Proving Java Termination: Reducing the Ugly to the Bad

Marc Brockschmidt

LuFG Informatik 2, RWTH Aachen University, Germany

September 2012

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:

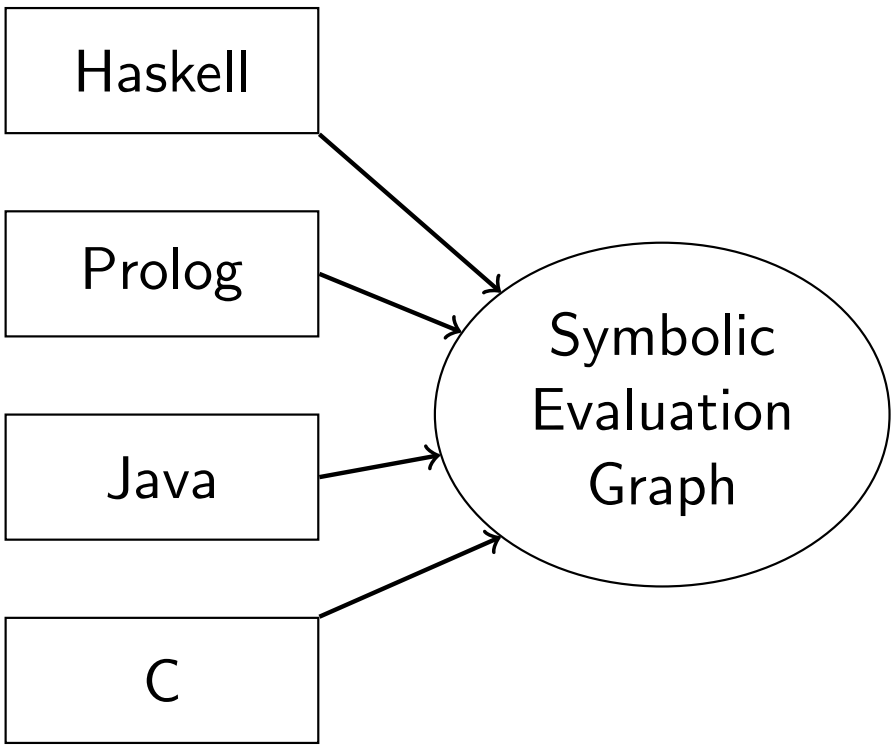
- Termination Analysis for C (via Polynomial Orders)
  KITTeL – (*Falke, Kapur, Sinz*, since '11)
- Termination Analysis for Java (via Path Length, CLP backend)
  Julia – (*Spoto, Mesnard, Payet*, since '08)
  COSTA – (*Albert, Arenas, Codish, Genaim, Puebla, Zanardini*, since '08)

Rewriting-backed approach: Idea

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

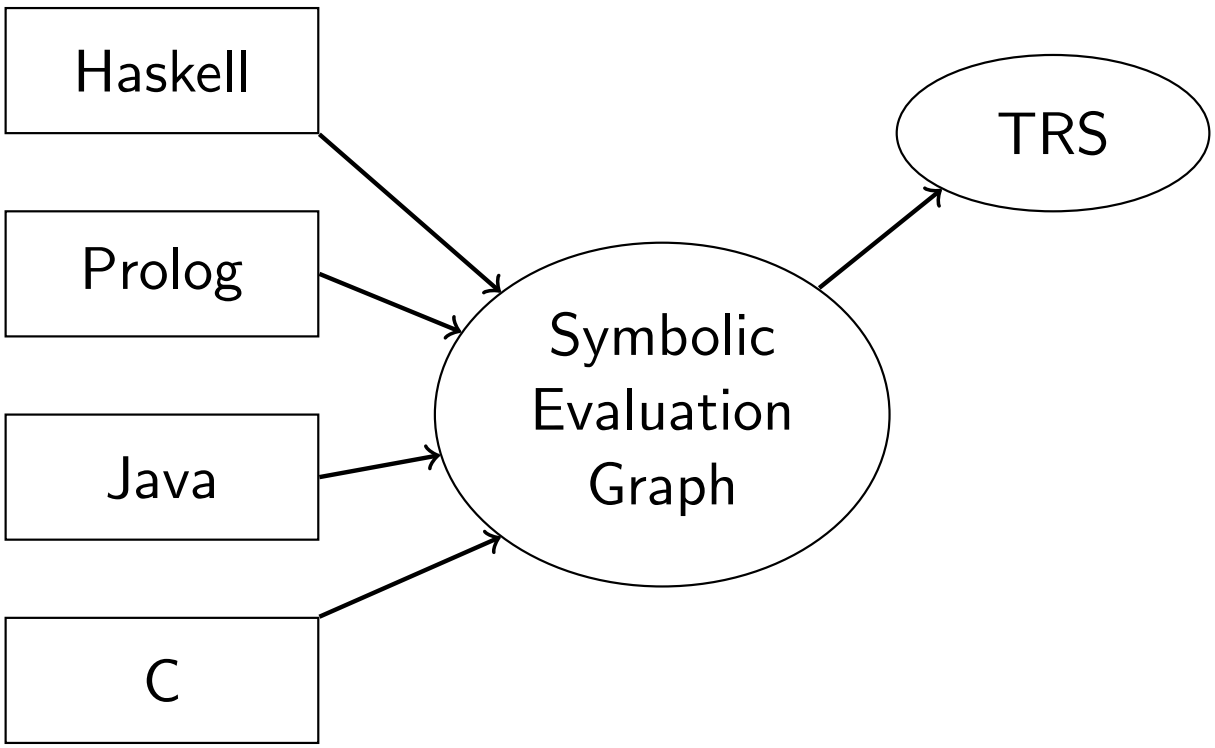
Rewriting-backed approach: Idea

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



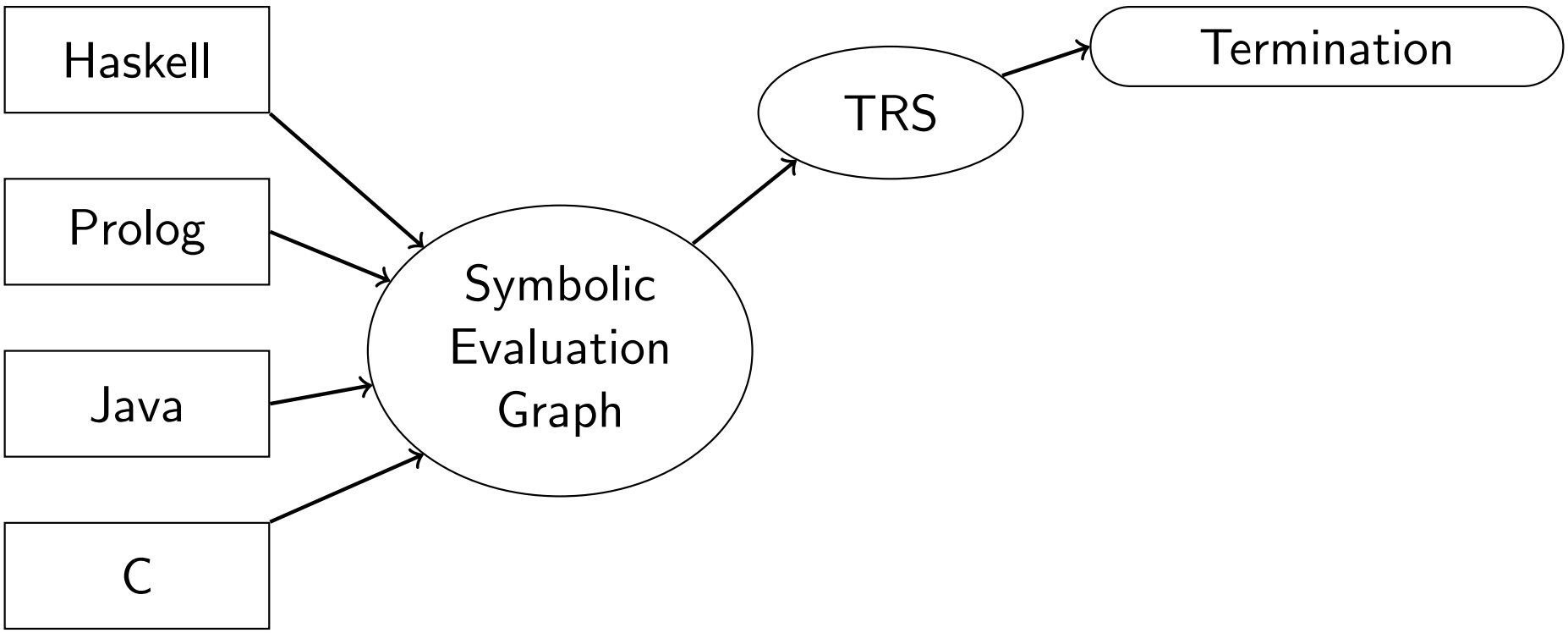
Rewriting-backed approach: Idea

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



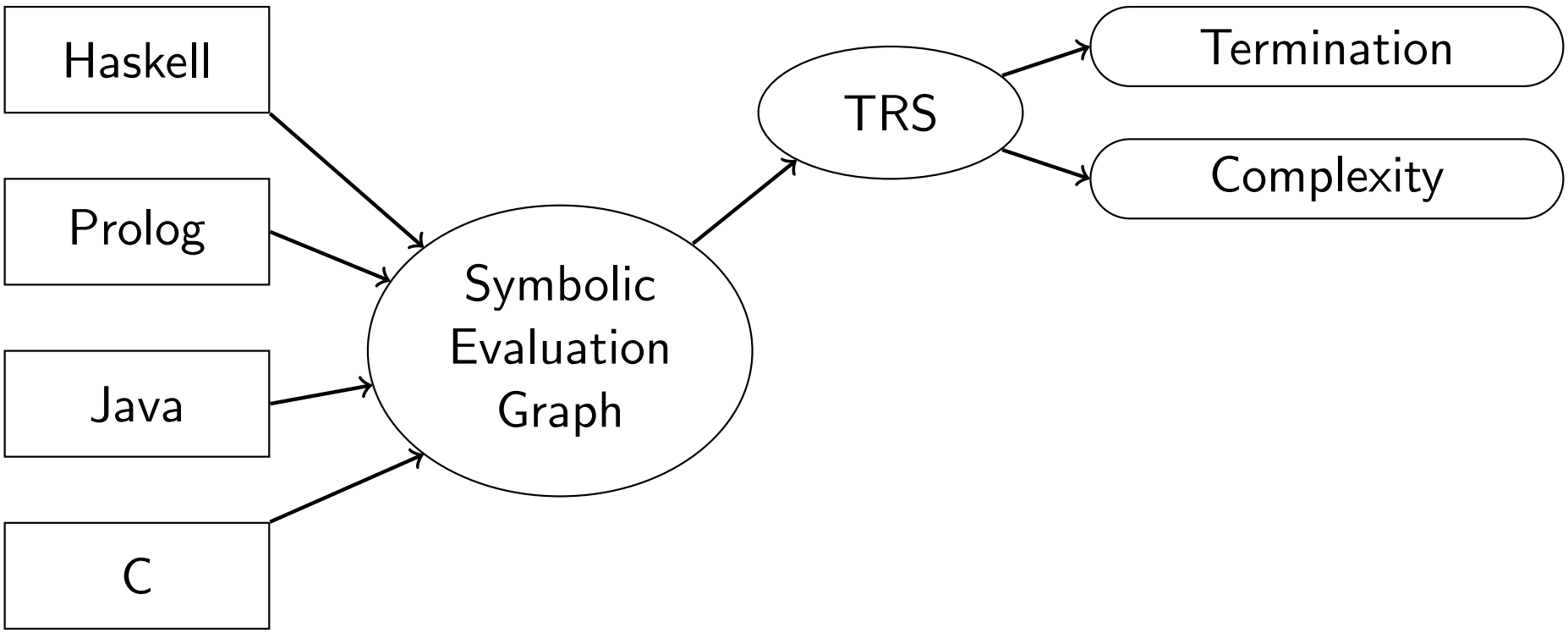
Rewriting-backed approach: Idea

- 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



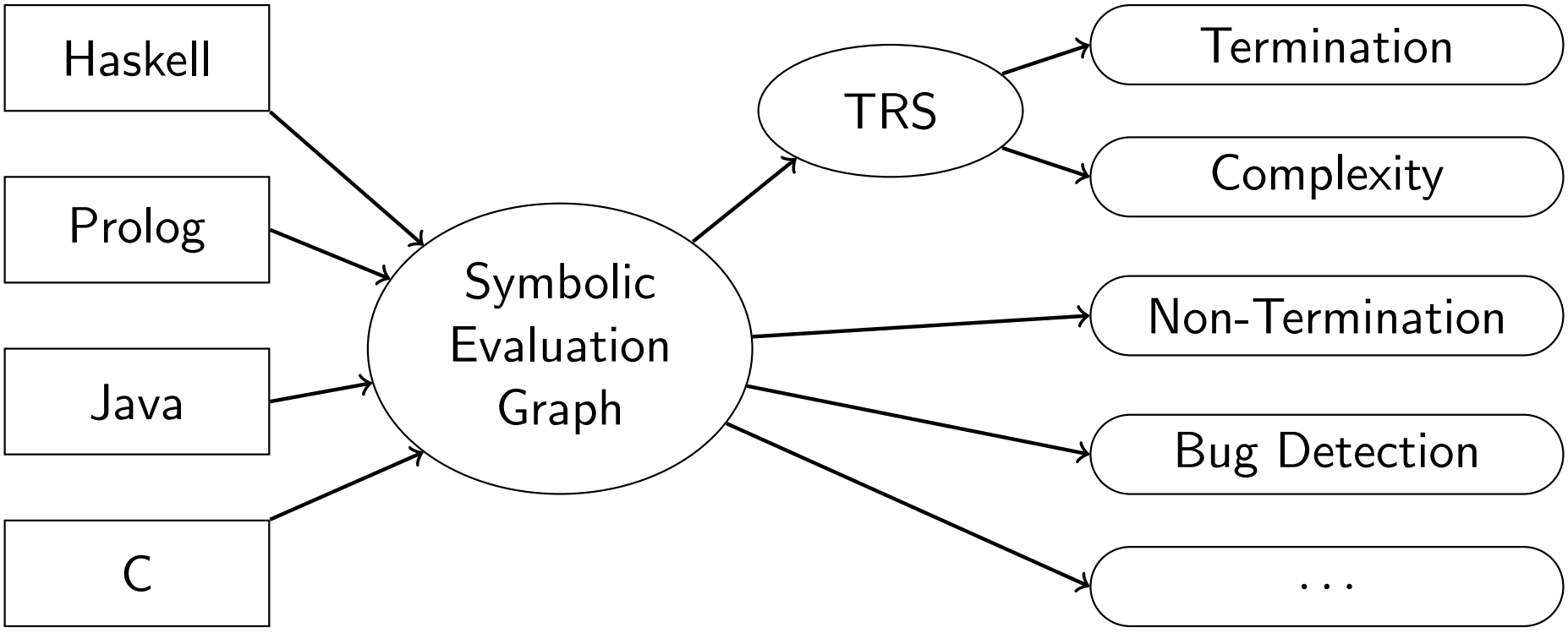
Rewriting-backed approach: Idea

- 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

Handling of user-defined acyclic data structures:

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

Rewriting-backed approach: Advantages

Handling of user-defined acyclic 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 acyclic 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 acyclic 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

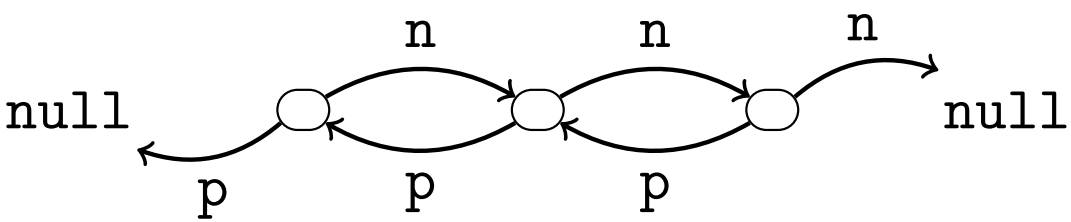
$\Rightarrow$ Complex orders for user-defined data structures possible

length: the example

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

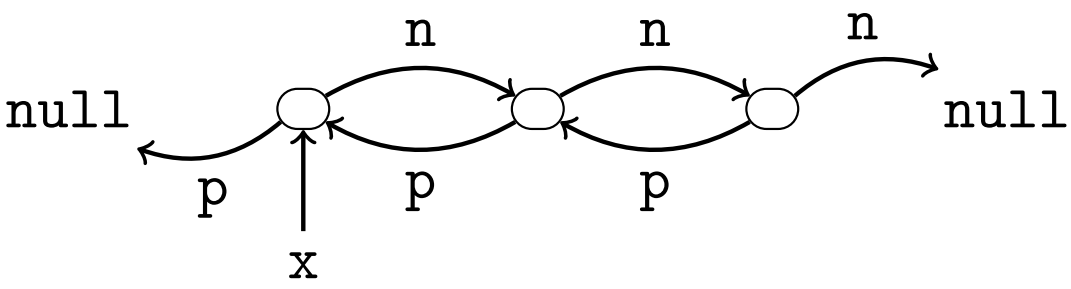
length: the example

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



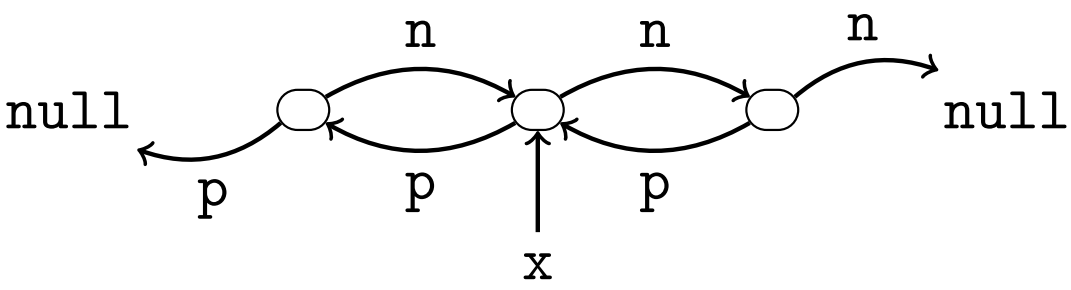
length: the example

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



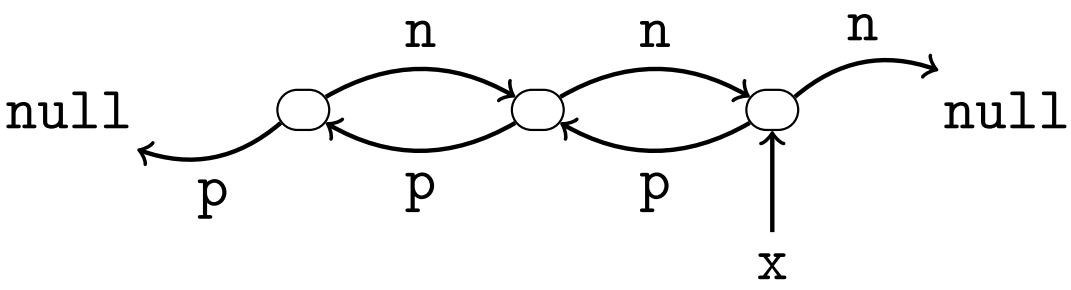
length: the example

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



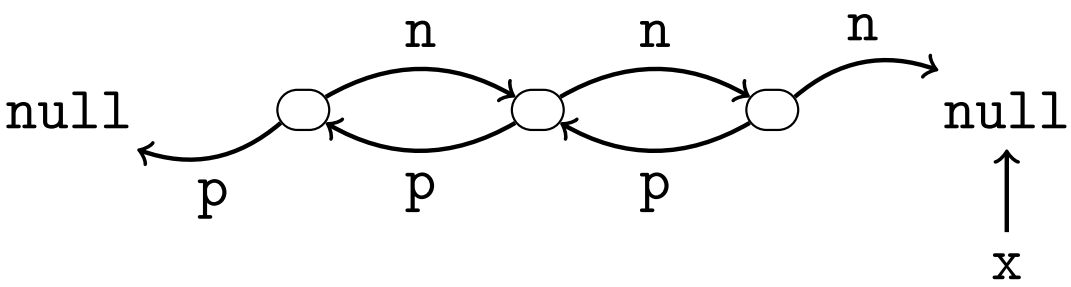
length: the example

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



length: the example

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



length: the example

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

```
00: iconst_1      #load 1
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
```

Abstract Java virtual machine states

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

```
00: iconst_1      #load 1
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
```

Abstract Java virtual machine states

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

```
00: iconst_1      #load 1
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
$o_1 : \mathrm{L}(?)$	$o_1 \circlearrowleft_{\{p,n\}}$	

- Next program instruction

Abstract Java virtual machine states

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

```
00: iconst_1      #load 1
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:

- Next program instruction
- Local variables
- Operand stack

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

Abstract Java virtual machine states

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

```
00: iconst_1      #load 1
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:

Abstract Java virtual machine states

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

```
00: iconst_1      #load 1
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`

Abstract Java virtual machine states

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

```
00: iconst_1      #load 1
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: $o_2 : \mathrm{L}(\mathrm{n} = o_3)$

Abstract Java virtual machine states

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

```
00: iconst_1      #load 1
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: $i_1 : \mathbb{Z}$

Abstract Java virtual machine states

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

```
00: iconst_1      #load 1
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: $i_1 : \mathbb{Z}$

Only explicit sharing

Abstract Java virtual machine states

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

```
00: iconst_1      #load 1
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: $o_2 : \mathrm{L}(\mathtt{n} = o_3)$
- Unknown integer: $i_1 : \mathbb{Z}$

Heap predicates: **Only explicit sharing**

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

Abstract Java virtual machine states

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

```
00: iconst_1      #load 1
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: $o_1 \diagdown\!\!\diagup o_2$

Abstract Java virtual machine states

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

```
00: iconst_1      #load 1
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

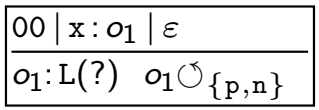
Heap information:

- 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: $o_1 \diagdown\!\!\diagup o_2$
- Reference might have cycles containing all fields F: $o_1\circlearrowleft_F$

```
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
```



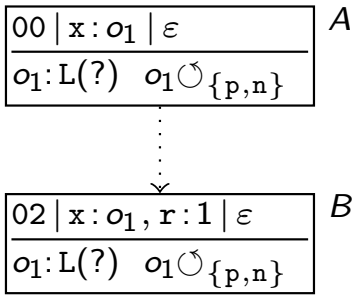
A

State A:

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

```
int length(L x) {
  int r = 1;
  while (x != null) {
    x = x.n; r++;   }
  return r;      }
```

```
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
```



State *A*:

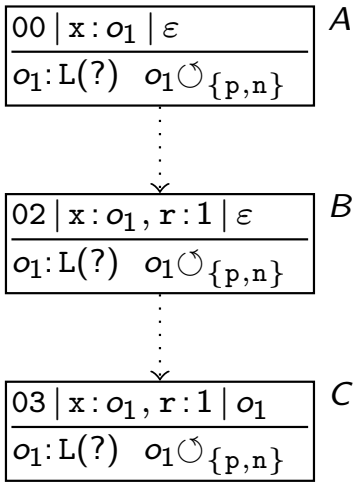
- `x` some list, might contain cycles using `p` and `n`

State *B*:

- Initialized variable `r` to 1

```
int length(L x) {
  int r = 1;
  while (x != null) {
    x = x.n; r++;   }
  return r;     }
```

```
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
```



State *A*:

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

State *B*:

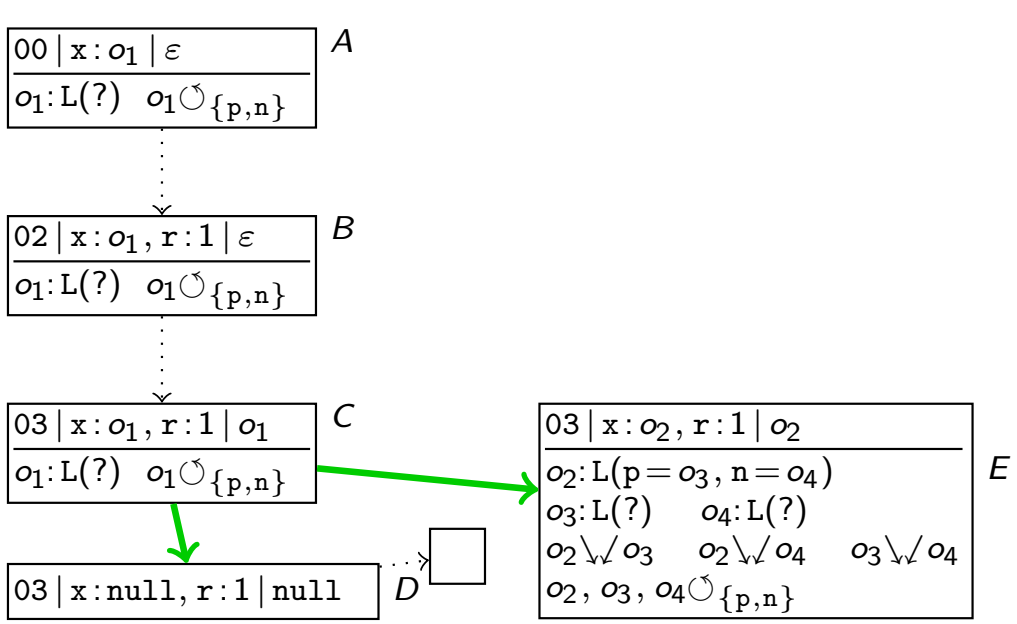
- Initialized variable `r` to 1

State *C*:

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

```
int length(L x) {
  int r = 1;
  while (x != null) {
    x = x.n; r++;   }
  return r;     }
```

```
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
```



State *A*:

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

State *B*:

- Initialized variable `r` to 1

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

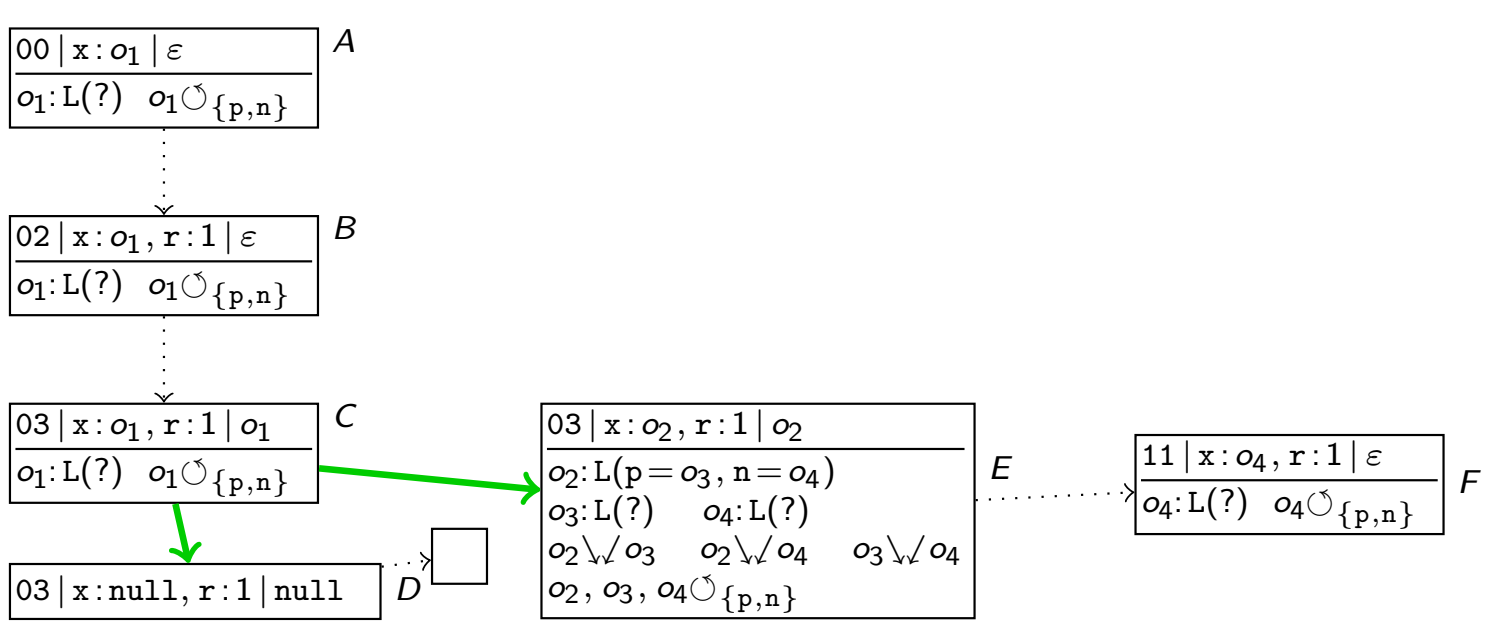
- `x` (o_1) null? We do not know!

⇒ 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 = 1;
  while (x != null) {
    x = x.n; r++;   }
  return r;     }
```

```
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
```

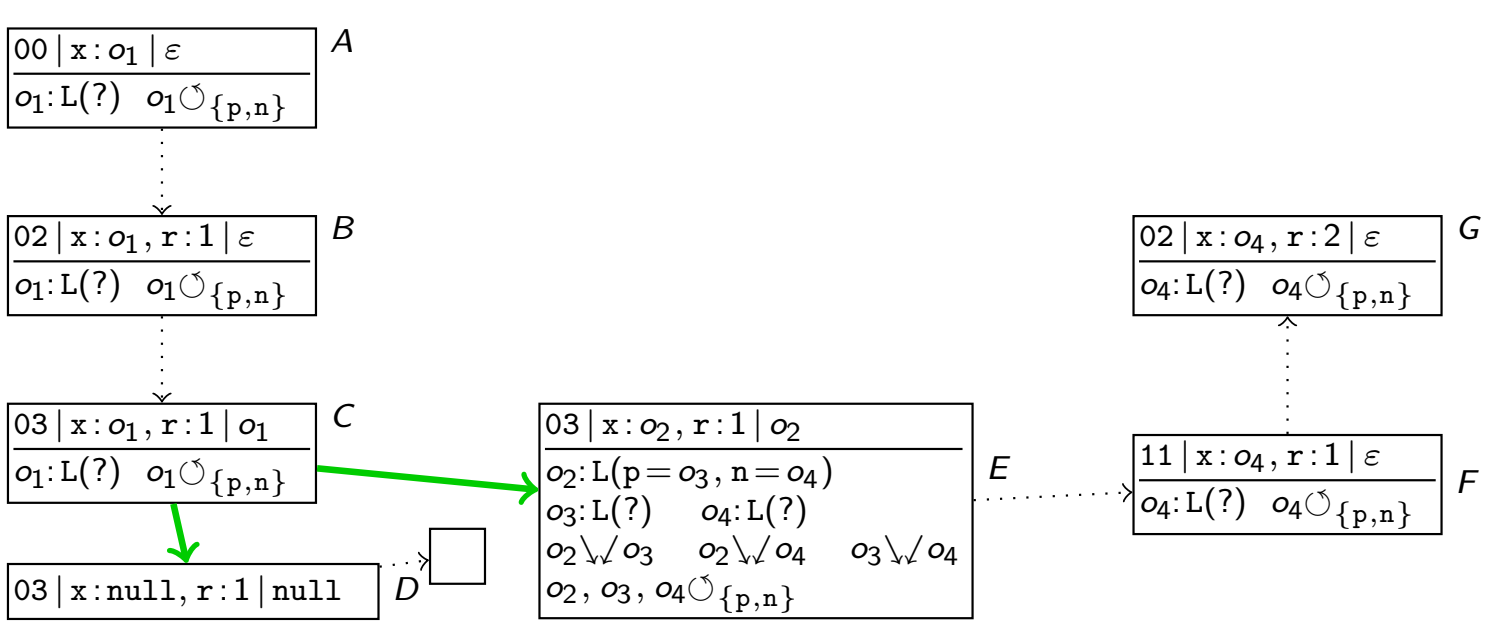


State *F*:

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

```
int length(L x) {
  int r = 1;
  while (x != null) {
    x = x.n; r++;   }
  return r;       }
```

```
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
```



State *F*:

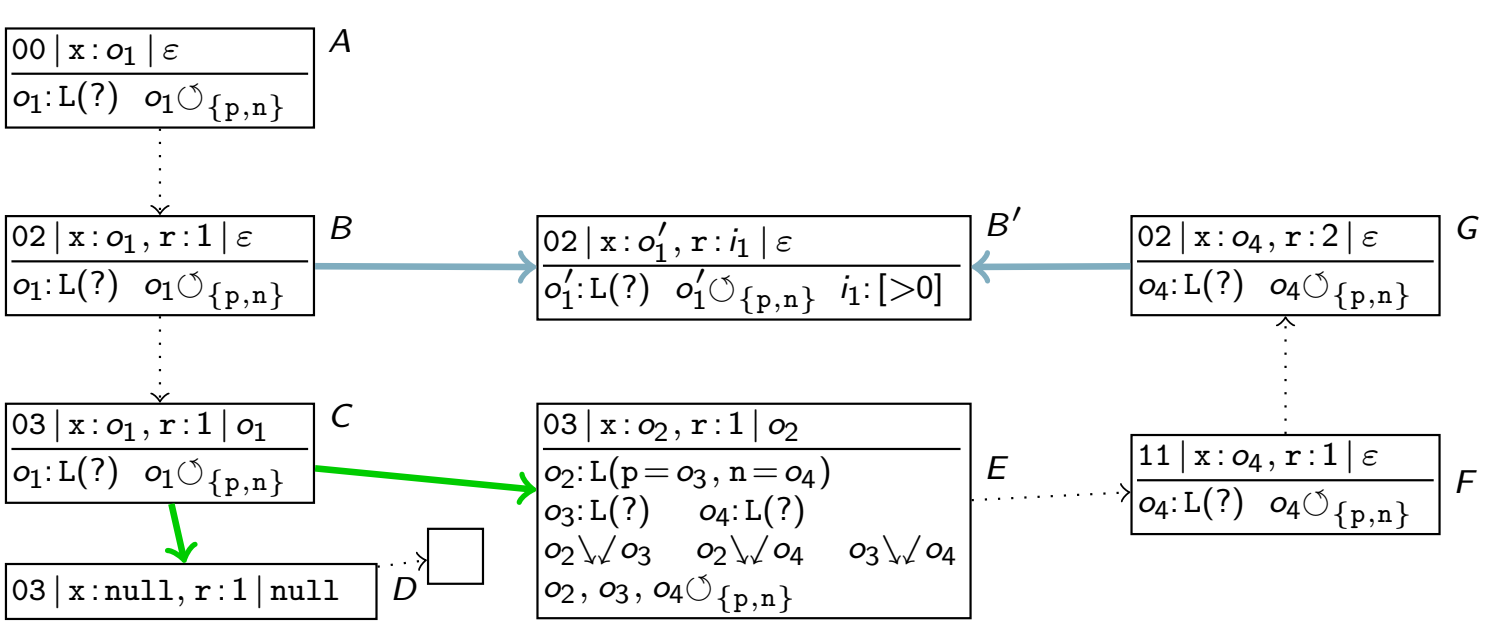
- Stored `x.n` to `x` (allowing for GC)

State *G*:

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

```
int length(L x) {
  int r = 1;
  while (x != null) {
    x = x.n; r++;   }
  return r;     }
```

```
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
```



State F:

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

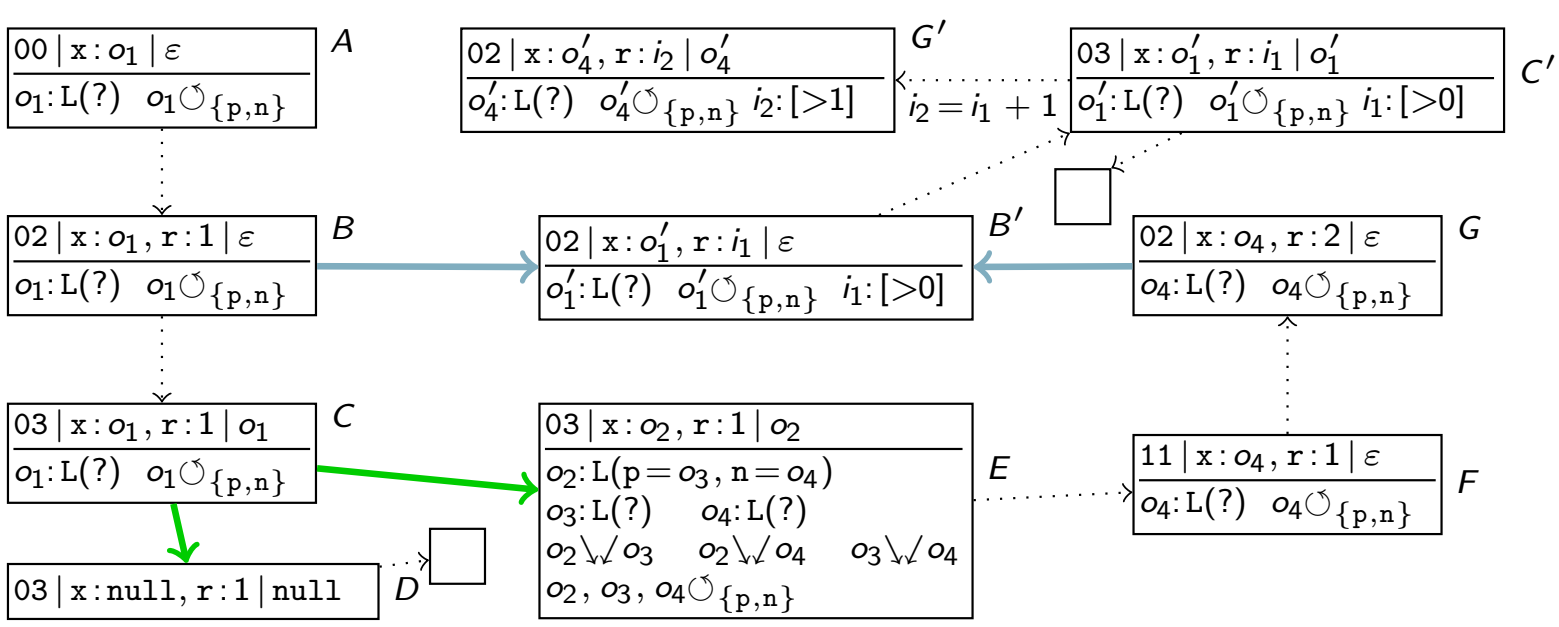
States G, B':

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

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

```
int length(L x) {
  int r = 1;
  while (x != null) {
    x = x.n; r++;   }
  return r;       }
```

```
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
```



State F:

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

States G, B':

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

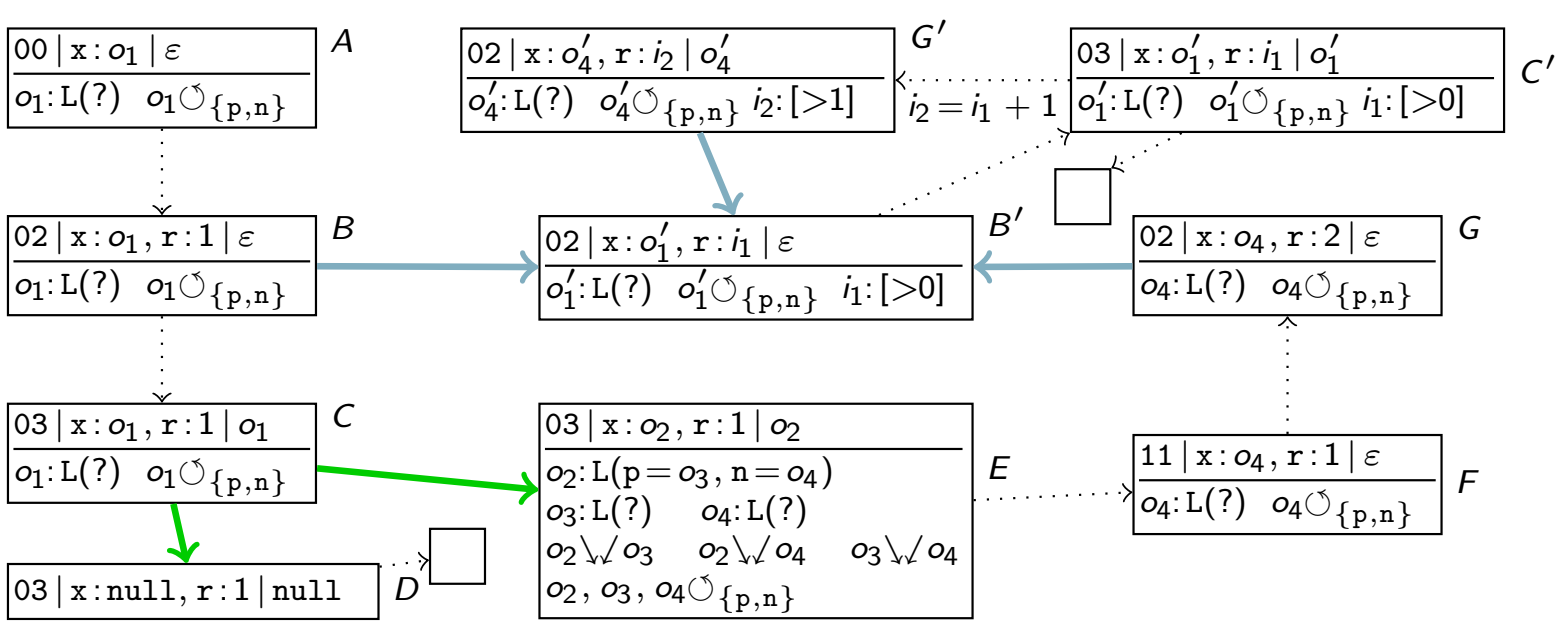
$\Rightarrow$ Generalization: "Merge" states B, G

States C', G':

- Repetition of C, G

```
int length(L x) {
  int r = 1;
  while (x != null) {
    x = x.n; r++;   }
  return r;       }
```

```
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
```



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 = 1;
  while (x != null) {
    x = x.n; r++;   }
  return r;       }
```

Transforming values to terms

$$\frac{03 \mid \mathtt{x} : o_2, \mathtt{r} : 1 \mid o_2}{\begin{array}{l} o_2 : \mathrm{L}(\mathtt{p} = o_3, \mathtt{n} = o_4) \\ o_3 : \mathrm{L}(?) \quad o_4 : \mathrm{L}(?) \\ o_2 \between o_3 \quad o_2 \between o_4 \quad o_3 \between o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \end{array}} \; E$$

Transforming values to terms

$$\frac{03 \mid \mathtt{x} : o_2, \mathtt{r} : 1 \mid o_2}{\begin{array}{l} o_2 : \mathtt{L}(\mathtt{p} = o_3, \mathtt{n} = o_4) \\ o_3 : \mathtt{L}(?) \quad o_4 : \mathtt{L}(?) \\ o_2 \between o_3 \quad o_2 \between o_4 \quad o_3 \between o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \end{array}} \; E$$

- Known integers transformed to themselves

Transforming values to terms

$$\frac{03 \mid \mathtt{x} : o_2, \mathtt{r} : 1 \mid o_2}{\begin{array}{l} o_2 : \mathtt{L}(\mathtt{p} = o_3, \mathtt{n} = o_4) \\ o_3 : \mathtt{L}(?) \quad o_4 : \mathtt{L}(?) \\ o_2 \between o_3 \quad o_2 \between o_4 \quad o_3 \between o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \end{array}} \; E$$

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

$o_3, o_4 \quad 1$

Transforming values to terms

$$\frac{03 \mid \mathtt{x} : o_2, \mathtt{r} : 1 \mid o_2}{\begin{array}{l} o_2 : \mathtt{L}(\mathtt{p} = o_3, \mathtt{n} = o_4) \\ o_3 : \mathtt{L}(?) \quad o_4 : \mathtt{L}(?) \\ o_2 \not\perp o_3 \quad o_2 \not\perp o_4 \quad o_3 \not\perp o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \end{array}} \; E$$

- 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

$$\overbrace{\mathrm{L}(o_3, o_4)}^{o_2}\ 1$$

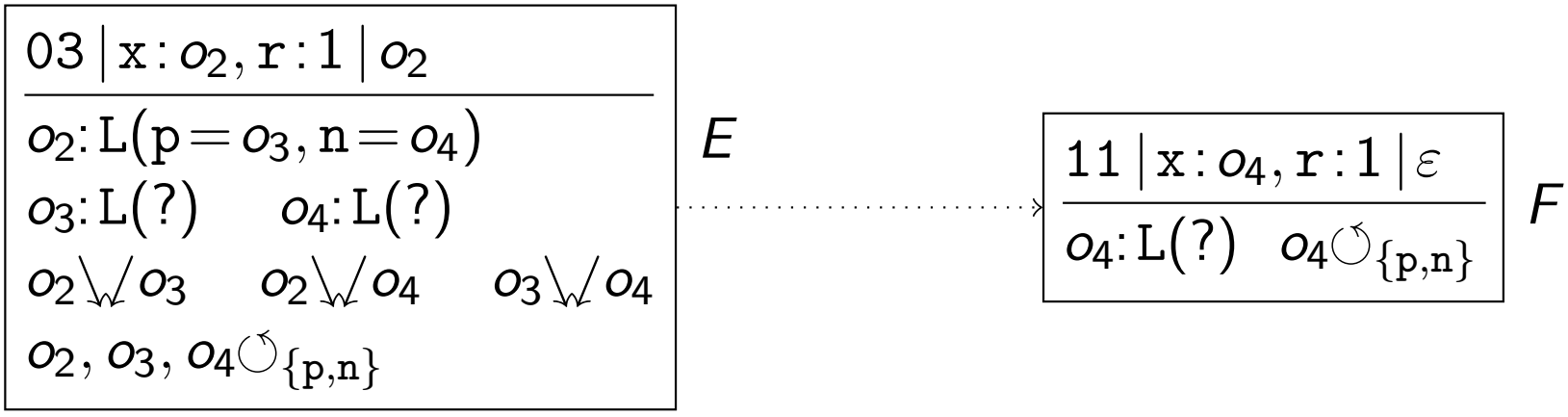
Transforming states to terms

$$\frac{03 \mid \mathtt{x} : o_2, \mathtt{r} : 1 \mid o_2}{\begin{array}{l} o_2 : \mathrm{L}(\mathtt{p} = o_3, \mathtt{n} = o_4) \\ o_3 : \mathrm{L}(?) \quad o_4 : \mathrm{L}(?) \\ o_2 \not\backslash\!/ o_3 \quad o_2 \not\backslash\!/ o_4 \quad o_3 \not\backslash\!/ o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \end{array}} \; E$$

- State s transformed to term with symbol f_s
- All local variables, stack entrys as arguments

$$\mathrm{f}_E(\overbrace{\mathrm{L}(o_3, o_4)}^{o_2}, 1, \overbrace{\mathrm{L}(o_3, o_4)}^{o_2})$$

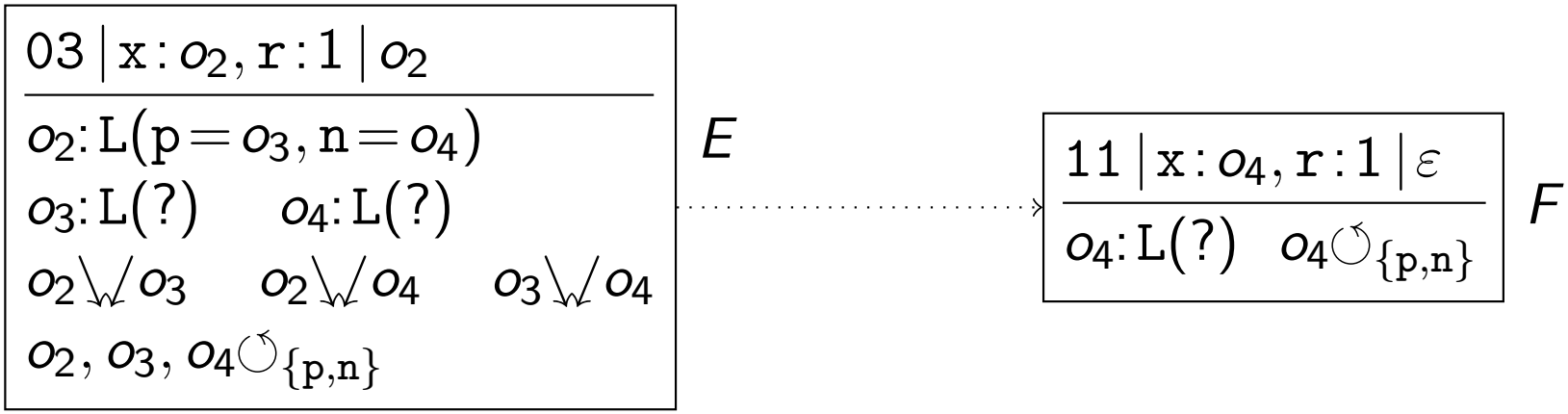
Transforming edges to rules



- State s transformed to term with symbol f_s
- All local variables, stack entrys as arguments

$$\mathsf{f}_E(\overbrace{\mathrm{L}(o_3, o_4)}^{o_2}, 1, \overbrace{\mathrm{L}(o_3, o_4)}^{o_2})$$

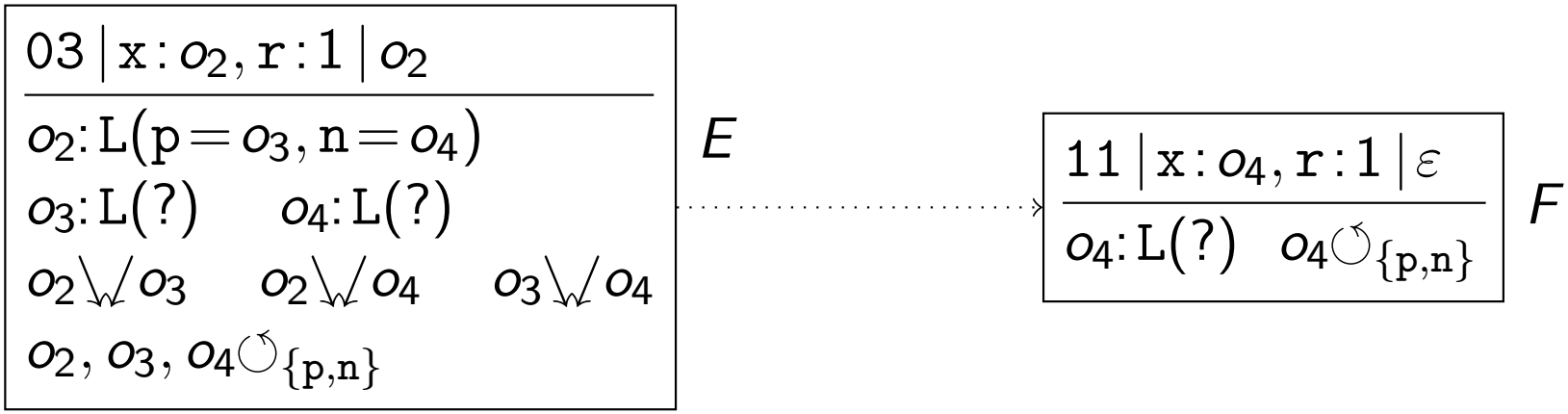
Transforming edges to rules



- State s transformed to term with symbol f_s
- All local variables, stack entrys as arguments
- Evaluation edges: Encode states, put in $\rightarrow$

$$\mathsf{f}_E(\overbrace{\mathsf{L}(o_3, o_4)}^{o_2}, 1, \overbrace{\mathsf{L}(o_3, o_4)}^{o_2}) \quad \rightarrow \quad \mathsf{f}_F(o_4\ , 1)$$

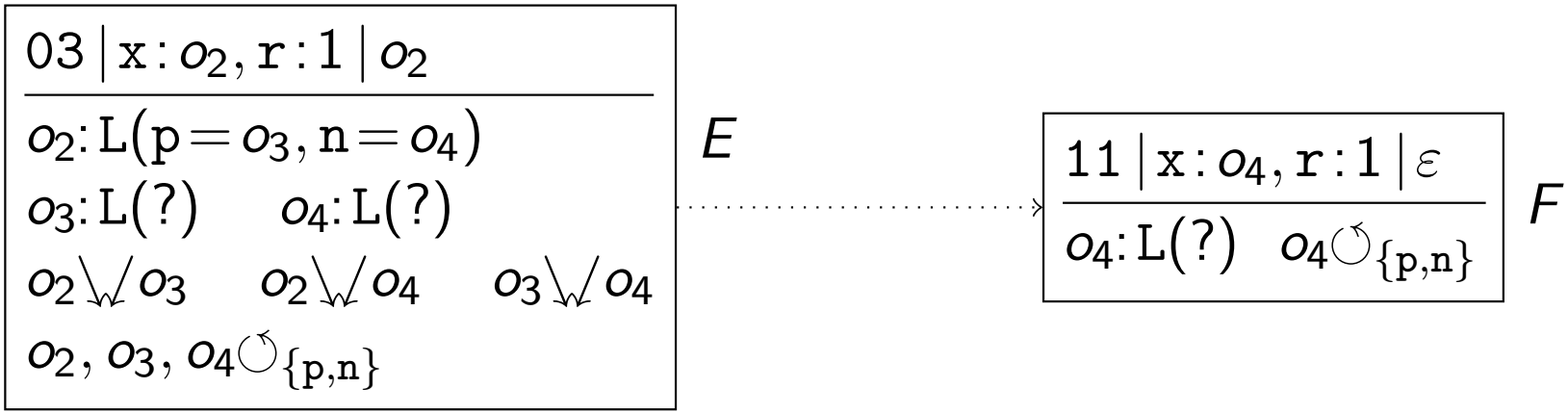
Transforming edges to rules



- State s transformed to term with symbol f_s
- All local variables, stack entrys as arguments
- Evaluation edges: Encode states, put in $\rightarrow$
- Problem: Terms cannot represent cycles $\curvearrowright$ free var on rhs

$$\mathsf{f}_E(\overbrace{\mathsf{L}(o_3, o_4)}^{o_2}, 1, \overbrace{\mathsf{L}(o_3, o_4)}^{o_2}) \quad \rightarrow \quad \mathsf{f}_F(o_4', 1)$$

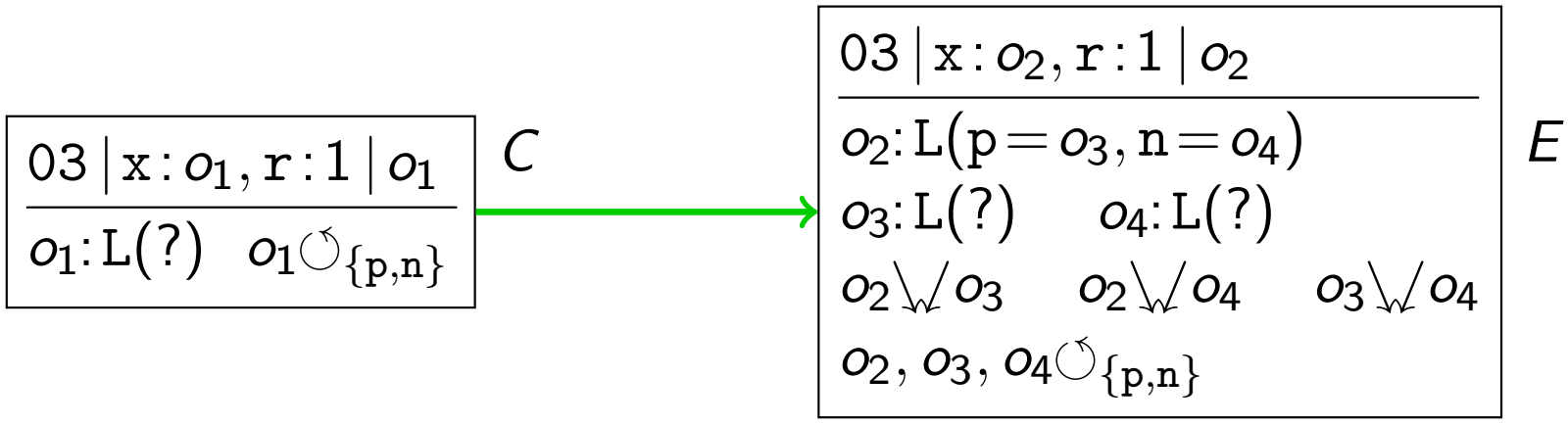
Transforming edges to rules



- State s transformed to term with symbol f_s
- All local variables, stack entrys as arguments
- Evaluation edges: Encode states, put in $\rightarrow$
- Problem: Terms cannot represent cycles $\curvearrowright$ free var on rhs
- Solution: Only encode non-cyclic parts!

$$\mathsf{f}_E(\overbrace{\mathsf{L}(\quad o_4)}^{o_2}, 1, \overbrace{\mathsf{L}(\quad o_4)}^{o_2}) \quad \rightarrow \quad \mathsf{f}_F(o_4\ , 1)$$

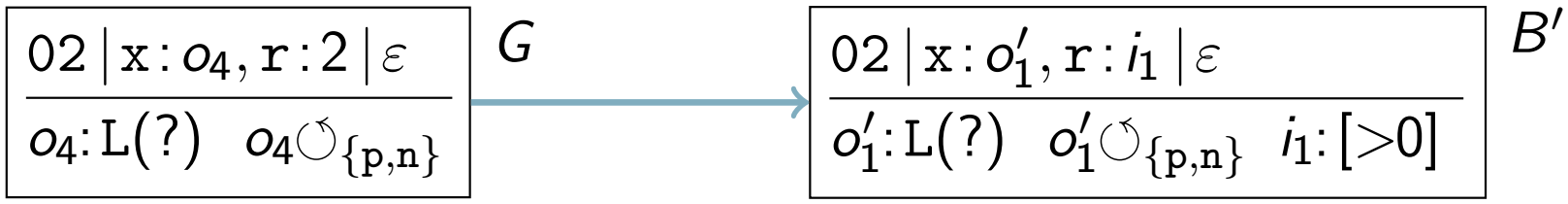
Transforming edges to rules



- State s transformed to term with symbol f_s
- All local variables, stack entrys as arguments
- Evaluation edges: Encode states, put in $\rightarrow$
- Problem: Terms cannot represent cycles $\curvearrowright$ free var on rhs
- Solution: Only encode non-cyclic parts!
- Refinement edges: Encode target state twice, relabel

$$\mathsf{f}_C(\mathsf{L}(o_4), 1, \mathsf{L}(o_4)) \rightarrow \mathsf{f}_E(\mathsf{L}(o_4), 1, \mathsf{L}(o_4))$$

Transforming edges to rules

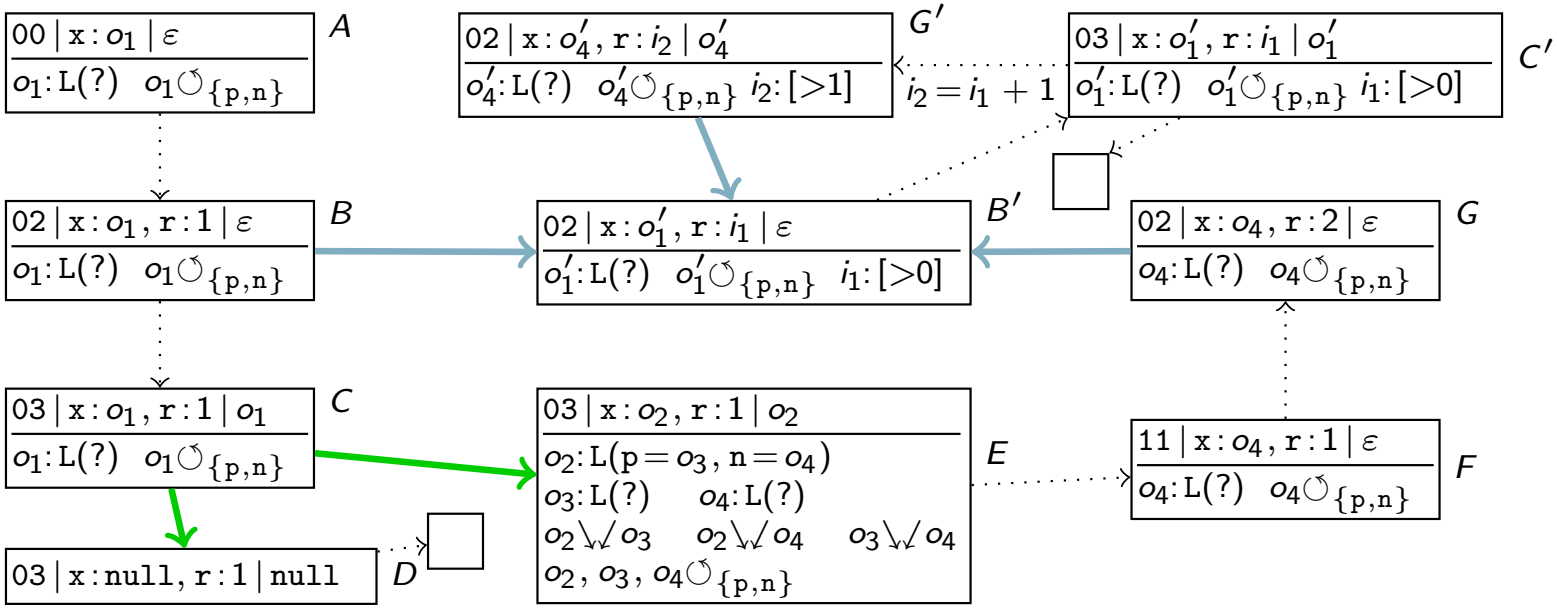


- State s transformed to term with symbol f_s
- All local variables, stack entrys as arguments
- Evaluation edges: Encode states, put in $\rightarrow$
- Problem: Terms cannot represent cycles $\curvearrowright$ free var on rhs
- Solution: Only encode non-cyclic parts!
- Refinement edges: Encode target state twice, relabel
- Instantiation edges: Encode source state twice, relabel

$$f_G(o_4, 2) \rightarrow f_{B'}(o_4, 2)$$

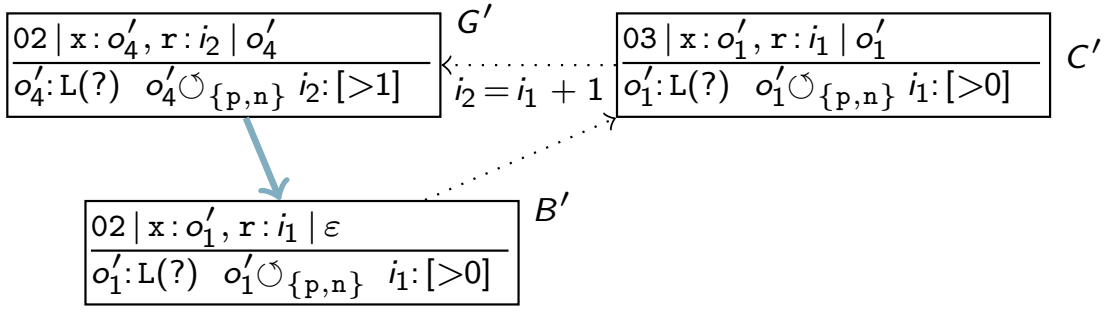
The example TRS

```
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

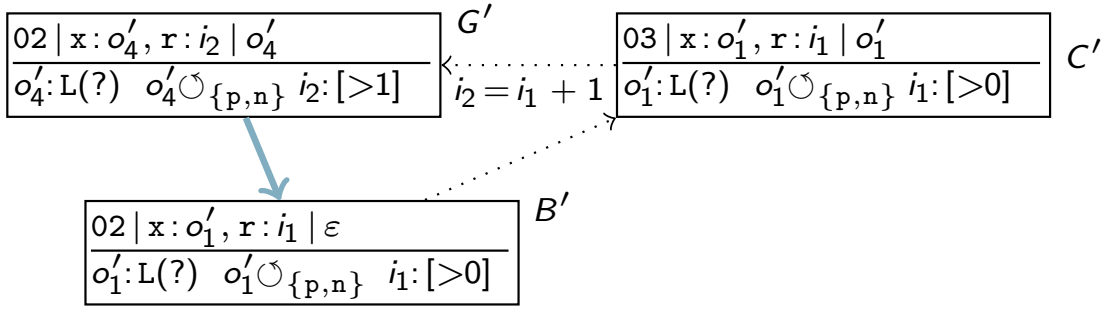
```
00: iconst_1
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```



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
```

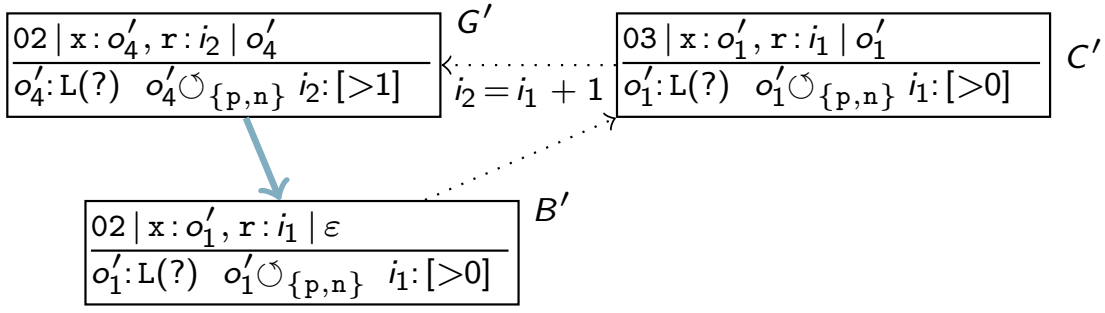


1. Only consider SCCs!
2. Transform all edges as before, simplify:

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

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:

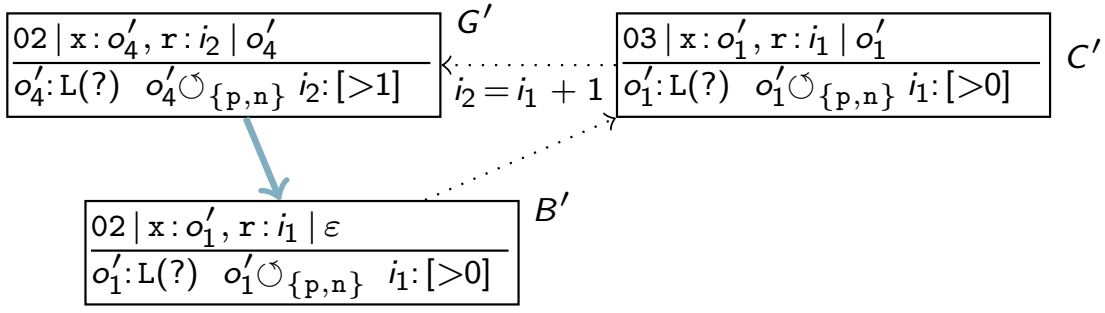
$$\mathsf{f}_{B'}(\mathsf{L}(o'_4), i_1) \quad \to \quad \mathsf{f}_{B'}(o'_4, i_1 + 1)$$

3. Termination trivially proven with

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

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

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

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 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 detecting (and ignoring) irrelevant cyclicity
    - by automatically finding and counting markers
- *Non*-termination analysis [FoVeOOS'11]

Plans for AProVE

- Integrate existing shape analyses

Plans for AProVE

- Integrate existing shape analyses
- Modularize analysis: pre-compute information for common libraries

Plans for AProVE

- Integrate existing shape analyses
- Modularize analysis: pre-compute information for common libraries
- Handle bounded integers properly

Plans for AProVE

- Integrate existing shape analyses
- Modularize analysis: pre-compute information for common libraries
- Handle bounded integers properly
- Extend to C with pointer arithmetic

Plans for AProVE

- Integrate existing shape analyses
- Modularize analysis: pre-compute information for common libraries
- Handle bounded integers properly
- Extend to C with pointer arithmetic
- Asymptotic runtime complexity analysis (via TRS)

Plans for AProVE

- Integrate existing shape analyses
- Modularize analysis: pre-compute information for common libraries
- Handle bounded integers properly
- Extend to C with pointer arithmetic
- Asymptotic runtime complexity analysis (via TRS)
- Provide to developers as Eclipse plugin

Proving Java Termination: Reducing the Ugly to the Bad

- Evaluated on collection of 387 programs (including the *Termination Problem Data Base*):

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

Proving Java Termination: Reducing the Ugly to the Bad

- Evaluated on collection of 387 programs (including the *Termination Problem Data Base*):

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

- Won Termination Competition 2012

Proving Java Termination: Reducing the Ugly to the Bad

- Evaluated on collection of 387 programs (including the *Termination Problem Data Base*):

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

- Won Termination Competition 2012
- Symbolic Evaluation Graphs facilitate and simplify complex analyses