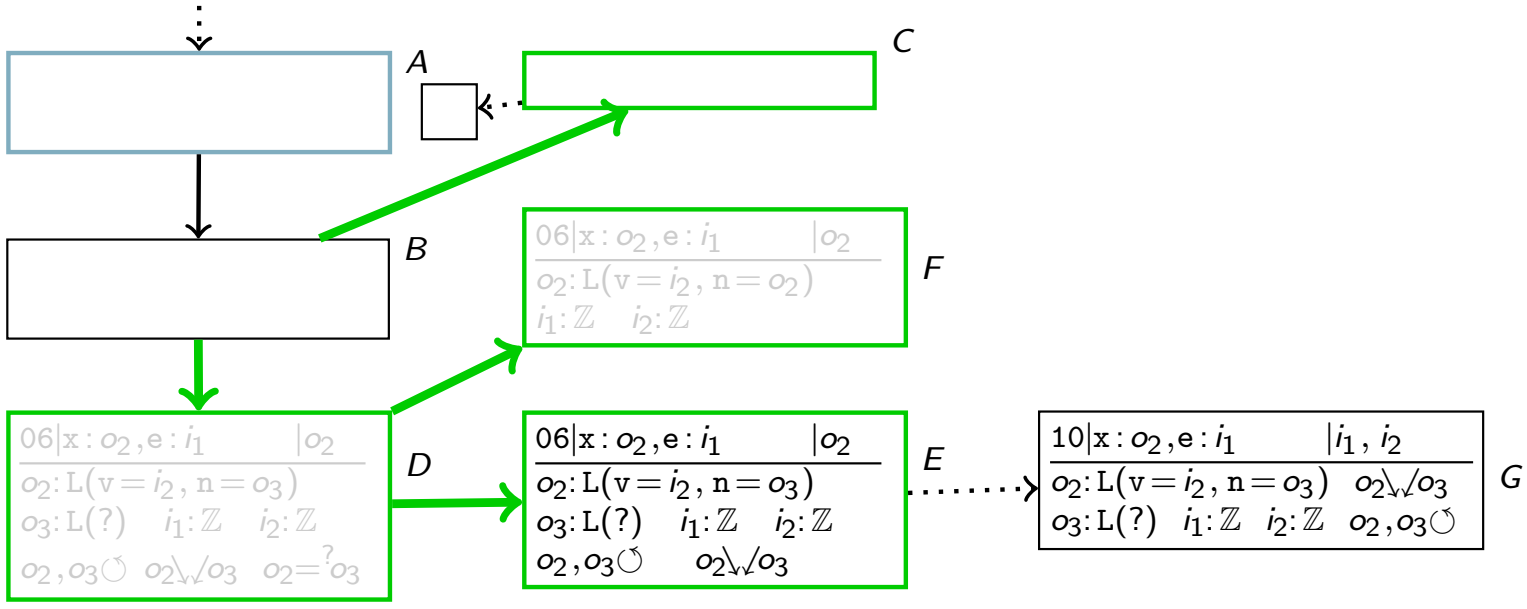


States E, F:

- 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
```

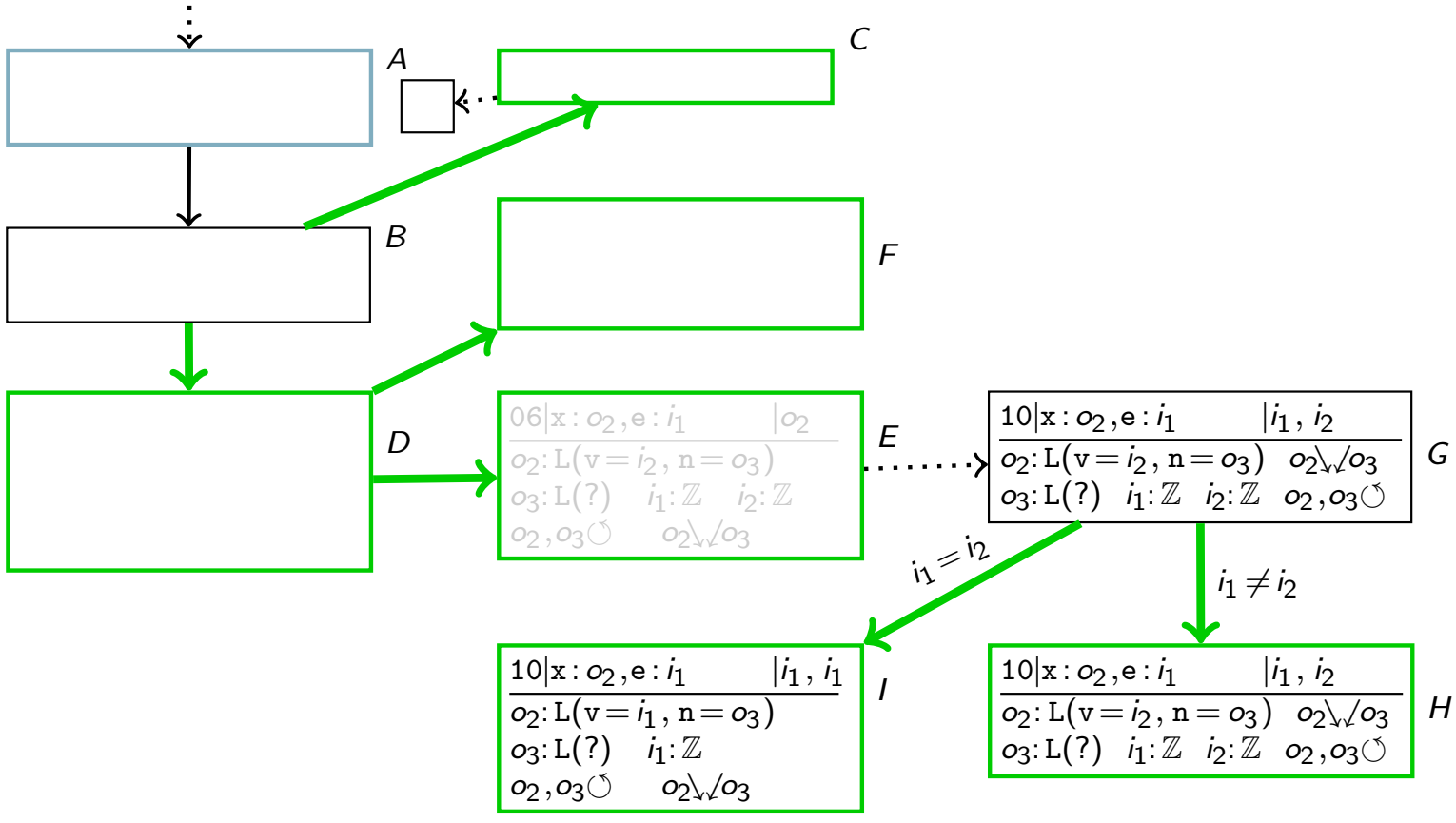


State G:

- 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
```

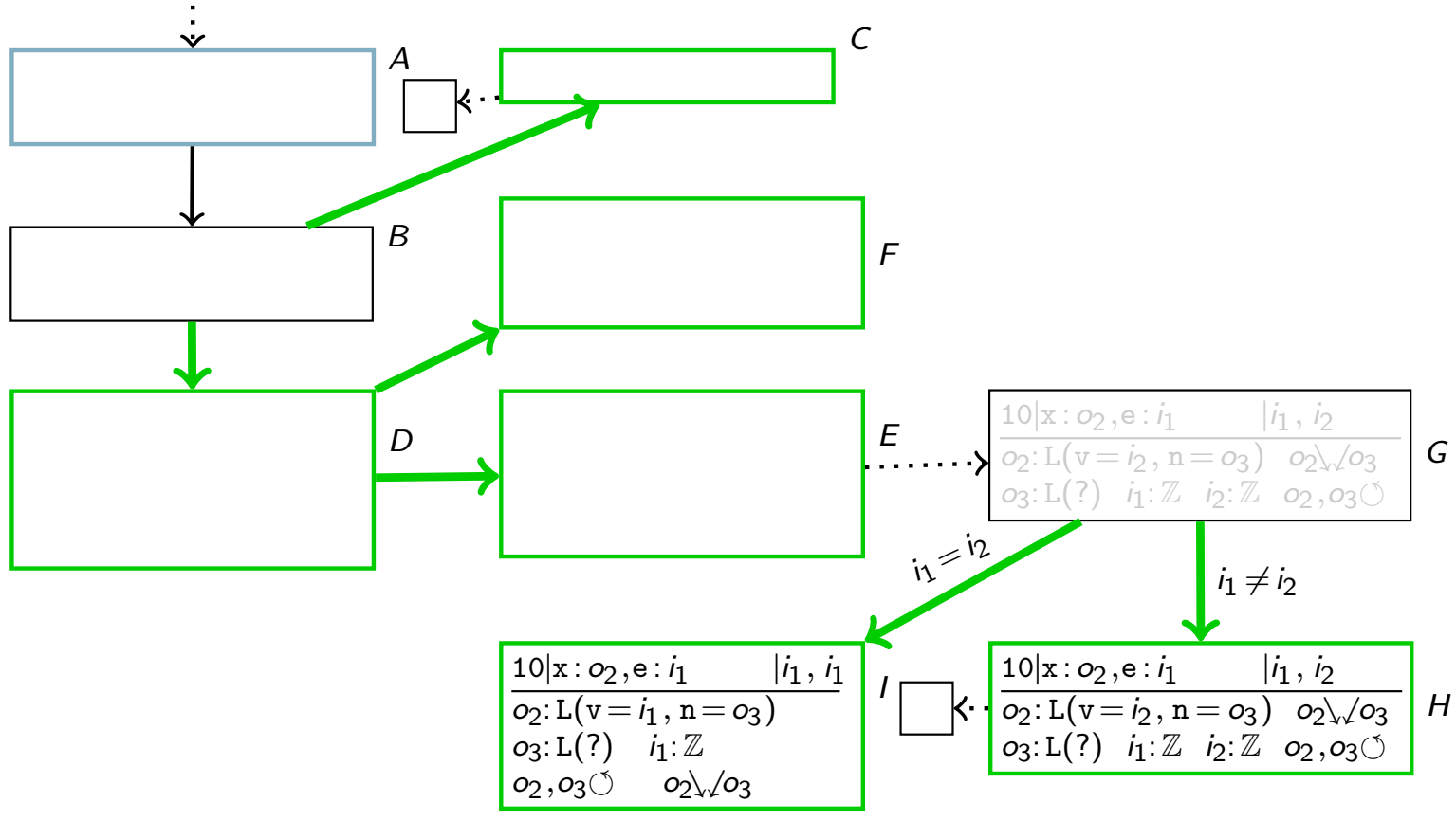



```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

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
```

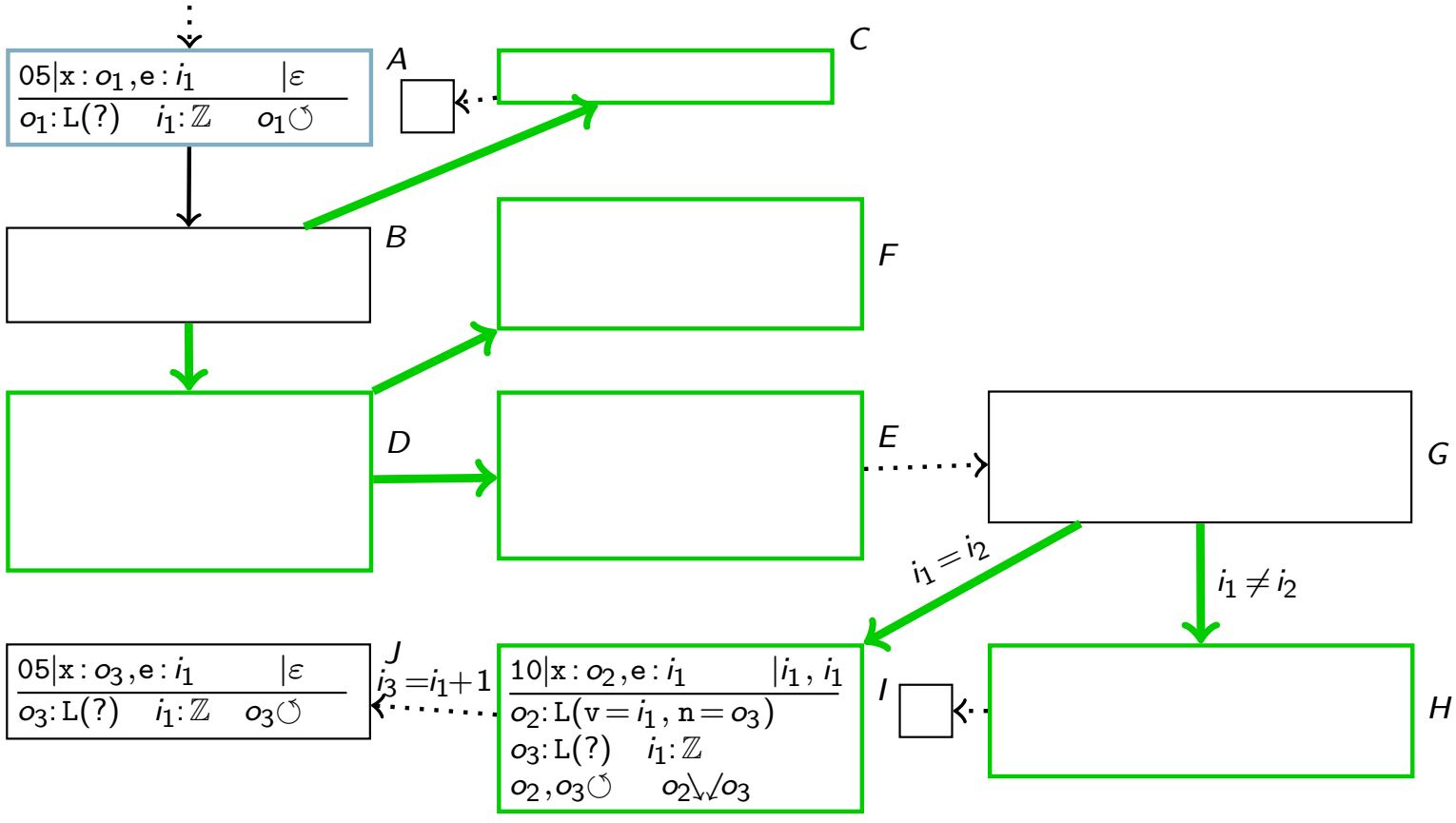



```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

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
```

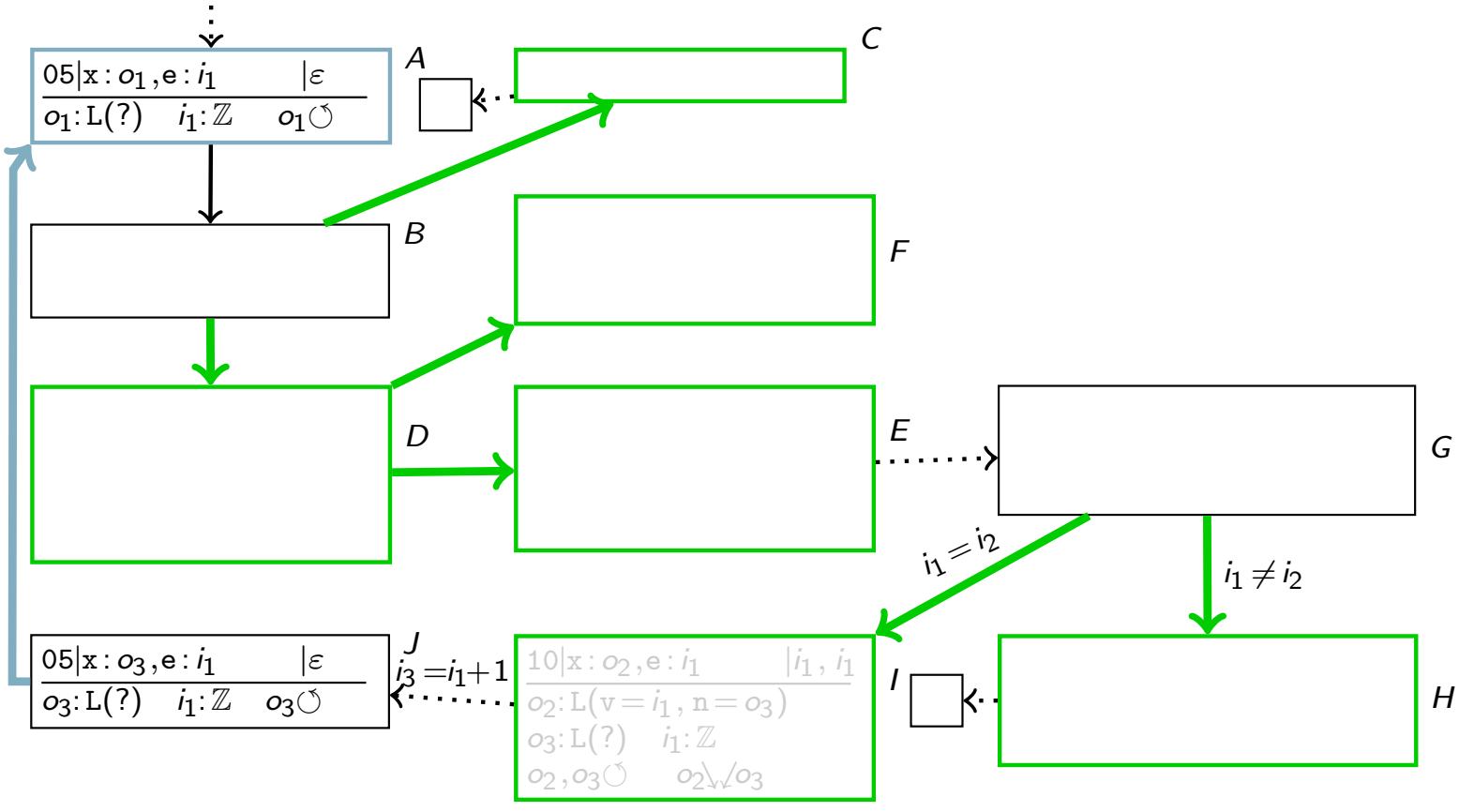



```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

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

```
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
```

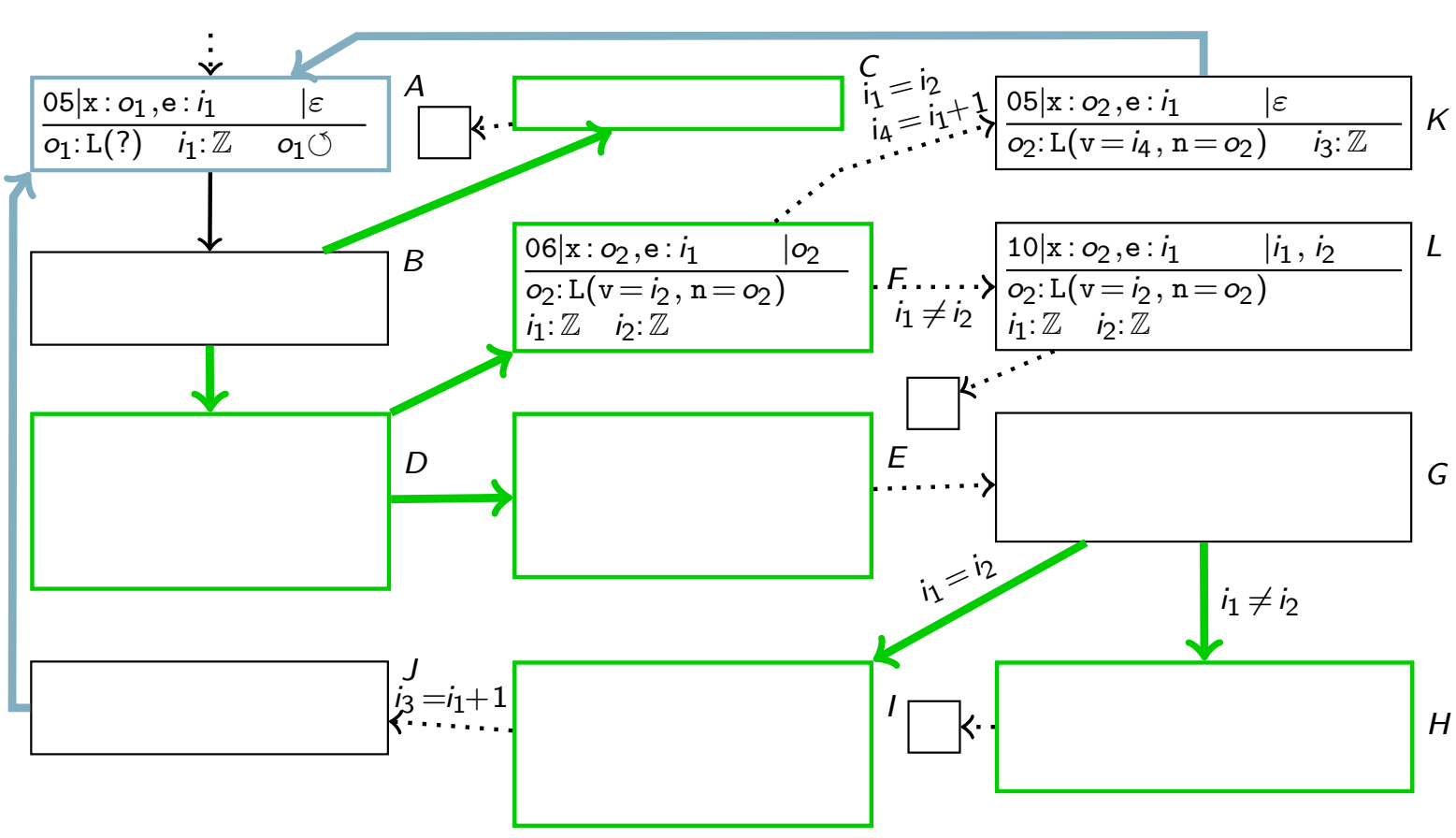



```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

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

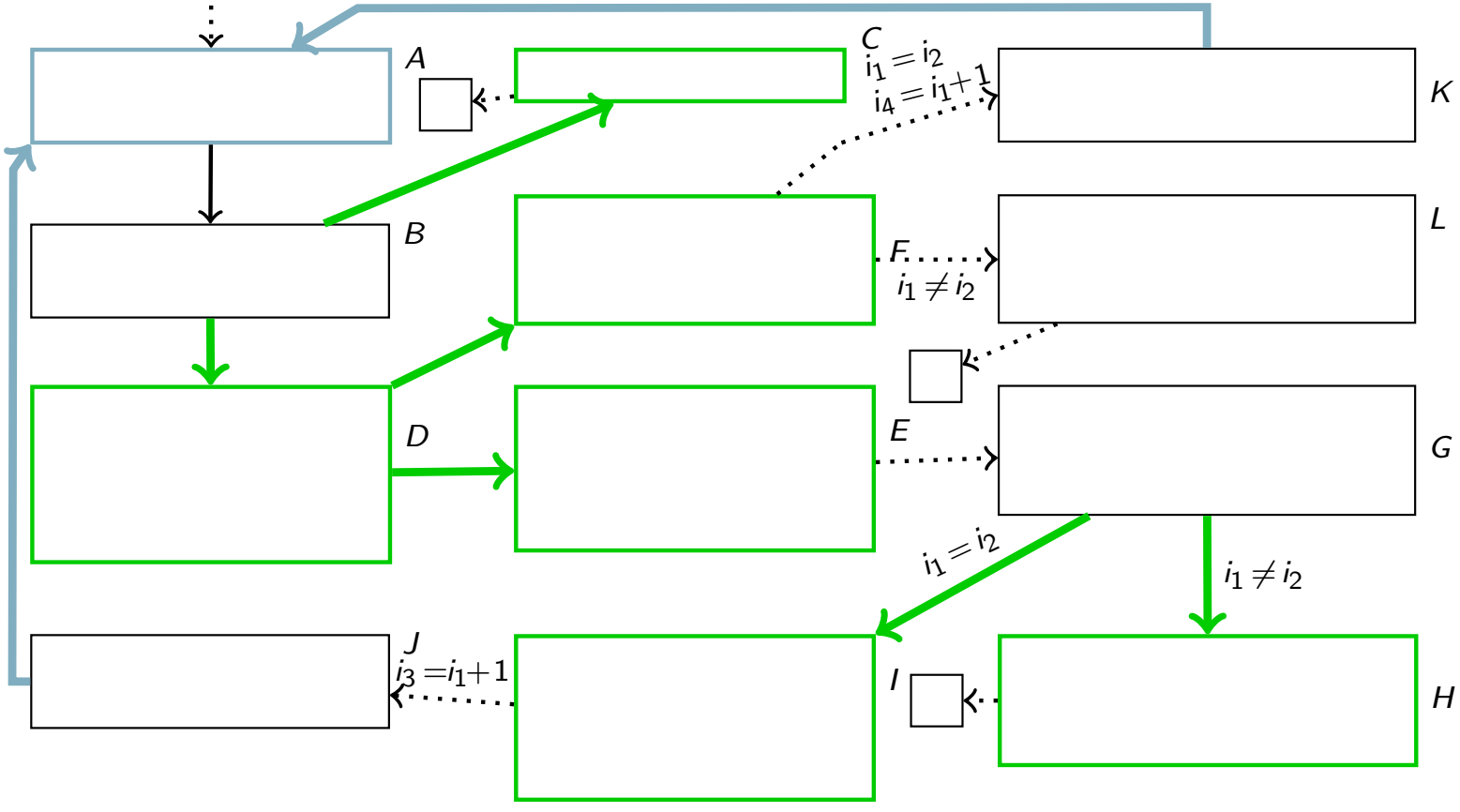
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



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; }}
```

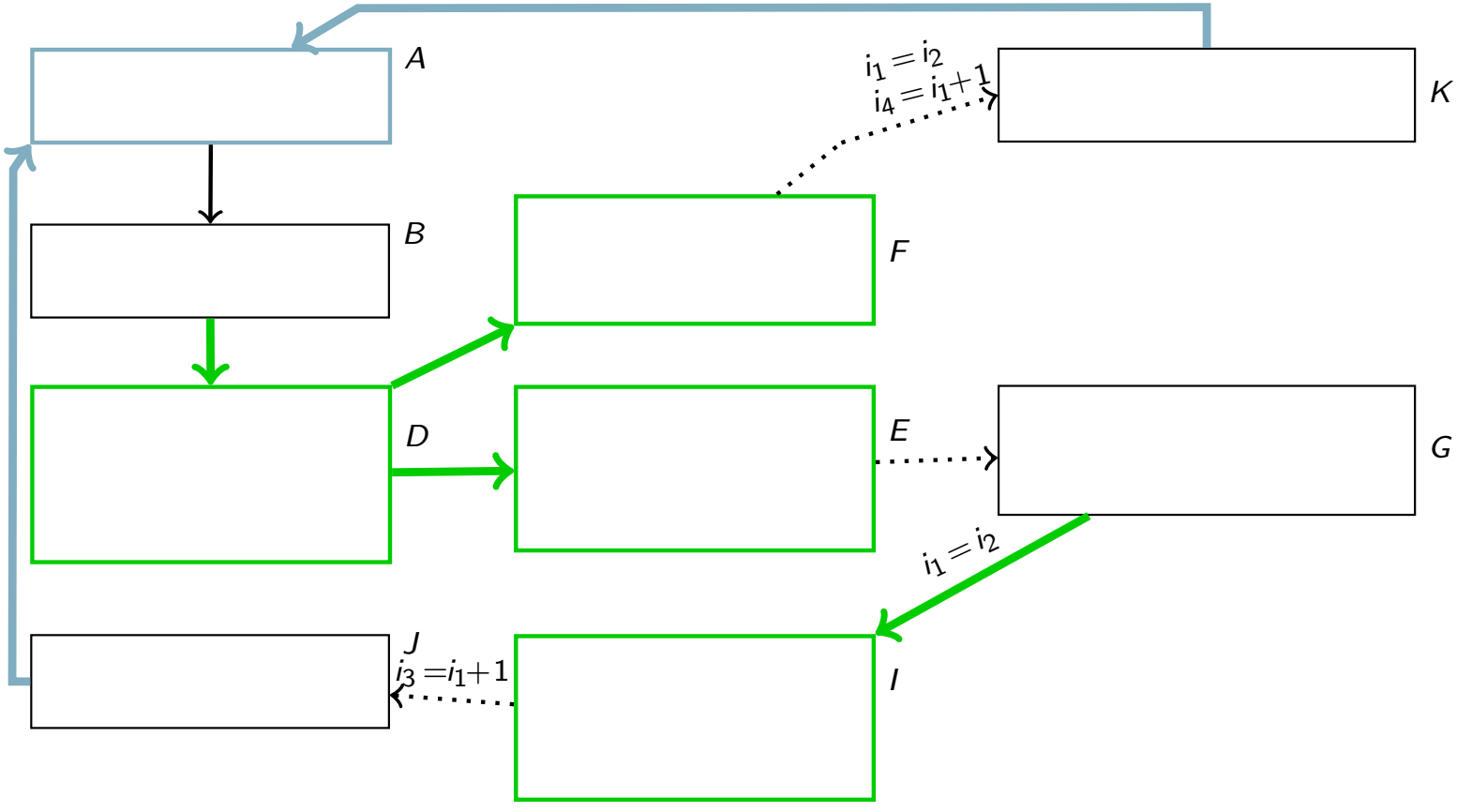
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



- All leaves program ends ⇒ 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; }}
```

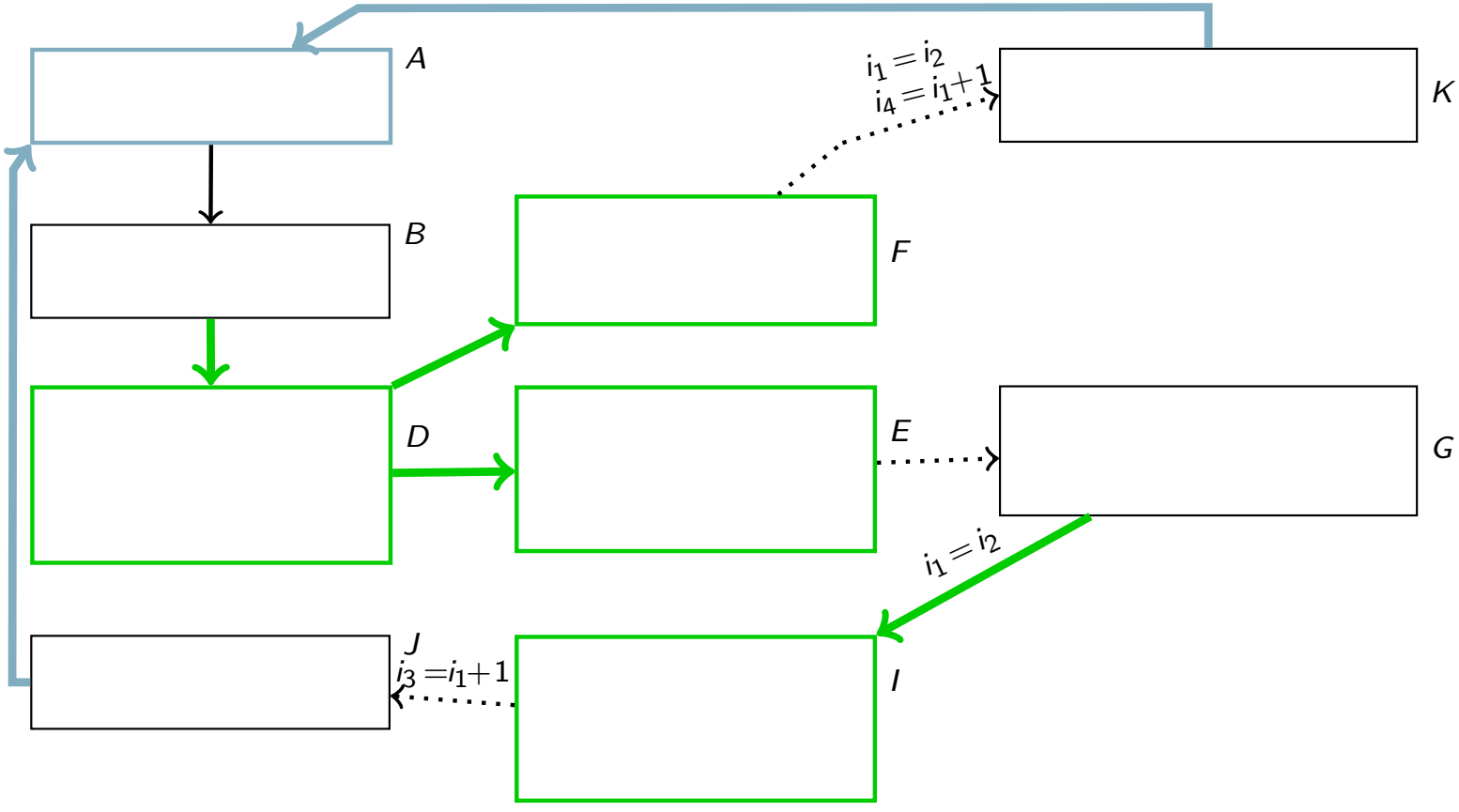
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



- 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
```

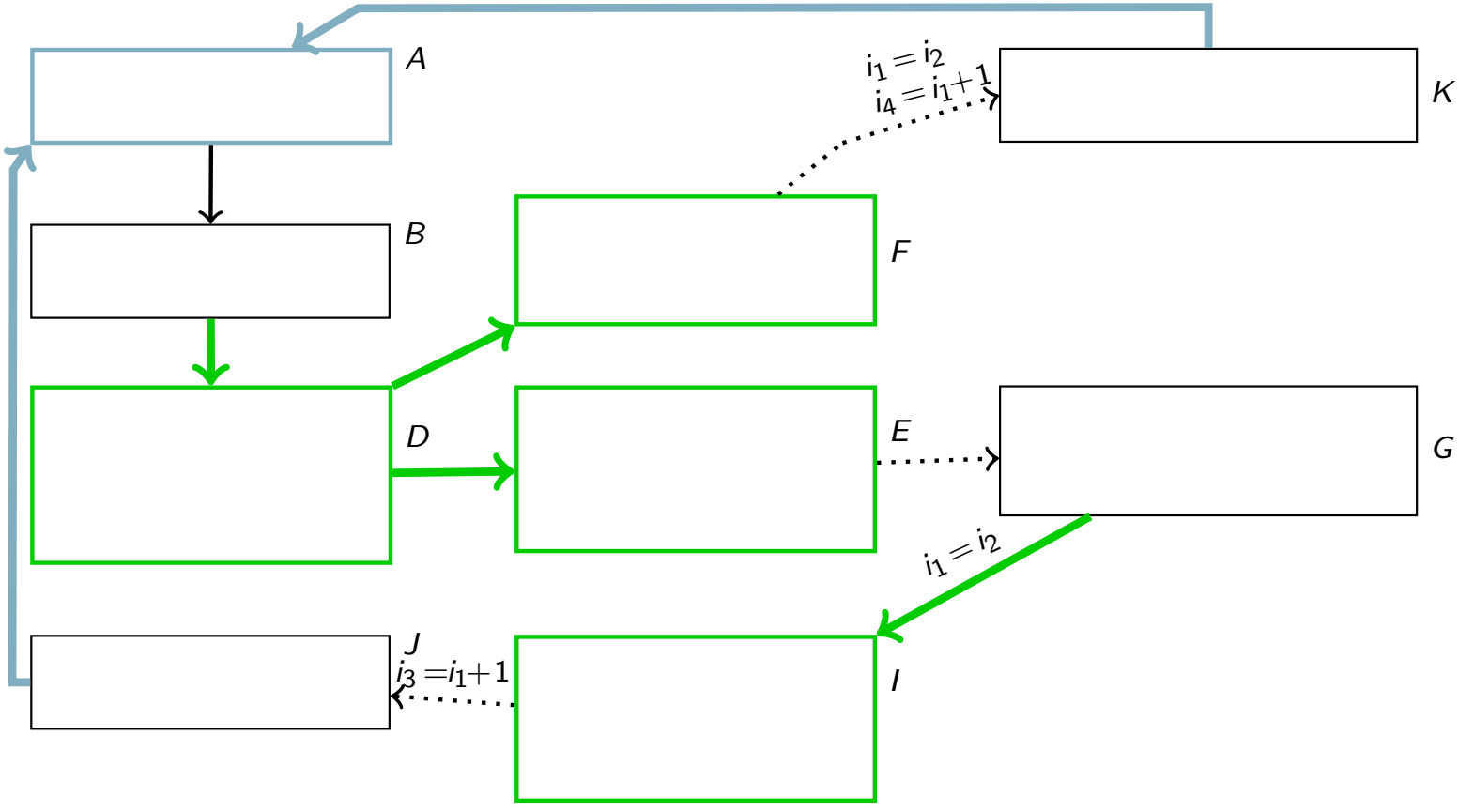



```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

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

  High-level argument: Number of unvisited elements strictly decreasing

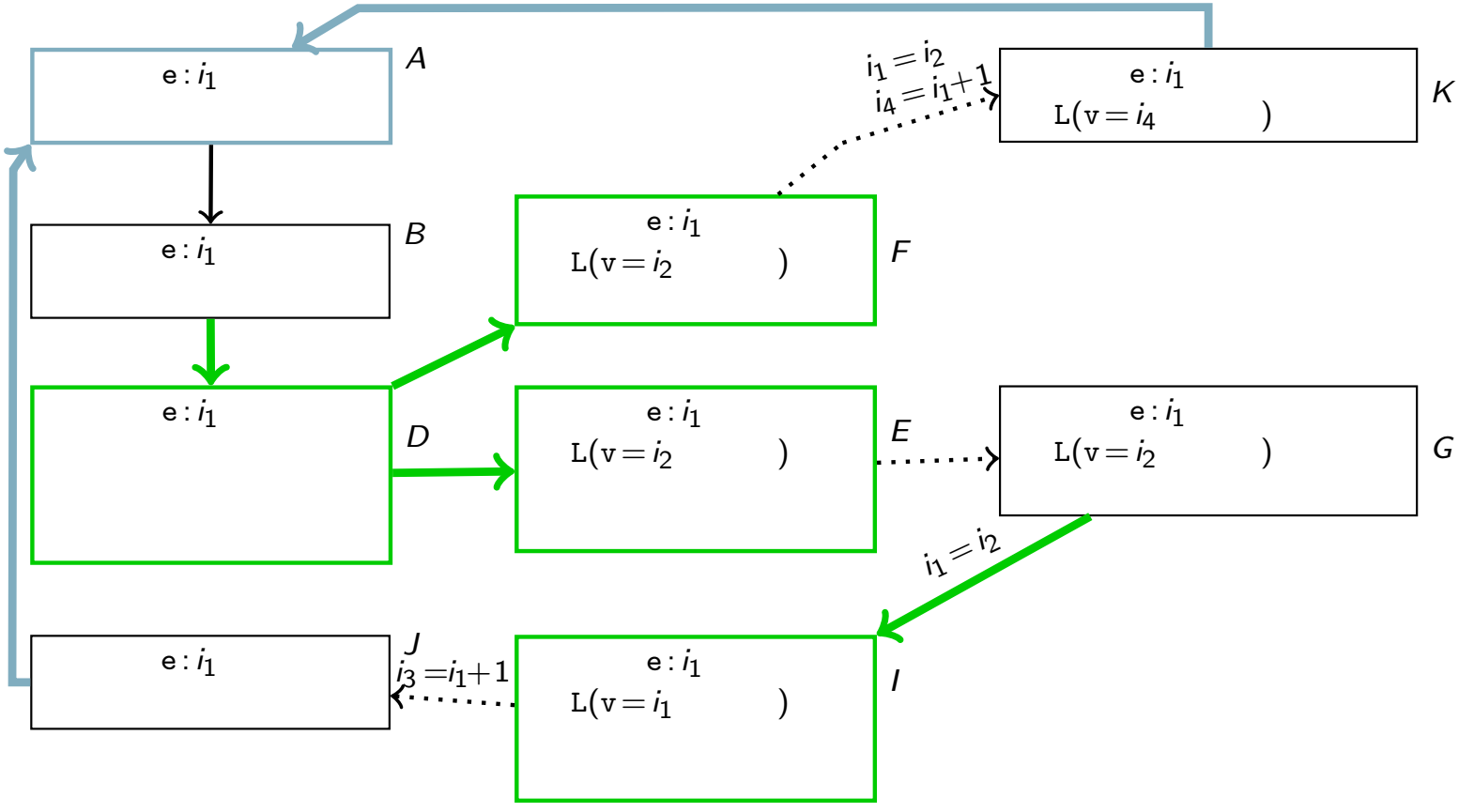
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



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

```
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
```

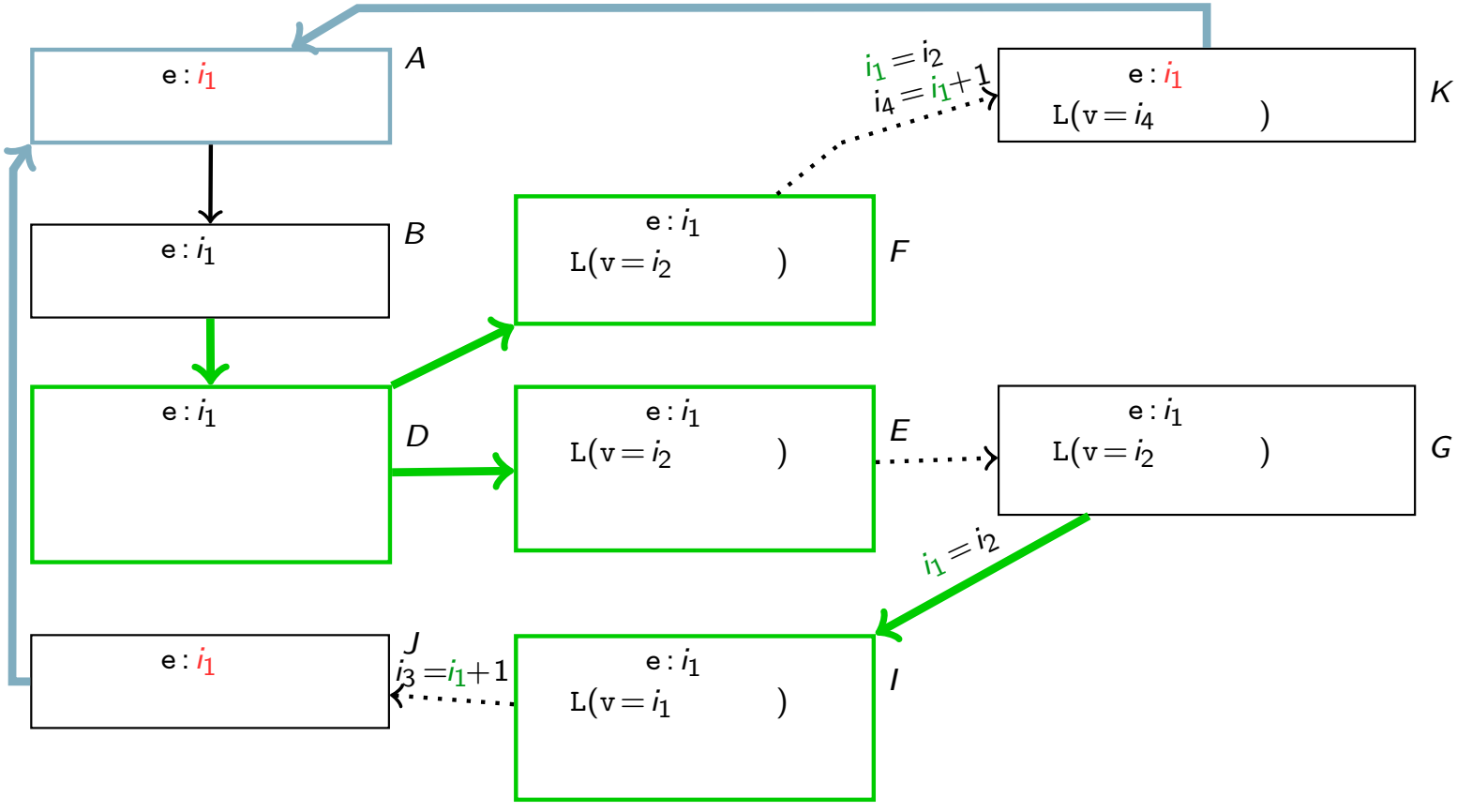


Q: What is an "unvisited element", formally?

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

```
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
```



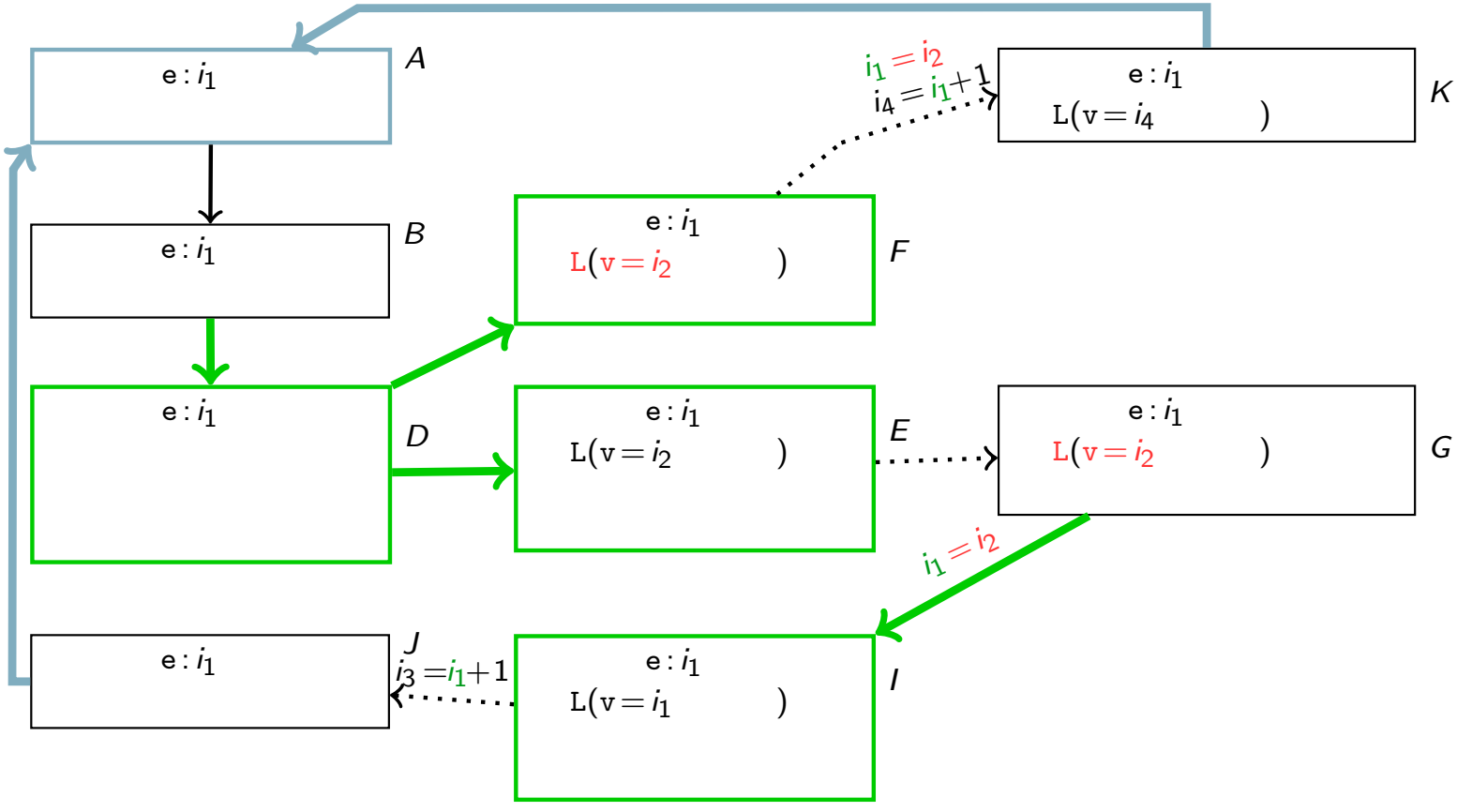
Q: What is an "unvisited element", formally?

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
```



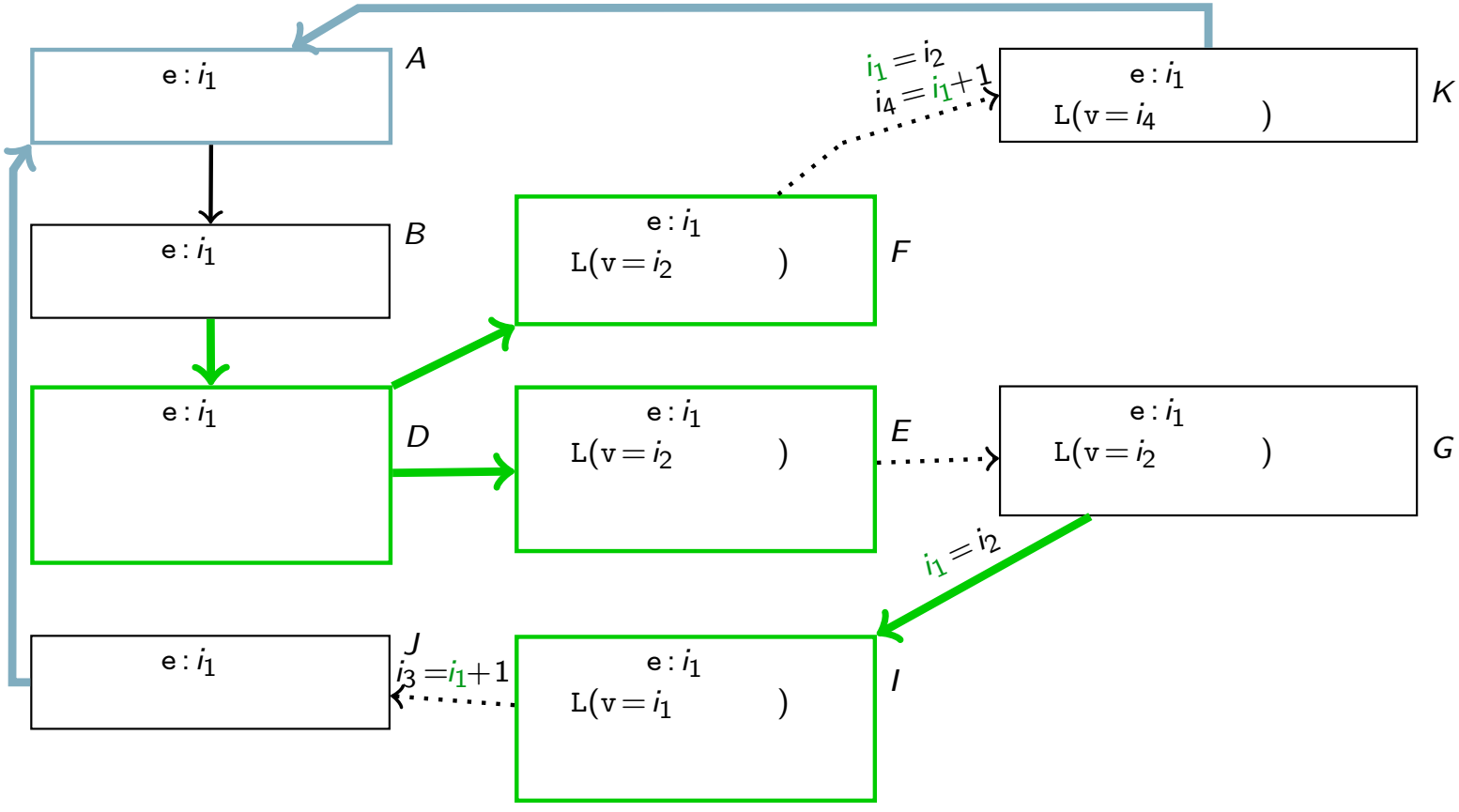

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

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

A: One with L.v $= i_1 =$ e

- Automatically finding this relation:
  1. Identify constant c in SCC
  2. Search property $M =$ 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
```



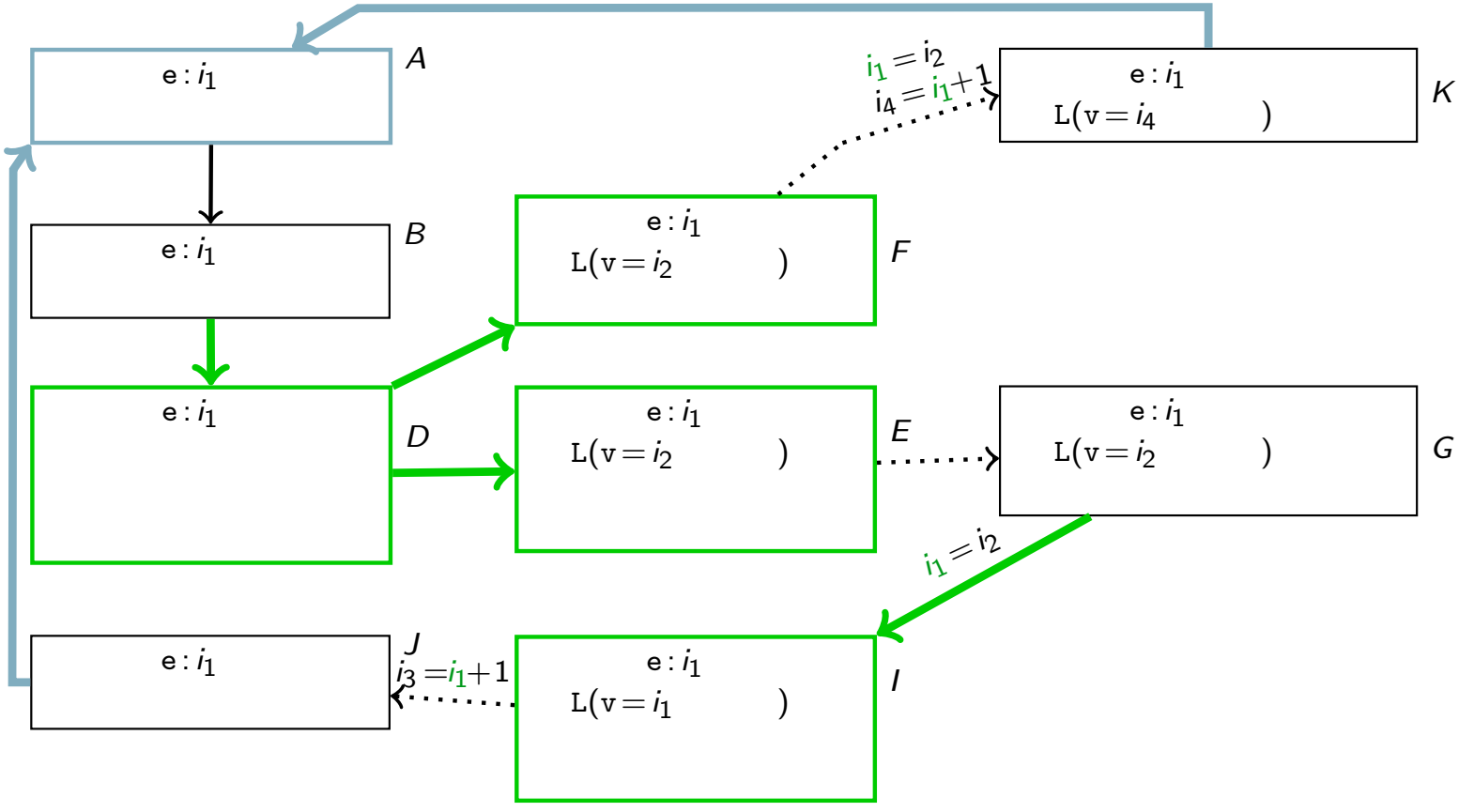

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n; }}
```

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

A: One with L.v $= i_1 =$ e

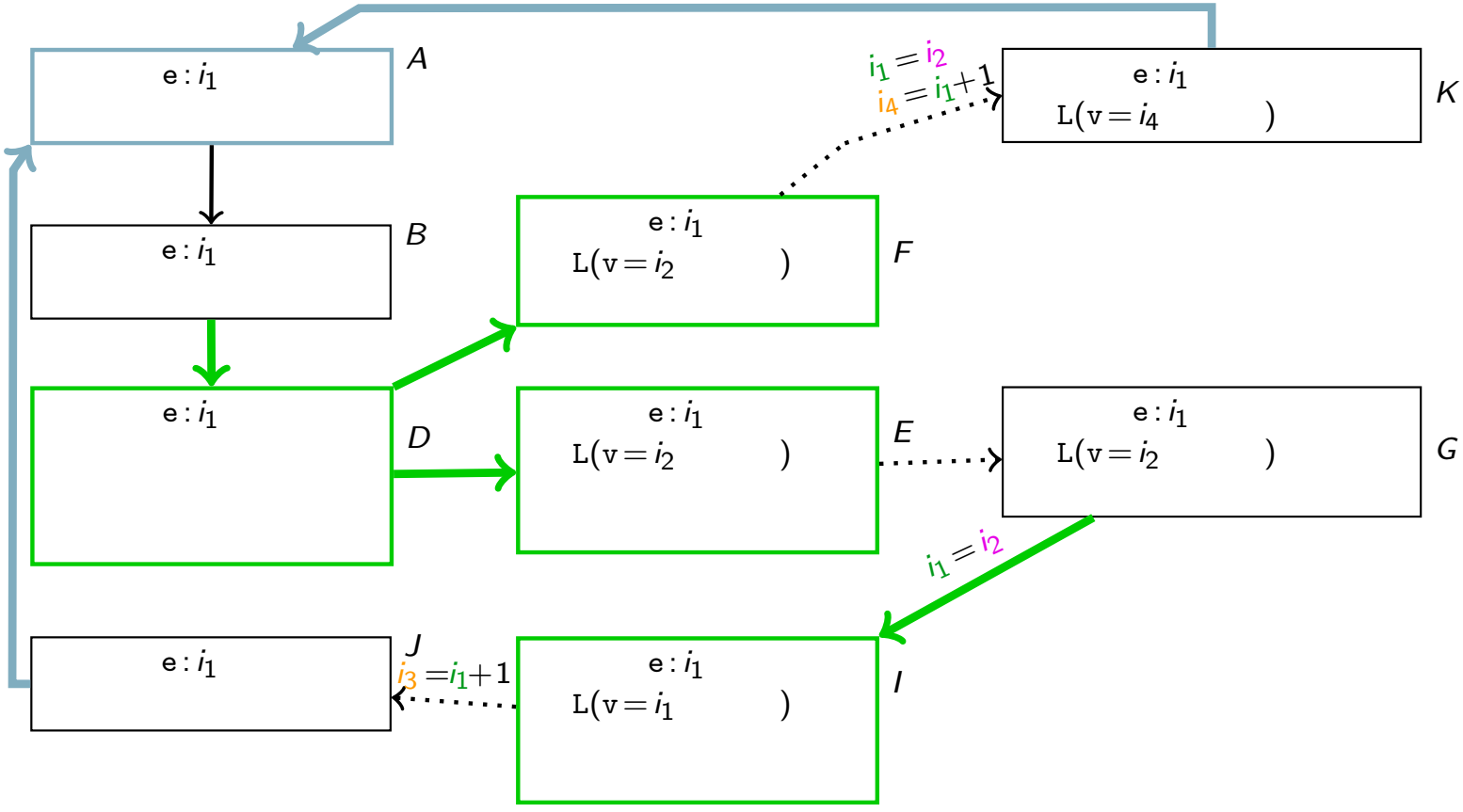
- Automatically finding this relation:
  1. Identify constant c in SCC
  2. Search property $M =$ C.f $\bowtie$ c checked on all cycles
- Track number of objects where C.f $\bowtie$ c holds ($\#_M$)

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

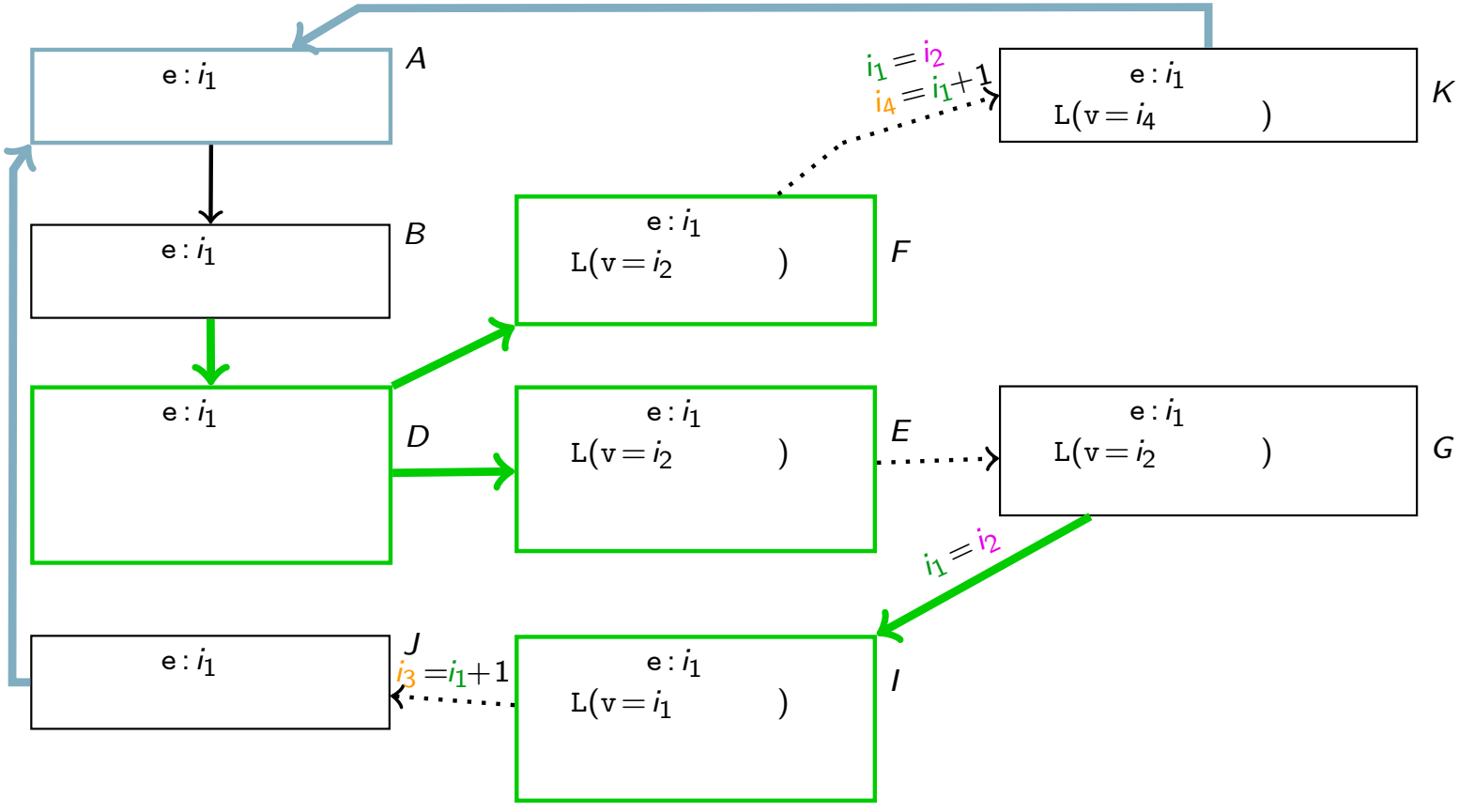
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



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

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

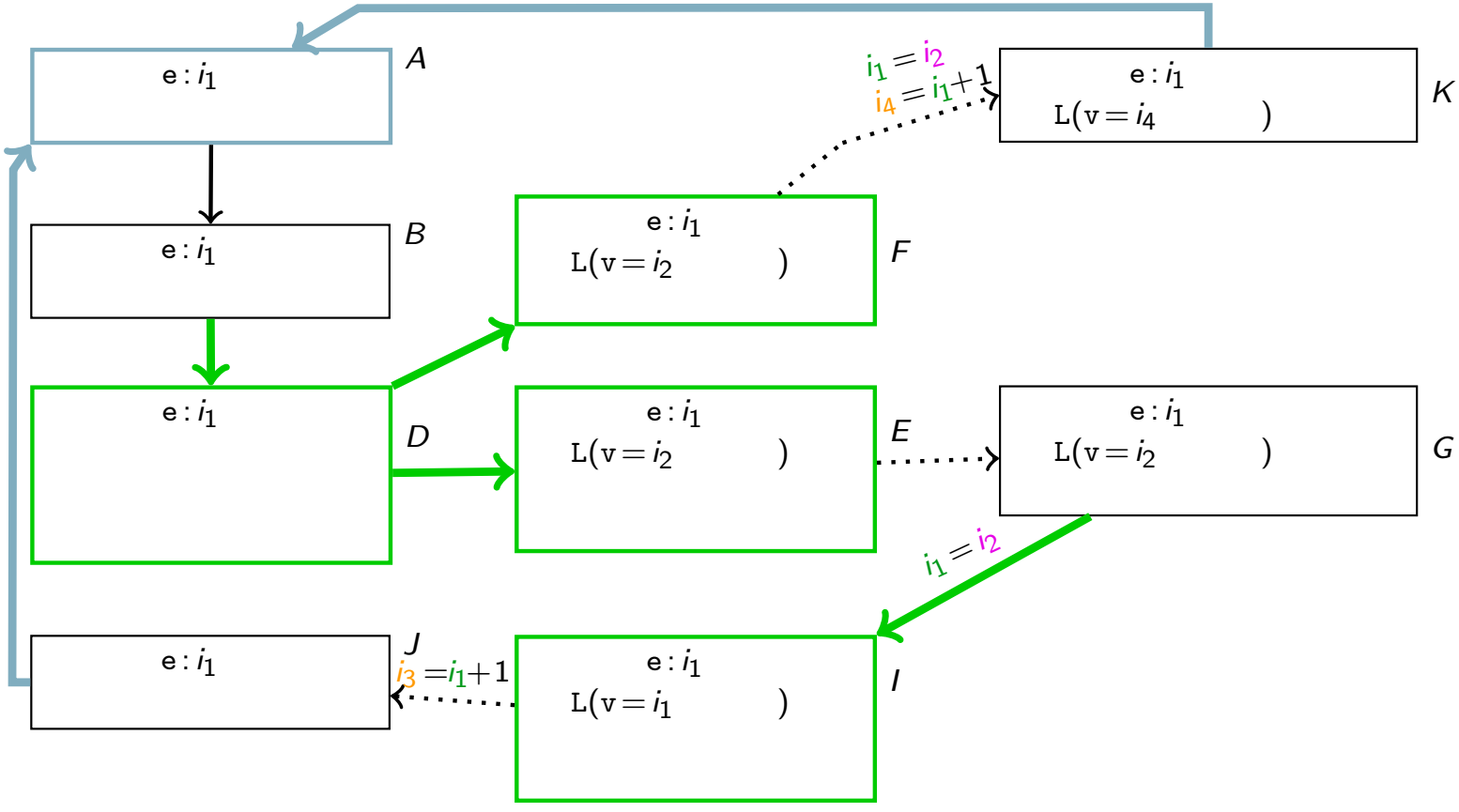
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



Property $M = \texttt{C.f} \bowtie c$ (here: $\texttt{L.v} = i_1$). 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

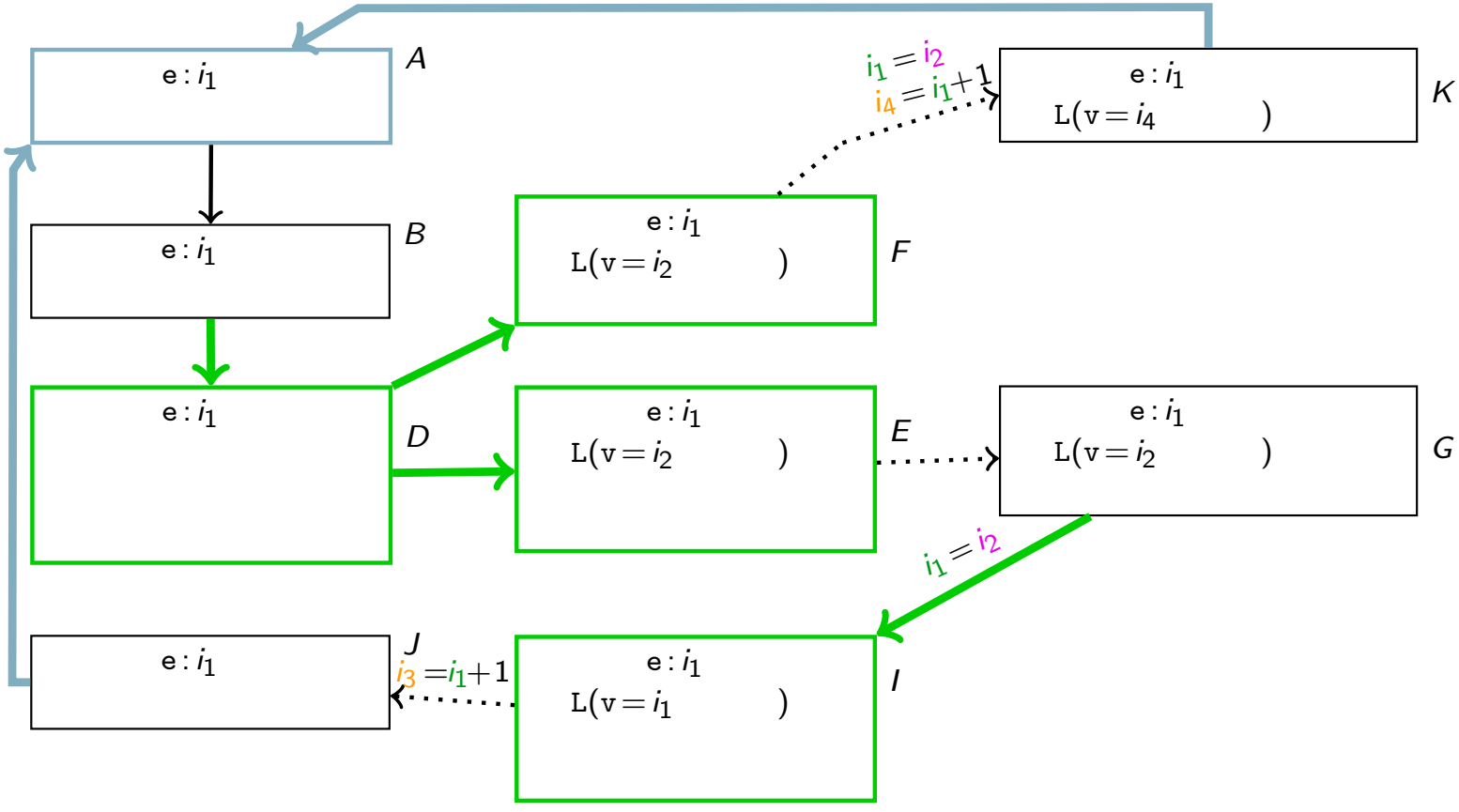
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



Property $M = \texttt{C.f} \bowtie c$ (here: $\texttt{L.v} = i_1$). 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

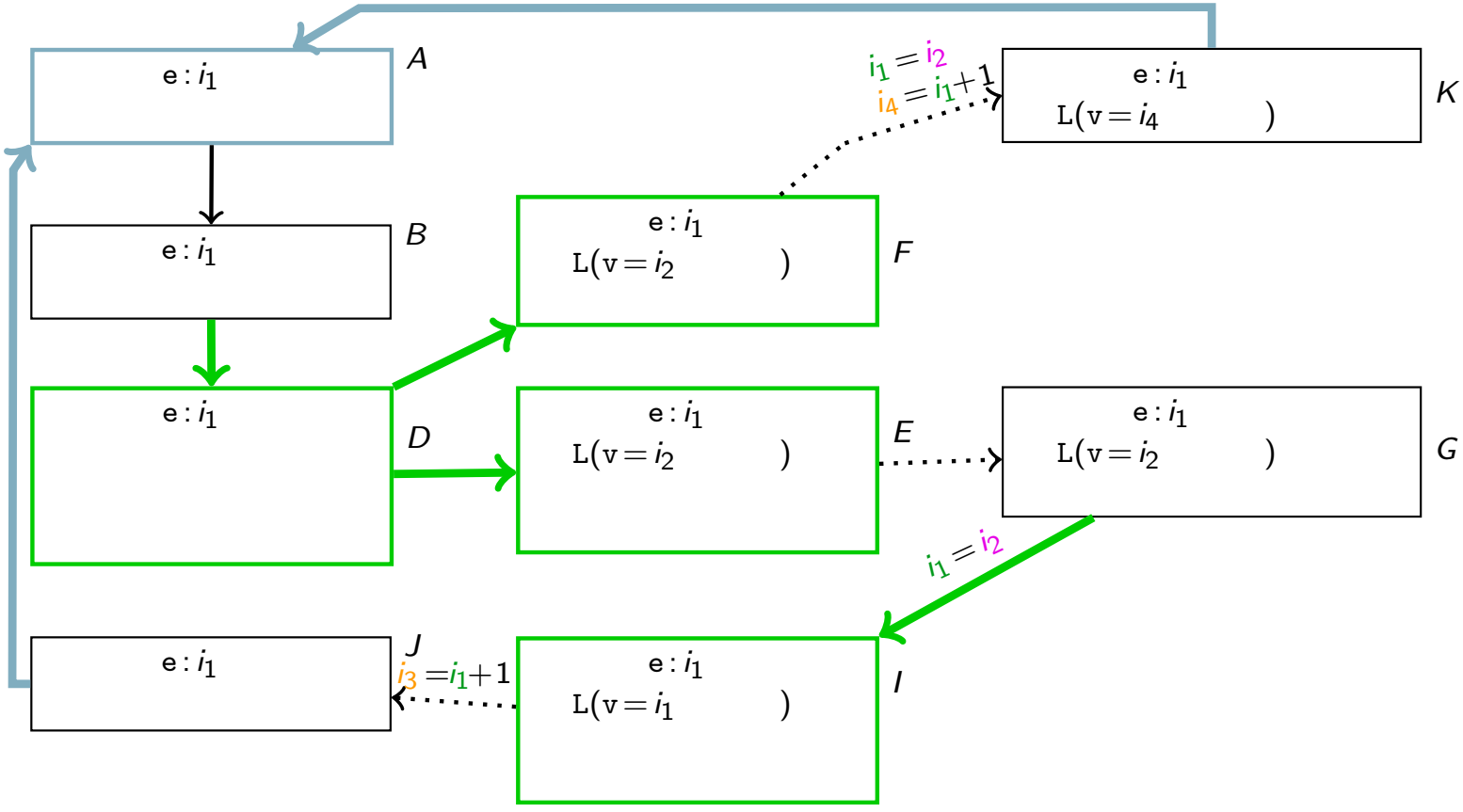
```
00: aload_0
01: getfield v
04: istore_1
05: aload_0
06: getfield v
09: iload_1
10: if_icmpne 28
13: aload_0
14: iload_1
15: iconst_1
16: iadd
17: putfield v
20: aload_0
21: getfield n
24: astore_0
25: goto 5
28: return
```



Property $M = \texttt{C.f} \bowtie c$ (here: $\texttt{L.v} = i_1$). 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.

```
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
```



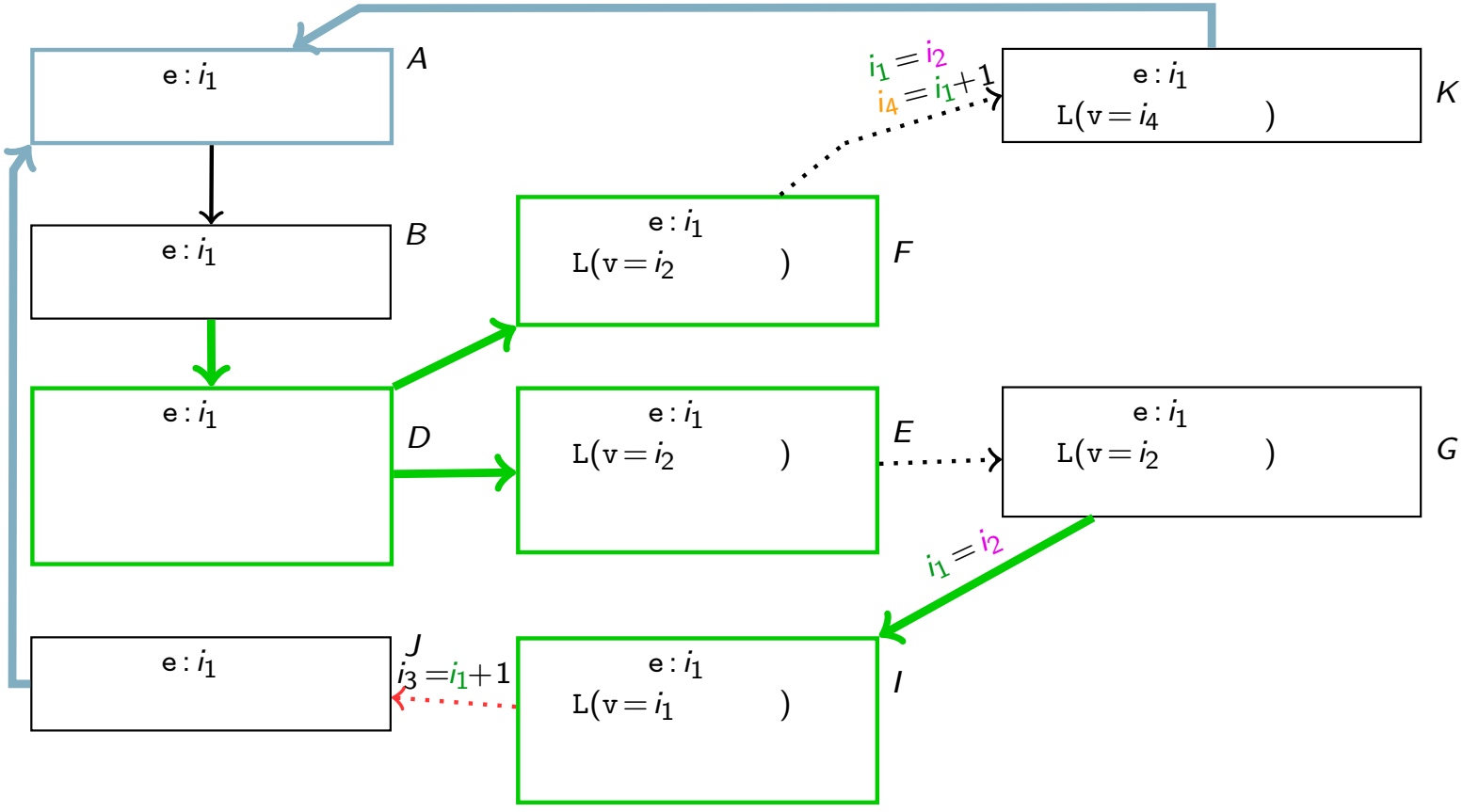
Property $M = \texttt{C.f} \bowtie c$ (here: $\texttt{L.v} = i_1$). When does $\#_M$ change?

- `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

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

```
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
```



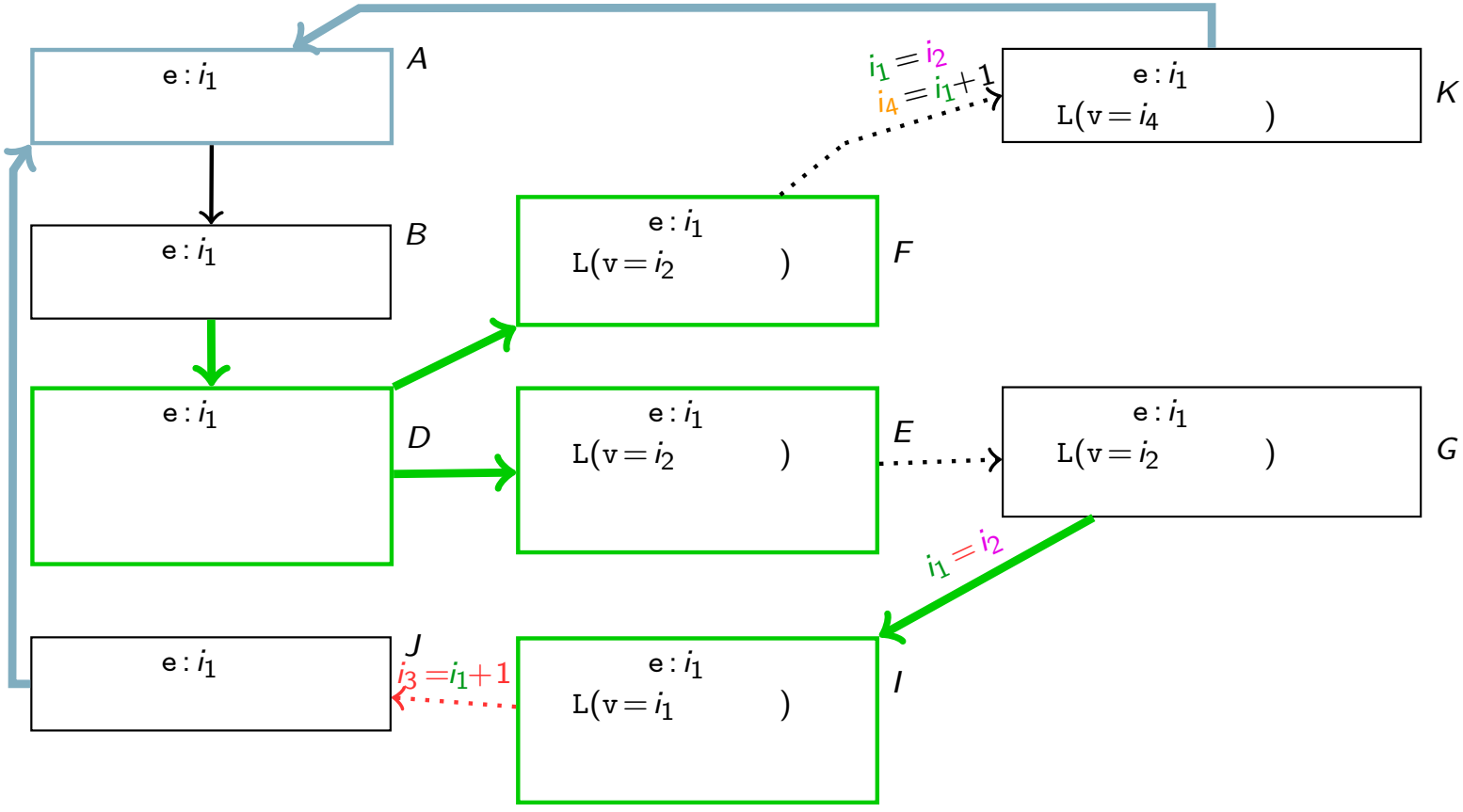
Property $M = \texttt{C.f} \bowtie c$ (here: $\texttt{L.v} = i_1$). When does $\#_M$ change?

- `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

$\Rightarrow i_1 = i_2 \wedge i_3 = i_1 + 1 \;\to\; i_1 = i_1 \wedge \neg i_3 = i_1$

```
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
```



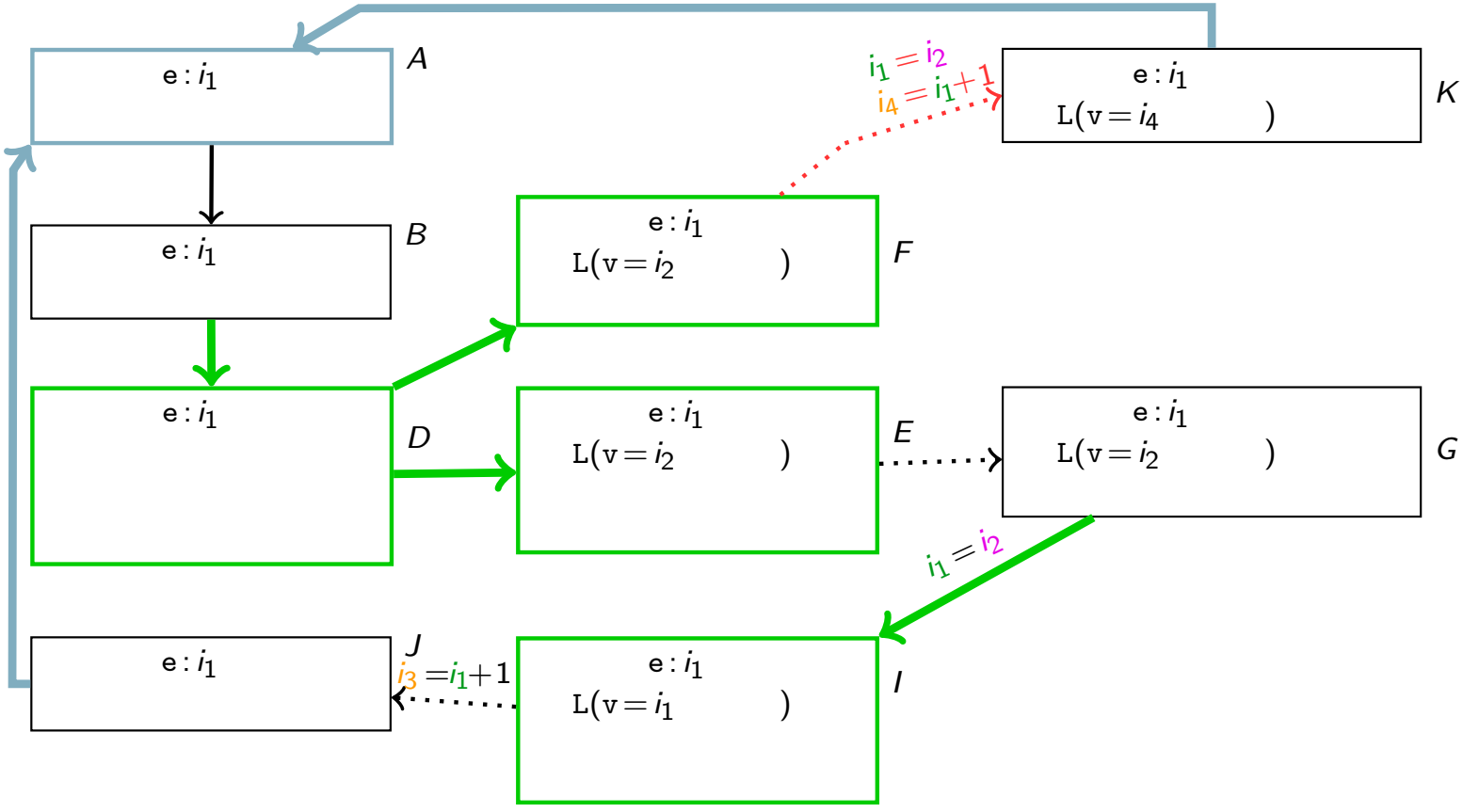
Property $M = \texttt{C.f} \bowtie c$ (here: $\texttt{L.v} = i_1$). When does $\#_M$ change?

- `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: $F \to K$: 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
```

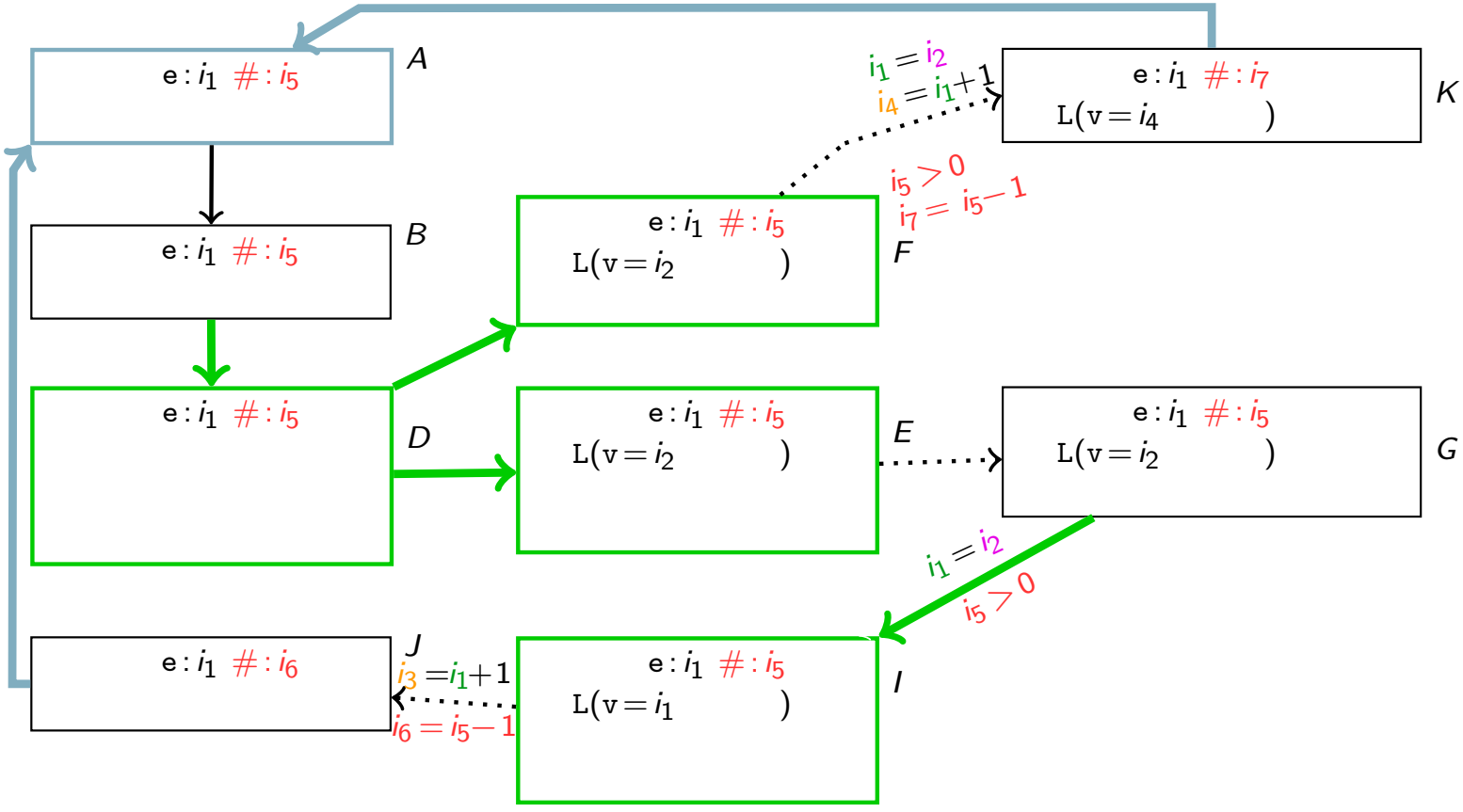


Property $M = \texttt{C.f} \bowtie c$ (here: $\texttt{L.v} = i_1$). When does $\#_M$ change?

- `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.

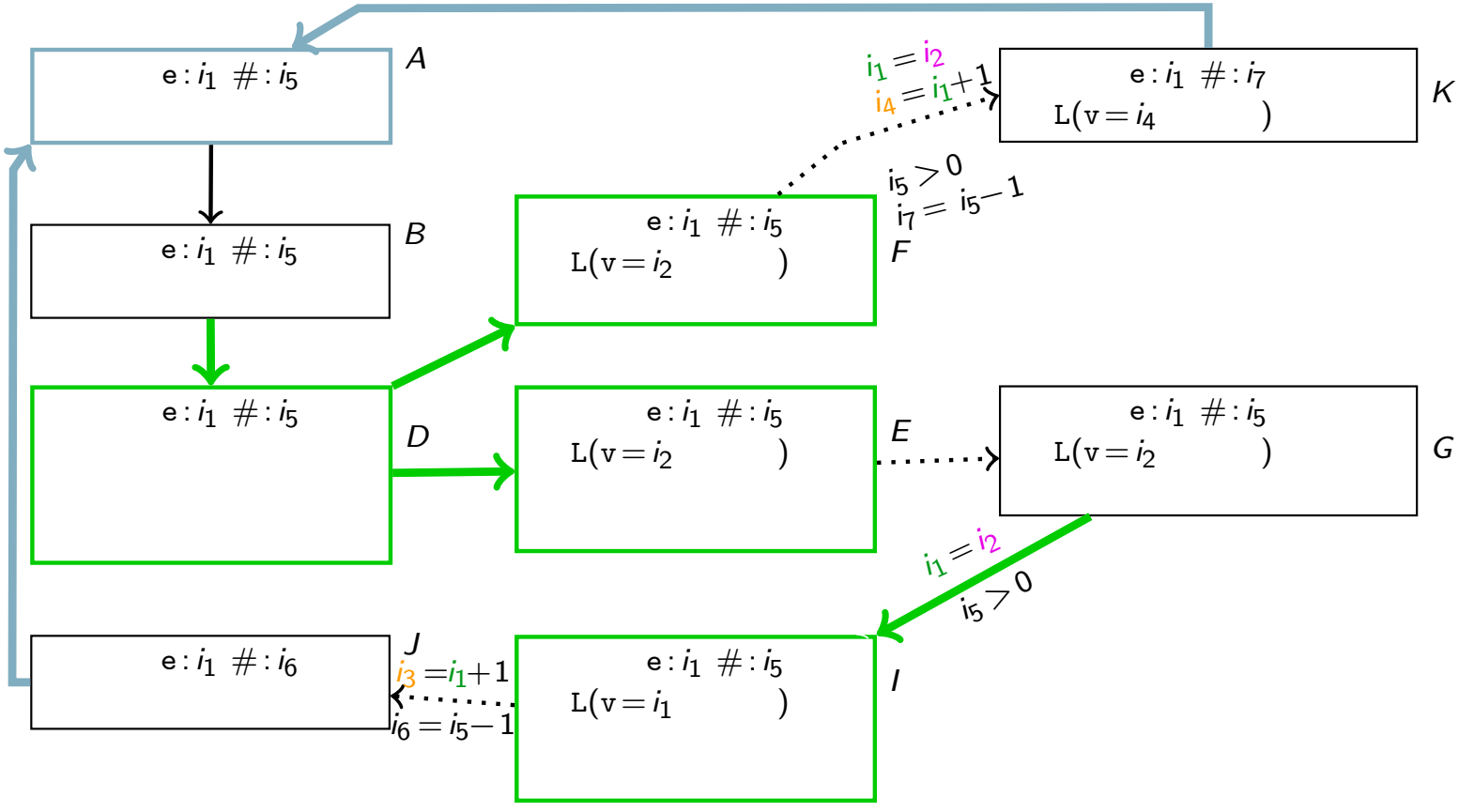
- New `C` object is created: Same for default value

```
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:

$$\begin{aligned} f_A(\ldots, i_6) &\to f_A(\ldots, i_6 - 1) && \mid i_6 > 0 \\ f_A(\ldots, i_7) &\to f_A(\ldots, i_7 - 1) && \mid i_7 > 0 \end{aligned}$$

AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof w.r.t. JINJA [VITA'10]

AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof w.r.t. JINJA [VITA'10]
- Built-in, implicit analyses for nullness, aliasing, sharing, cyclicity

AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof w.r.t. JINJA [VITA'10]
- 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]

AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof w.r.t. JINJA [VITA'10]
- 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]

AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof w.r.t. JINJA [VITA'10]
- 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 cyclic data [CAV'12]
    - by measuring distances
    - by detecting (and ignoring) irrelevant cyclicity
    - by automatically finding and counting markers

AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof w.r.t. JINJA [VITA'10]
- 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 cyclic data [CAV'12]
    - by measuring distances
    - by detecting (and ignoring) irrelevant cyclicity
    - by automatically finding and counting markers
- *Non*-termination analysis [FoVeOOS'11]

Proving Termination of Heap-Manipulating Java Programs

- Evaluated on collection of 387 programs:

	Yes	**No**	**Fail**	**Run** (s)
AProVE	276	88	22	8.4
Julia	191	22	174	4.7
COSTA	160	0	227	11.0

(*Termination Problem Data Base*, classes from `java.utils.*`)

Proving Termination of Heap-Manipulating Java Programs

- Evaluated on collection of 387 programs:

	Yes	**No**	**Fail**	**Run** (s)
AProVE	276	88	22	8.4
Julia	191	22	174	4.7
COSTA	160	0	227	11.0

(*Termination Problem Data Base*, classes from `java.utils.*`)

- Won Termination Competition 2012

Proving Termination of Heap-Manipulating Java Programs

- Evaluated on collection of 387 programs:

	Yes	**No**	**Fail**	**Run** (s)
AProVE	276	88	22	8.4
Julia	191	22	174	4.7
COSTA	160	0	227	11.0

(*Termination Problem Data Base*, classes from `java.utils.*`)

- Won Termination Competition 2012
- Open problems:
  - Abstraction refinement
  - Modular analysis

Proving Termination of Heap-Manipulating Java Programs

- Evaluated on collection of 387 programs:

	Yes	**No**	**Fail**	**Run** (s)
AProVE	276	88	22	8.4
Julia	191	22	174	4.7
COSTA	160	0	227	11.0

(*Termination Problem Data Base*, classes from `java.utils.*`)

- Won Termination Competition 2012
- Open problems:
  - Abstraction refinement
  - Modular analysis

Specialized abstract domains:

- **easy to automate**
- **very effective**