

Modular Termination Analysis for JAVA BYTECODE by Term Rewriting

Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

joint work with C. Otto and M. Brockschmidt

Automated Termination Tools for TRSs

- AProVE (*Aachen*)
- CARIBOO (*Nancy*)
- CiME (*Orsay*)
- Jambox (*Amsterdam*)
- Matchbox (*Leipzig*)
- MU-TERM (*Valencia*)
- MultumNonMultum (*Kassel*)
- TEPARLA (*Eindhoven*)
- Termptation (*Barcelona*)
- TORPA (*Eindhoven*)
- TPA (*Eindhoven*)
- TTT (*Innsbruck*)
- VMTL (*Vienna*)

Automated Termination Tools for TRSs

- AProVE (*Aachen*)
- CARIBOO (*Nancy*)
- CiME (*Orsay*)
- Jambox (*Amsterdam*)
- Matchbox (*Leipzig*)
- MU-TERM (*Valencia*)
- MultumNonMulta (*Kassel*)
- TEPARLA (*Eindhoven*)
- Termptation (*Barcelona*)
- TORPA (*Eindhoven*)
- TPA (*Eindhoven*)
- TTT (*Innsbruck*)
- VMTL (*Vienna*)
- *Annual International Competition of Termination Tools*
- well-developed field
- active research
- powerful techniques & tools

Automated Termination Tools for TRSs

- AProVE (*Aachen*)
- CARIBOO (*Nancy*)
- CiME (*Orsay*)
- Jambox (*Amsterdam*)
- Matchbox (*Leipzig*)
- MU-TERM (*Valencia*)
- MultumNonMulta (*Kassel*)
- TEPARLA (*Eindhoven*)
- Termtation (*Barcelona*)
- TORPA (*Eindhoven*)
- TPA (*Eindhoven*)
- TTT (*Innsbruck*)
- VMTL (*Vienna*)
- *Annual International Competition of Termination Tools*
- well-developed field
- active research
- powerful techniques & tools
- **But:**
What about application in practice?

Termination of Imperative Programs

Direct Approaches

Termination of Imperative Programs

Direct Approaches

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

Termination of Imperative Programs

Direct Approaches

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

Termination of Imperative Programs

Direct Approaches

- Synthesis of Linear Ranking Functions
(Colon & Sipma, 01), (Podelski & Rybalchenko, 04), ...
- **Terminator**: Termination Analysis by Abstraction & Model Checking
(Cook, Podelski, Rybalchenko et al., since 05)
- **Julia** & **COSTA**: Termination Analysis of JAVA BYTECODE
*(Spoto, Mesnard, Payet, 10),
(Albert, Arenas, Codish, Genaim, Puebla, Zanardini, 08)*

Termination of Imperative Programs

Direct Approaches

- Synthesis of Linear Ranking Functions
(Colon & Sipma, 01), (Podelski & Rybalchenko, 04), ...
- **Terminator**: Termination Analysis by Abstraction & Model Checking
(Cook, Podelski, Rybalchenko et al., since 05)
- **Julia** & **COSTA**: Termination Analysis of JAVA BYTECODE
*(Spoto, Mesnard, Payet, 10),
(Albert, Arenas, Codish, Genaim, Puebla, Zanardini, 08)*
- ...

Termination of Imperative Programs

Direct Approaches

- Synthesis of Linear Ranking Functions
(Colon & Sipma, 01), (Podelski & Rybalchenko, 04), ...
 - **Terminator**: Termination Analysis by Abstraction & Model Checking
(Cook, Podelski, Rybalchenko et al., since 05)
 - **Julia** & **COSTA**: Termination Analysis of JAVA BYTECODE
*(Spoto, Mesnard, Payet, 10),
(Albert, Arenas, Codish, Genaim, Puebla, Zanardini, 08)*
 - ...
-
- used at Microsoft for verifying Windows device drivers

Termination of Imperative Programs

Direct Approaches

- Synthesis of Linear Ranking Functions
(Colon & Sipma, 01), (Podelski & Rybalchenko, 04), ...
 - **Terminator**: Termination Analysis by Abstraction & Model Checking
(Cook, Podelski, Rybalchenko et al., since 05)
 - **Julia** & **COSTA**: Termination Analysis of JAVA BYTECODE
*(Spoto, Mesnard, Payet, 10),
(Albert, Arenas, Codish, Genaim, Puebla, Zanardini, 08)*
 - ...
-
- used at Microsoft for verifying Windows device drivers
 - **no use of TRS-techniques** (stand-alone methods)

Termination of Imperative Programs

Rewrite-Based Approach

Termination of Imperative Programs

Rewrite-Based Approach

- analyze JAVA BYTECODE (JBC) instead of JAVA
- using TRS-techniques for JBC is challenging

Termination of Imperative Programs

Rewrite-Based Approach

- analyze JAVA BYTECODE (JBC) instead of JAVA
- using TRS-techniques for JBC is challenging
 - sharing and aliasing
 - side effects
 - cyclic data objects
 - object-orientation
 - recursion
 - ...

Termination of Imperative Programs

- **New approach** (RTA '10)

Termination of Imperative Programs

- **New approach** (RTA '10)
 - **Frontend**
 - evaluate JBC a few steps \Rightarrow **termination graph**
termination graph captures side effects, sharing, cyclic data objects etc.

Termination of Imperative Programs

- **New approach** (RTA '10)
 - **Frontend**
 - evaluate JBC a few steps \Rightarrow **termination graph**
termination graph captures side effects, sharing, cyclic data objects etc.
 - transform **termination graph** \Rightarrow **TRS**

Termination of Imperative Programs

- **New approach** (RTA '10)
 - **Frontend**
 - evaluate JBC a few steps \Rightarrow **termination graph**
termination graph captures side effects, sharing, cyclic data objects etc.
 - transform **termination graph** \Rightarrow **TRS**
 - **Backend**
 - prove termination of the resulting TRS
(using existing techniques & tools)

Termination of Imperative Programs

- **New approach** (RTA '10)
 - **Frontend**
 - evaluate JBC a few steps \Rightarrow **termination graph**
termination graph captures side effects, sharing, cyclic data objects etc.
 - transform **termination graph** \Rightarrow **TRS**
 - **Backend**
 - prove termination of the resulting TRS
(using existing techniques & tools)
- implemented in **AProVE**

Termination of Imperative Programs

- **New approach** (RTA '10, RTA '11)
 - **Frontend**
 - evaluate JBC a few steps \Rightarrow **termination graph**
termination graph captures side effects, sharing, cyclic data objects etc.
 - transform **termination graph** \Rightarrow **TRS**
 - **Backend**
 - prove termination of the resulting TRS
(using existing techniques & tools)
- implemented in **AProVE**
 - modular termination graphs (separate graphs for separate methods)
 \Rightarrow scalability & recursion

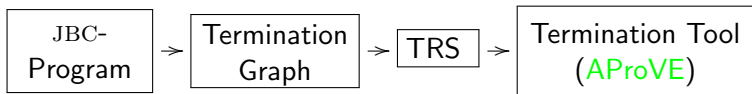
Termination of Imperative Programs

- **New approach** (RTA '10, RTA '11)
 - **Frontend**
 - evaluate JBC a few steps \Rightarrow **termination graph**
termination graph captures side effects, sharing, cyclic data objects etc.
 - transform **termination graph** \Rightarrow **TRS**
 - **Backend**
 - prove termination of the resulting TRS
(using existing techniques & tools)
- implemented in **AProVE**
 - modular termination graphs (separate graphs for separate methods)
 \Rightarrow scalability & recursion
 - successfully evaluated on JBC-collection

Termination of Imperative Programs

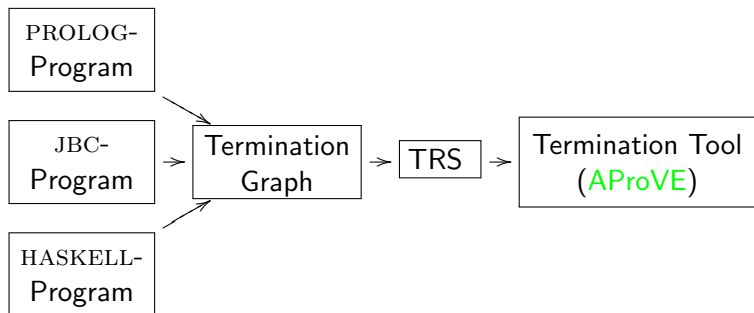
- **New approach** (RTA '10, RTA '11)
 - **Frontend**
 - evaluate JBC a few steps \Rightarrow **termination graph**
termination graph captures side effects, sharing, cyclic data objects etc.
 - transform **termination graph** \Rightarrow **TRS**
 - **Backend**
 - prove termination of the resulting TRS
(using existing techniques & tools)
- implemented in **AProVE**
 - modular termination graphs (separate graphs for separate methods)
 \Rightarrow scalability & recursion
 - successfully evaluated on JBC-collection
 - competitive termination tool for JBC

Termination of Imperative Programs



- implemented in **AProVE**
 - modular termination graphs (separate graphs for separate methods)
⇒ scalability & recursion
 - successfully evaluated on JBC-collection
 - competitive termination tool for JBC

Termination of Imperative Programs



- implemented in **AProVE**
 - modular termination graphs (separate graphs for separate methods)
⇒ scalability & recursion
 - successfully evaluated on JBC-collection
 - competitive termination tool for JBC

Termination of Imperative Programs

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

Termination of Imperative Programs

other techniques:

abstract objects to numbers

- List-object representing $[0, 1, 2]$ is abstracted to length 3

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

Termination of Imperative Programs

other techniques:

abstract objects to **numbers**

- List-object representing $[0, 1, 2]$ is abstracted to **length 3**

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

our technique:

abstract objects to **terms**

Termination of Imperative Programs

other techniques:

abstract objects to **numbers**

- List-object representing $[0, 1, 2]$ is abstracted to **length 3**

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

our technique:

abstract objects to **terms**

- introduce function symbol for every class

Termination of Imperative Programs

other techniques:

abstract objects to **numbers**

- List-object representing $[0, 1, 2]$ is abstracted to **length 3**

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

our technique:

abstract objects to **terms**

- introduce function symbol for every class
- List-object representing $[0, 1, 2]$ is abstracted to **term**: `List(0, List(1, List(2, null)))`

Termination of Imperative Programs

other techniques:

abstract objects to **numbers**

- List-object representing $[0, 1, 2]$ is abstracted to **length 3**

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

our technique:

abstract objects to **terms**

- introduce function symbol for every class
- List-object representing $[0, 1, 2]$ is abstracted to **term**: `List(0, List(1, List(2, null)))`
- TRS-techniques generate suitable orders to compare arbitrary terms

Termination of Imperative Programs

other techniques:

abstract objects to **numbers**

- List-object representing $[0, 1, 2]$ is abstracted to **length 3**

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

our technique:

abstract objects to **terms**

- introduce function symbol for every class
- List-object representing $[0, 1, 2]$ is abstracted to **term**: `List(0, List(1, List(2, null)))`
- TRS-techniques generate suitable orders to compare arbitrary terms
- particularly powerful on **user-defined data types**

Termination of Imperative Programs

other techniques:

abstract objects to **numbers**

- List-object representing $[0, 1, 2]$ is abstracted to **length 3**

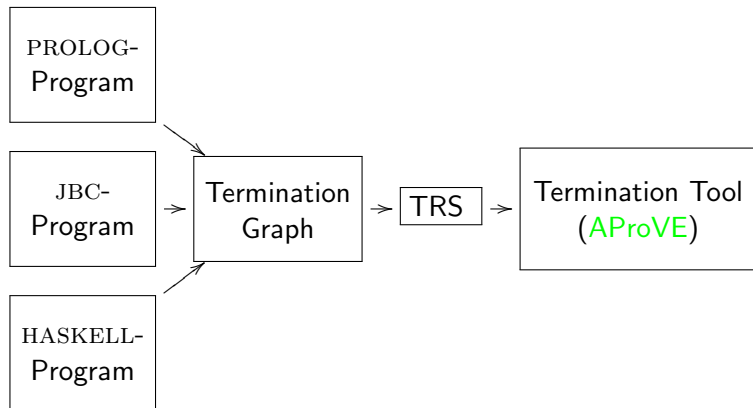
```
public class List {  
    int value;  
    List next;  
}
```

our technique:

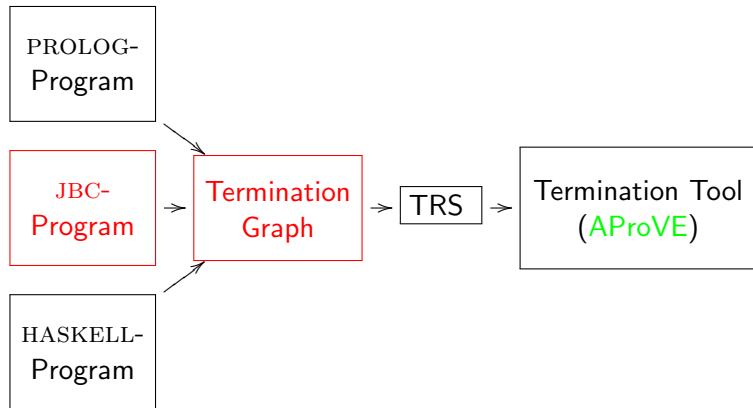
abstract objects to **terms**

- introduce function symbol for every class
- List-object representing $[0, 1, 2]$ is abstracted to **term**: `List(0, List(1, List(2, null)))`
- TRS-techniques generate suitable orders to compare arbitrary terms
- particularly powerful on **user-defined data types**
- powerful on **pre-defined data types** by using **Integer TRSs (RTA '09)**

From JBC to Termination Graphs



From JBC to Termination Graphs



Example

```
class List {
    List n;

    public void appE(int i) {
        if (n == null) {
            if (i <= 0) return;
            n = new List();
            i--;
        }
        n.appE(i);
    }
}
```

Example

```
00: aload_0      // load this to opstack
01: getfield n   // load this.n to opstack
04: ifnonnull 26 // go to 26 if n != null
07: iload_1      // load i to opstack
08: ifgt 12      // go to 12 if i > 0
11: return       // return (without value)
12: aload_0      // load this to opstack
13: new List     // create new List object
16: dup         // duplicate top of stack
17: invokespecial <init> // call constructor
20: putfield n   // write new List to n
23: iinc 1, -1   // decrement i by 1
26: aload_0      // load this to opstack
27: getfield n   // load this.n to opstack
30: iload_1      // load i to opstack
31: invokevirtual appE // recursive call
34: return      // return (without value)
```

```
class List {
    List n;

    public void appE(int i) {
        if (n == null) {
            if (i <= 0) return;
            n = new List();
            i--;
        }
        n.appE(i);
    }
}
```

Abstract States of the JVM

```
00: aload_0      // load this to opstack
01: getfield n   // load this.n to opstack
04: ifnonnull 26 // go to 26 if n != null
07: iload_1     // load i to opstack
08: ifgt 12     // go to 12 if i > 0
11: return      // return (without value)
12: aload_0     // load this to opstack
13: new List    // create new List object
16: dup        // duplicate top of stack
17: invokespecial <init> // call constructor
20: putfield n   // write new List to n
23: iinc 1, -1  // decrement i by 1
26: aload_0     // load this to opstack
27: getfield n   // load this.n to opstack
30: iload_1     // load i to opstack
31: invokevirtual appE // recursive call
34: return      // return (without value)
```

$$o_1, i_3 \mid 0 \mid t:o_1, i:i_3 \mid \varepsilon$$
$$o_1:\text{List}(n=o_2) \quad i_3:\mathbb{Z}$$
$$o_2:\text{List}(?)$$

Abstract States of the JVM

```
00: aload_0      // load this to opstack
01: getfield n   // load this.n to opstack
04: ifnonnull 26 // go to 26 if n != null
07: iload_1     // load i to opstack
08: ifgt 12     // go to 12 if i > 0
11: return      // return (without value)
12: aload_0     // load this to opstack
13: new List    // create new List object
16: dup        // duplicate top of stack
17: invokespecial <init> // call constructor
20: putfield n   // write new List to n
23: iinc 1, -1  // decrement i by 1
26: aload_0     // load this to opstack
27: getfield n   // load this.n to opstack
30: iload_1     // load i to opstack
31: invokevirtual appE // recursive call
34: return      // return (without value)
```

$o_1, i_3 \mid 0 \mid t:o_1, i:i_3 \mid \varepsilon$

$o_1:\text{List}(n=o_2) \quad i_3:\mathbb{Z}$

$o_2:\text{List}(?)$

stack frame

Abstract States of the JVM

```
00: aload_0      // load this to opstack
01: getfield n   // load this.n to opstack
04: ifnonnull 26 // go to 26 if n != null
07: iload_1      // load i to opstack
08: ifgt 12      // go to 12 if i > 0
11: return       // return (without value)
12: aload_0      // load this to opstack
13: new List     // create new List object
16: dup          // duplicate top of stack
17: invokespecial <init> // call constructor
20: putfield n   // write new List to n
23: iinc 1, -1   // decrement i by 1
26: aload_0      // load this to opstack
27: getfield n   // load this.n to opstack
30: iload_1      // load i to opstack
31: invokevirtual appE // recursive call
34: return       // return (without value)
```

$o_1, i_3 \mid 0 \mid t:o_1, i:i_3 \mid \varepsilon$

$o_1:\text{List}(n=o_2) \quad i_3:\mathbb{Z}$

$o_2:\text{List}(?)$

stack frame

① input arguments

Abstract States of the JVM

```
00: aload_0      // load this to opstack
01: getfield n   // load this.n to opstack
04: ifnonnull 26 // go to 26 if n != null
07: iload_1      // load i to opstack
08: ifgt 12      // go to 12 if i > 0
11: return       // return (without value)
12: aload_0      // load this to opstack
13: new List     // create new List object
16: dup          // duplicate top of stack
17: invokespecial <init> // call constructor
20: putfield n   // write new List to n
23: iinc 1, -1   // decrement i by 1
26: aload_0      // load this to opstack
27: getfield n   // load this.n to opstack
30: iload_1      // load i to opstack
31: invokevirtual appE // recursive call
34: return       // return (without value)
```

$\sigma_1, i_3 \mid 0 \mid \tau: \sigma_1, i: i_3 \mid \varepsilon$

$\sigma_1: \text{List}(n = \sigma_2) \quad i_3: \mathbb{Z}$

$\sigma_2: \text{List}(?)$

stack frame

- 1 input arguments
- 2 next program instruction

Abstract States of the JVM

```
00: aload_0      // load this to opstack
01: getfield n   // load this.n to opstack
04: ifnonnull 26 // go to 26 if n != null
07: iload_1     // load i to opstack
08: ifgt 12     // go to 12 if i > 0
11: return      // return (without value)
12: aload_0     // load this to opstack
13: new List    // create new List object
16: dup        // duplicate top of stack
17: invokespecial <init> // call constructor
20: putfield n  // write new List to n
23: iinc 1, -1 // decrement i by 1
26: aload_0     // load this to opstack
27: getfield n  // load this.n to opstack
30: iload_1     // load i to opstack
31: invokevirtual appE // recursive call
34: return     // return (without value)
```

$\sigma_1, i_3 \mid 0 \mid \tau: \sigma_1, i: i_3 \mid \varepsilon$

$\sigma_1: \text{List}(n = \sigma_2) \quad i_3: \mathbb{Z}$

$\sigma_2: \text{List}(?)$

stack frame

- 1 input arguments
- 2 next program instruction
- 3 values of local variables
(value of this is *reference* σ_1)

Abstract States of the JVM

```
00: aload_0      // load this to opstack
01: getfield n   // load this.n to opstack
04: ifnonnull 26 // go to 26 if n != null
07: iload_1     // load i to opstack
08: ifgt 12     // go to 12 if i > 0
11: return      // return (without value)
12: aload_0     // load this to opstack
13: new List    // create new List object
16: dup        // duplicate top of stack
17: invokespecial <init> // call constructor
20: putfield n   // write new List to n
23: iinc 1, -1  // decrement i by 1
26: aload_0     // load this to opstack
27: getfield n   // load this.n to opstack
30: iload_1     // load i to opstack
31: invokevirtual appE // recursive call
34: return      // return (without value)
```

$o_1, i_3 \mid 0 \mid t:o_1, i:i_3 \mid \varepsilon$

$o_1:\text{List}(n=o_2) \quad i_3:\mathbb{Z}$

$o_2:\text{List}(?)$

stack frame

- 1 input arguments
- 2 next program instruction
- 3 values of local variables
(value of this is *reference* o_1)
- 4 values on the operand stack

Abstract States of the JVM

```
00: aload_0      // load this to opstack
01: getfield n    // load this.n to opstack
04: ifnonnull 26  // go to 26 if n != null
07: iload_1      // load i to opstack
08: ifgt 12      // go to 12 if i > 0
11: return       // return (without value)
12: aload_0      // load this to opstack
13: new List     // create new List object
16: dup         // duplicate top of stack
17: invokespecial <init> // call constructor
20: putfield n    // write new List to n
23: iinc 1, -1   // decrement i by 1
26: aload_0      // load this to opstack
27: getfield n    // load this.n to opstack
30: iload_1      // load i to opstack
31: invokevirtual appE // recursive call
34: return       // return (without value)
```

$o_1, i_3 \mid 0 \mid t:o_1, i:i_3 \mid \varepsilon$

$o_1:List(n=o_2) \quad i_3:\mathbb{Z}$

$o_2:List(?)$

stack frame

- 1 input arguments
- 2 next program instruction
- 3 values of local variables
(value of this is *reference* o_1)
- 4 values on the operand stack

information about the heap

Abstract States of the JVM

```
00: aload_0      // load this to opstack
01: getfield n    // load this.n to opstack
04: ifnonnull 26  // go to 26 if n != null
07: iload_1      // load i to opstack
08: ifgt 12      // go to 12 if i > 0
11: return       // return (without value)
12: aload_0      // load this to opstack
13: new List     // create new List object
16: dup         // duplicate top of stack
17: invokespecial <init> // call constructor
20: putfield n    // write new List to n
23: iinc 1, -1   // decrement i by 1
26: aload_0      // load this to opstack
27: getfield n    // load this.n to opstack
30: iload_1      // load i to opstack
31: invokevirtual appE // recursive call
34: return       // return (without value)
```

$o_1, i_3 \mid 0 \mid t:o_1, i:i_3 \mid \varepsilon$

$o_1:\text{List}(n=o_2) \quad i_3:\mathbb{Z}$

$o_2:\text{List}(?)$

stack frame

- 1 input arguments
- 2 next program instruction
- 3 values of local variables
(value of this is *reference* o_1)
- 4 values on the operand stack

information about the heap

- object at o_1 has type List, n-field has value o_2

Abstract States of the JVM

```
00: aload_0      // load this to opstack
01: getfield n    // load this.n to opstack
04: ifnonnull 26  // go to 26 if n != null
07: iload_1      // load i to opstack
08: ifgt 12      // go to 12 if i > 0
11: return       // return (without value)
12: aload_0      // load this to opstack
13: new List     // create new List object
16: dup         // duplicate top of stack
17: invokespecial <init> // call constructor
20: putfield n    // write new List to n
23: iinc 1, -1   // decrement i by 1
26: aload_0      // load this to opstack
27: getfield n    // load this.n to opstack
30: iload_1      // load i to opstack
31: invokevirtual appE // recursive call
34: return       // return (without value)
```

$o_1, i_3 \mid 0 \mid t:o_1, i:i_3 \mid \varepsilon$

$o_1:\text{List}(n=o_2) \quad i_3:\mathbb{Z}$

$o_2:\text{List}(?)$

stack frame

- 1 input arguments
- 2 next program instruction
- 3 values of local variables
(value of this is *reference* o_1)
- 4 values on the operand stack

information about the heap

- object at o_1 has type List, n-field has value o_2
- object at address o_2 is null or of type List

Abstract States of the JVM

```
00: aload_0      // load this to opstack
01: getfield n    // load this.n to opstack
04: ifnonnull 26  // go to 26 if n != null
07: iload_1      // load i to opstack
08: ifgt 12      // go to 12 if i > 0
11: return       // return (without value)
12: aload_0      // load this to opstack
13: new List     // create new List object
16: dup         // duplicate top of stack
17: invokespecial <init> // call constructor
20: putfield n    // write new List to n
23: iinc 1, -1   // decrement i by 1
26: aload_0      // load this to opstack
27: getfield n    // load this.n to opstack
30: iload_1      // load i to opstack
31: invokevirtual appE // recursive call
34: return       // return (without value)
```

$o_1, i_3 \mid 0 \mid t:o_1, i:i_3 \mid \varepsilon$

$o_1:\text{List}(n=o_2) \quad i_3:\mathbb{Z}$

$o_2:\text{List}(?)$

stack frame

- 1 input arguments
- 2 next program instruction
- 3 values of local variables
(value of this is *reference* o_1)
- 4 values on the operand stack

information about the heap

- object at o_1 has type List, n-field has value o_2
- object at address o_2 is null or of type List
- i_3 is an arbitrary integer

Abstract States of the JVM

```
00: aload_0      // load this to opstack
01: getfield n    // load this.n to opstack
04: ifnonnull 26  // go to 26 if n != null
07: iload_1      // load i to opstack
08: ifgt 12      // go to 12 if i > 0
11: return       // return (without value)
12: aload_0      // load this to opstack
13: new List     // create new List object
16: dup         // duplicate top of stack
17: invokespecial <init> // call constructor
20: putfield n    // write new List to n
23: iinc 1, -1   // decrement i by 1
26: aload_0      // load this to opstack
27: getfield n    // load this.n to opstack
30: iload_1      // load i to opstack
31: invokevirtual appE // recursive call
34: return       // return (without value)
```

$o_1, i_3 \mid 0 \mid t:o_1, i:i_3 \mid \varepsilon$

$o_1:\text{List}(n=o_2) \quad i_3:\mathbb{Z}$

$o_2:\text{List}(?)$

stack frame

- 1 input arguments
- 2 next program instruction
- 3 values of local variables
(value of this is *reference* o_1)
- 4 values on the operand stack

information about the heap

- object at o_1 has type List, n-field has value o_2
- object at address o_2 is null or of type List
- i_3 is an arbitrary integer

explicit sharing
information

```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```

$\sigma_1, i_3 \mid 0 \mid t: \sigma_1, i: i_3 \mid \varepsilon$ $\sigma_1: \text{List}(n = \sigma_2) \quad i_3: \mathbb{Z}$ $\sigma_2: \text{List}(?)$

A

State A:

- do all calls of appE terminate?
- this is an arbitrary acyclic List
- i is an arbitrary integer


```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```

$\sigma_1, i_3 \mid 0 \mid t: \sigma_1, i: i_3 \mid \varepsilon$ $\sigma_1: \text{List}(n = \sigma_2) \quad i_3: \mathbb{Z}$ $\sigma_2: \text{List}(?)$	A
---	---

$\sigma_1, i_3 \mid 4 \mid t: \sigma_1, i: i_3 \mid \sigma_2$ $\sigma_1: \text{List}(n = \sigma_2) \quad i_3: \mathbb{Z}$ $\sigma_2: \text{List}(?)$	B
--	---

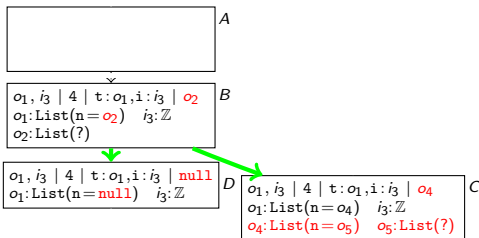
State B:

- “aload_0” loads value σ_1 of this on opstack
- “getfield n” replaces σ_1 by σ_2 on opstack (value of its n-field)
- A connected to B by *evaluation edge*

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



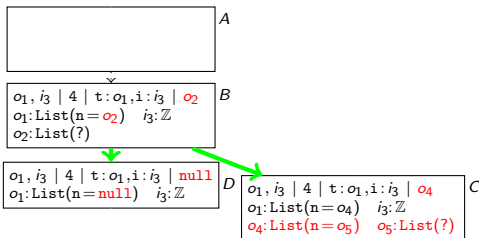
States C and D:

- “ifnonnull 26” needs to know whether o_2 is null
- *refine* information about heap (*refinement edges*)

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



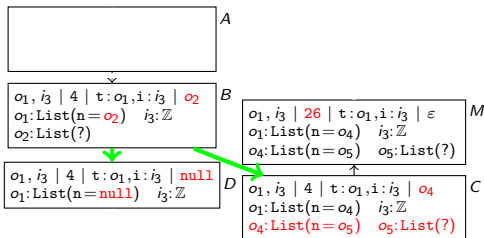
States C and D:

- “ifnonnull 26” needs to know whether o_2 is null
- *refine* information about heap (*refinement edges*)
- in C, replace o_2 by “ $o_4 : \text{List}(n = o_5)$ ”

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



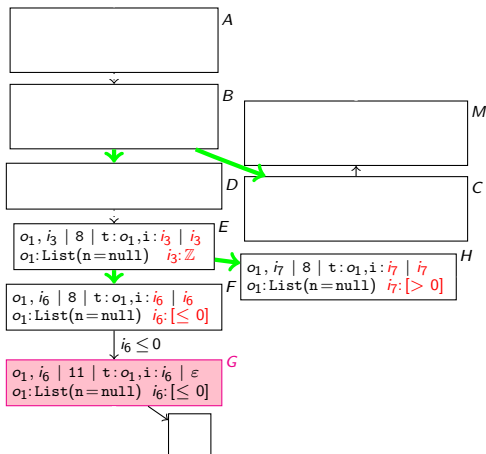
States C and D:

- “ifnonnull 26” needs to know whether o_2 is null
- refine information about heap (*refinement edges*)
- in C, replace o_2 by “ $o_4 : \text{List}(n = o_5)$ ”, evaluation to M

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



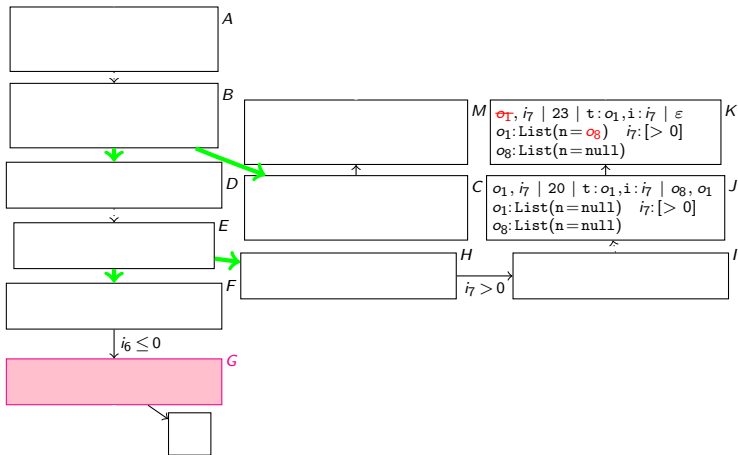
State G:

- “ifgt 12” needs to know whether $i_3 > 0$
- refine information about heap (*refinement edges*)
- evaluation to *return state G*

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



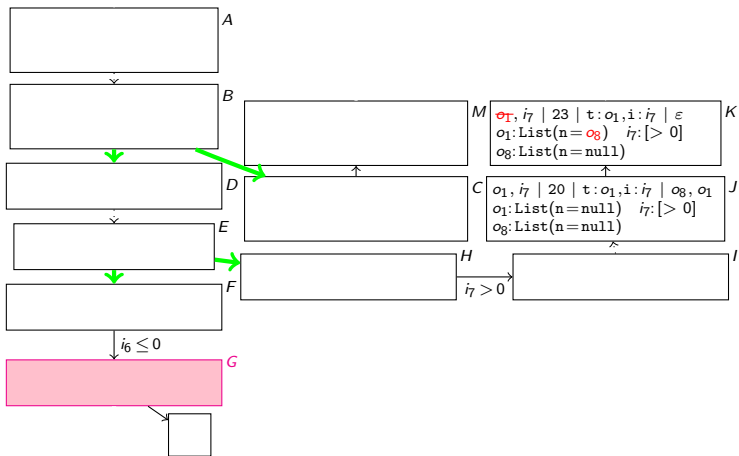
State K:

- “putfield n” writes o_8 to n-field of o_1

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



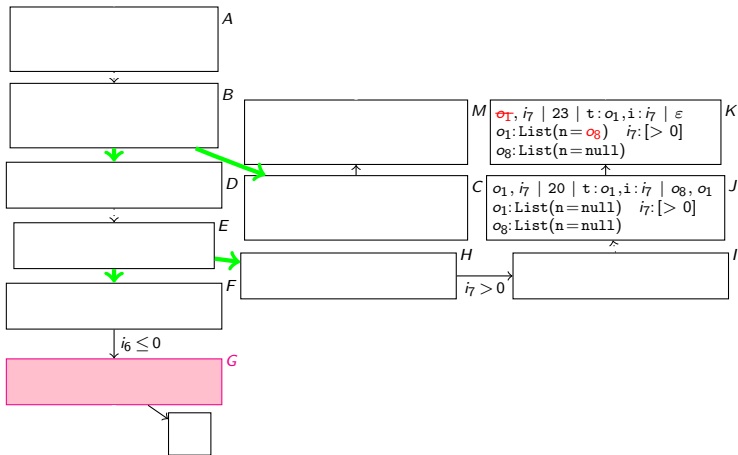
State K:

- “putfield n” writes o_8 to n-field of o_1
- *side effect* which changes original *input argument* o_1

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



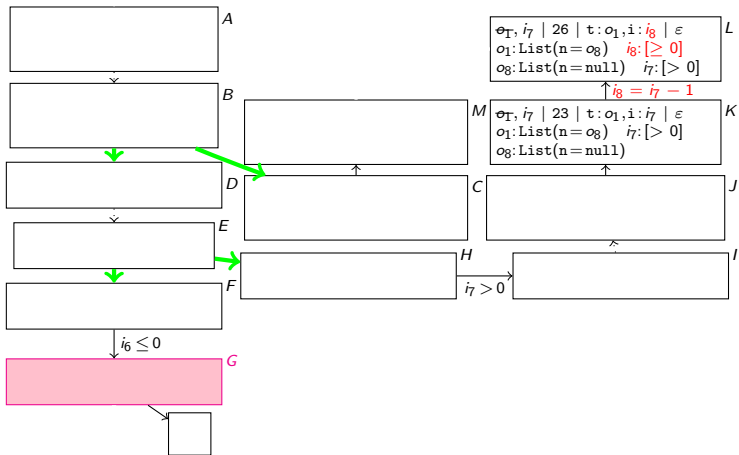
State K:

- “putfield n” writes σ_8 to n-field of σ_1
- *side effect* which changes original *input argument* σ_1
- switch boolean flag of input argument σ_1 to *false*


```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



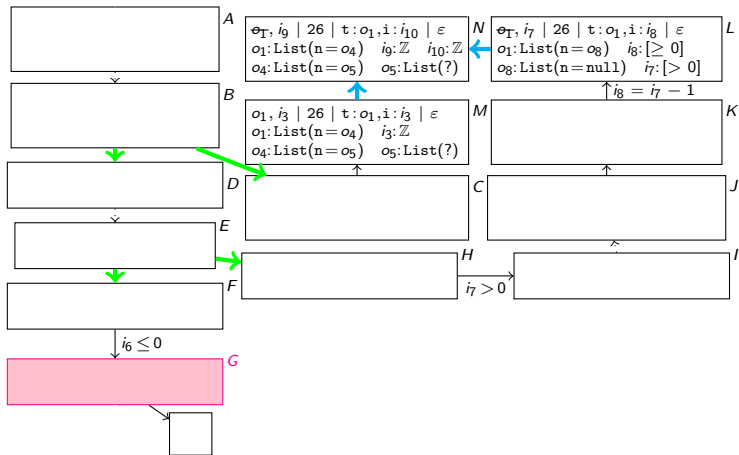
State L:

- decrement i_7 by 1

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



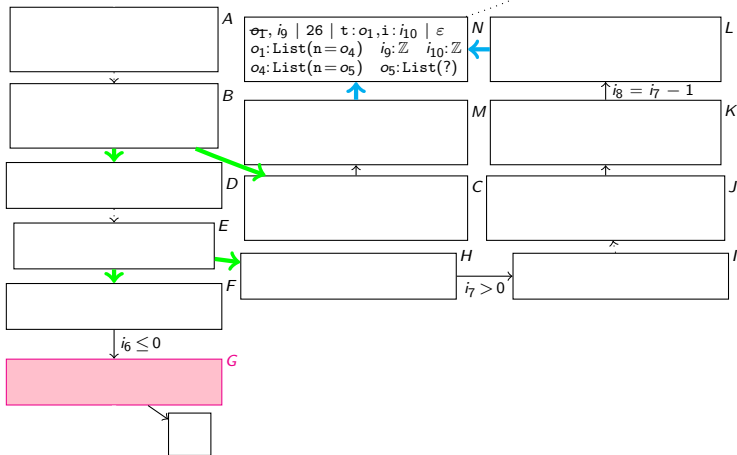
State N:

- L and M are *similar*
- *generalize* them to state N , which represents a superset of L and M
- L and M are *instances* of N (*instance edges*)

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



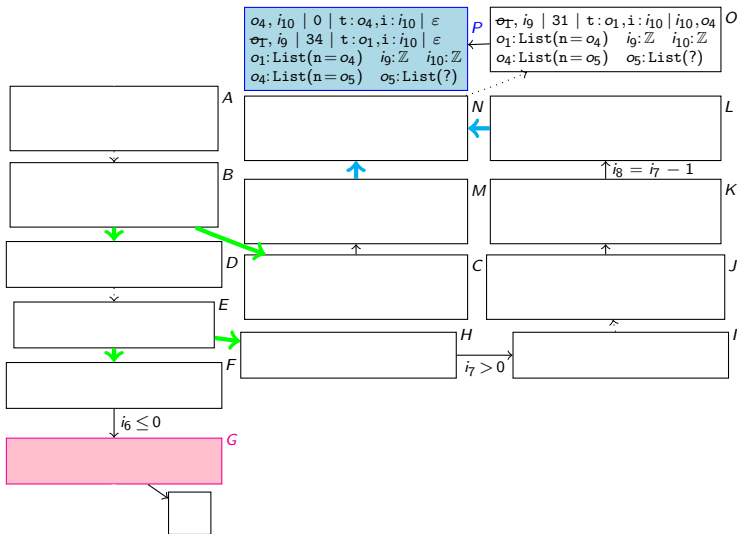
State O :

- “aload_0” and “getfield” load value o_4 of this.n on opstack
- “iload_1” loads value i_{10} of i on opstack

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



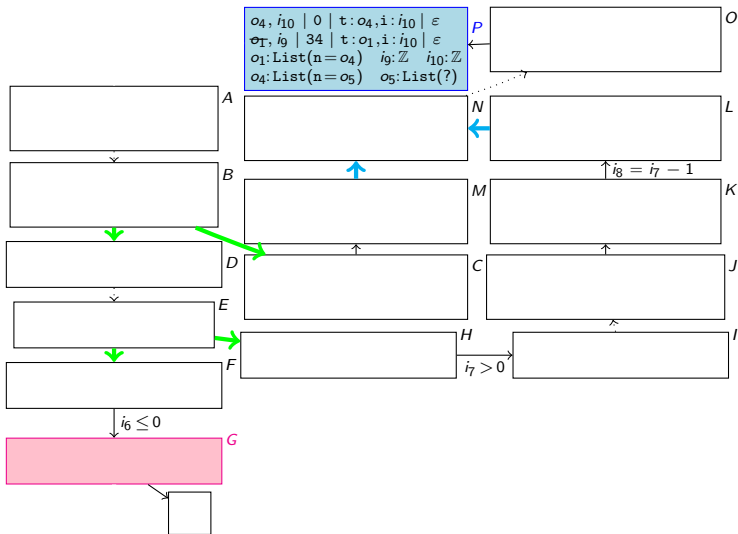
State P :

- recursive call of `appE` on arguments o_4, i_{10}
- *call state* P
- new stack frame on top of call stack, at position 0 of `appE`

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```

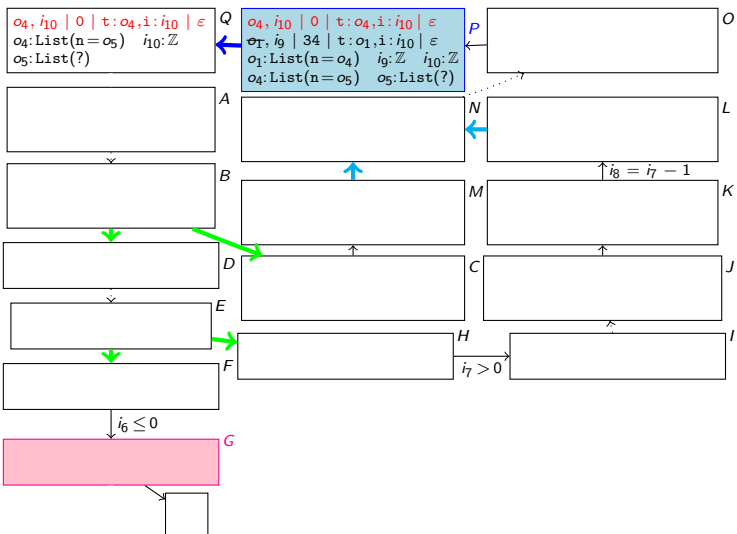


- repeated symbolic evaluation \Rightarrow unbounded growth of call stack \Rightarrow infinite termination graph

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



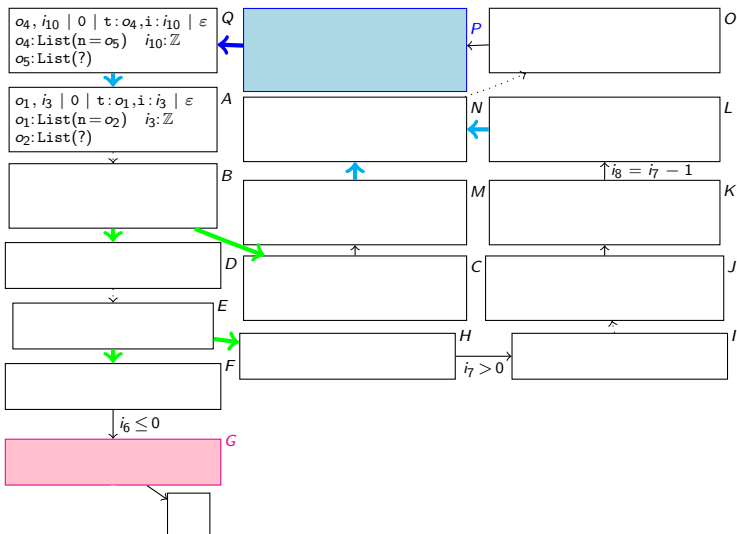
State Q:

- repeated symbolic evaluation \Rightarrow unbounded growth of call stack \Rightarrow infinite termination graph
- solution: *split* call stack, *call edge* to Q with P's top frame

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



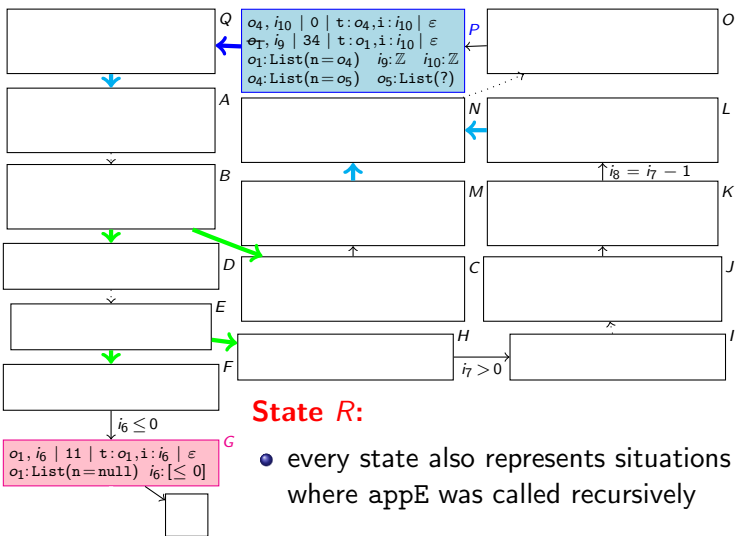
State Q:

- repeated symbolic evaluation \Rightarrow unbounded growth of call stack \Rightarrow infinite termination graph
- solution: *split* call stack, *call edge* to Q with P's top frame
- Q is *instance* of A (*instance edge*)

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

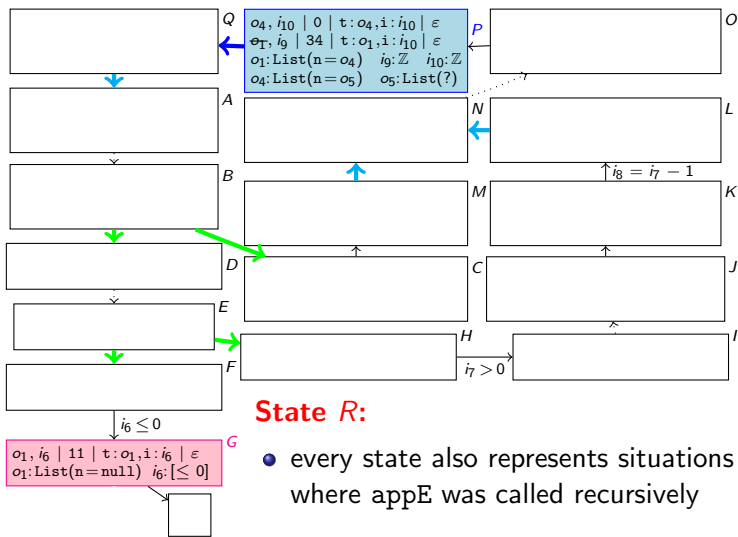
```




```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



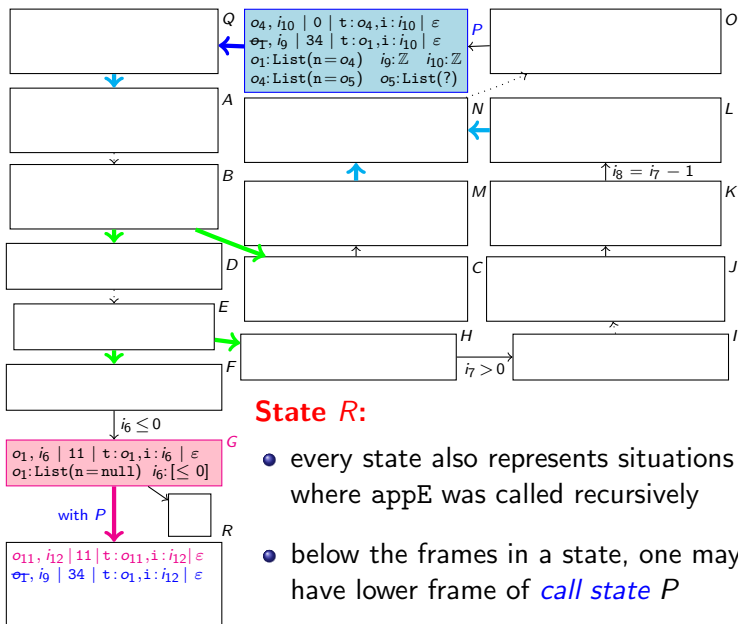
State R:

- every state also represents situations where `appE` was called recursively
- below the frames in a state, one may have lower frame of *call state* P

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



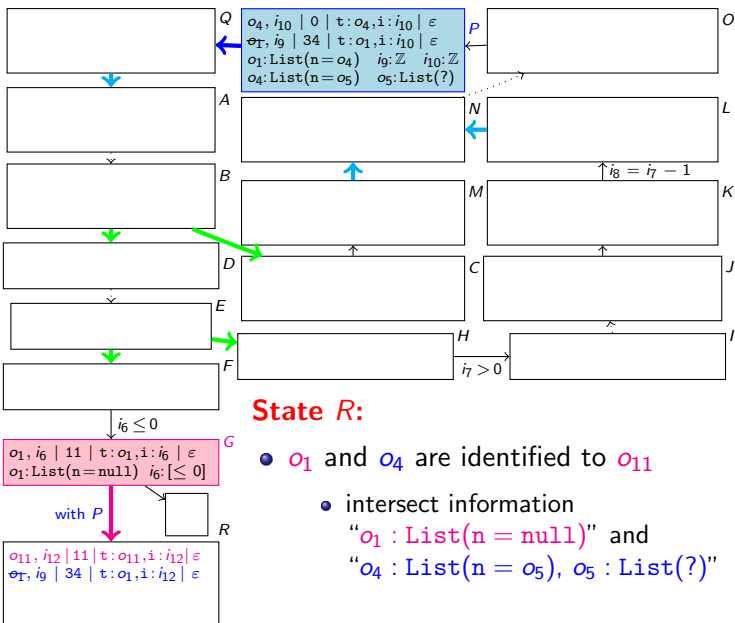
State R:

- every state also represents situations where `appE` was called recursively
- below the frames in a state, one may have lower frame of *call state P*
- return state G* gets additional successor *R* (*context edge*)

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

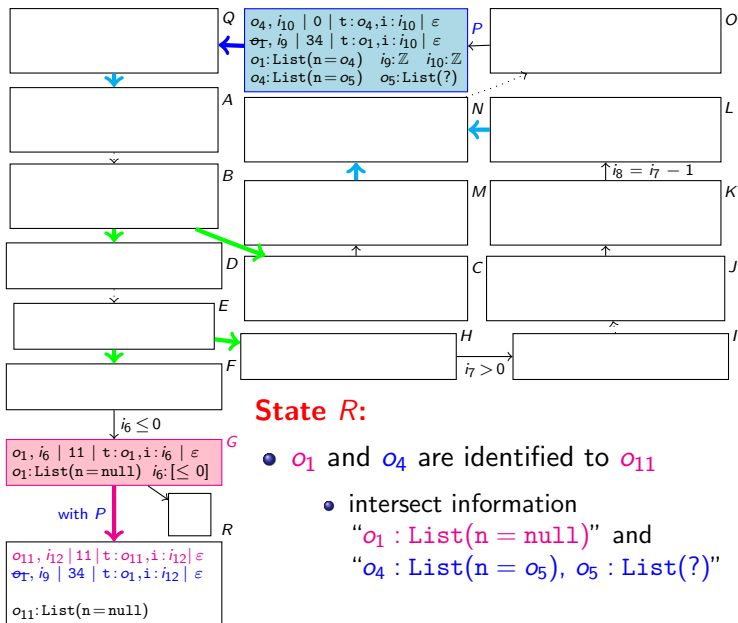
```



```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



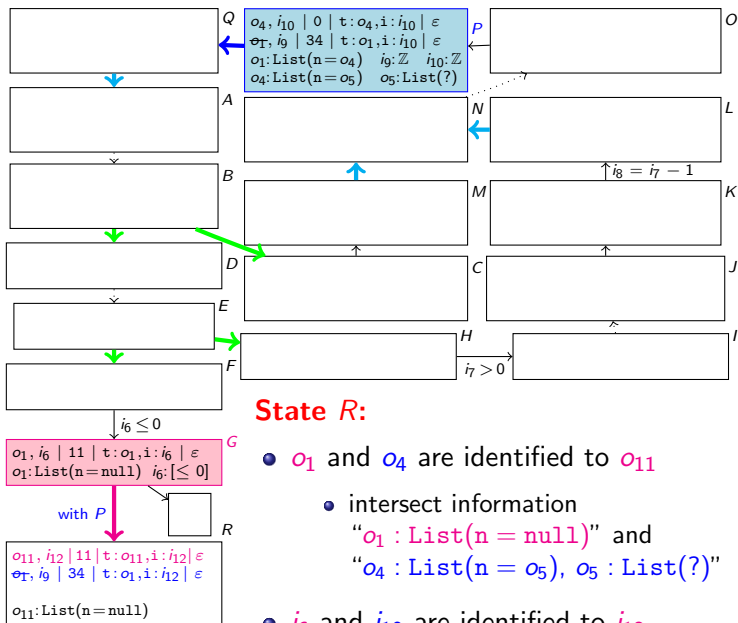
State R:

- o_1 and o_4 are identified to o_{11}
- intersect information
“ $o_1 : \text{List}(n = \text{null})$ ” and
“ $o_4 : \text{List}(n = o_5), o_5 : \text{List}(?)$ ”

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



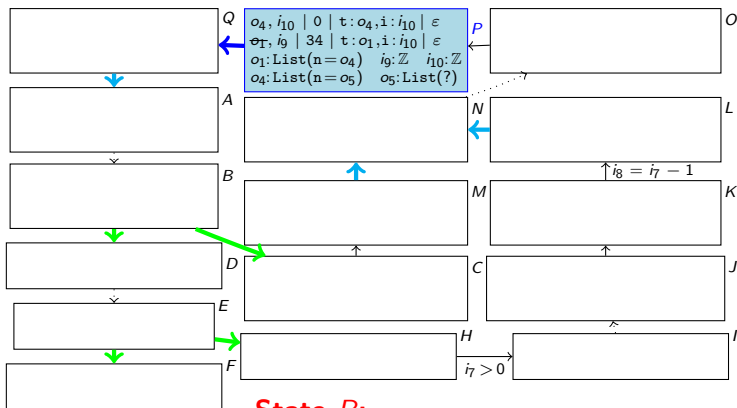
State R:

- o_1 and o_4 are identified to o_{11}
 - intersect information " $o_1: \text{List}(n = \text{null})$ " and " $o_4: \text{List}(n = o_5), o_5: \text{List}(?)$ "
- i_6 and i_{10} are identified to i_{12}
 - intersect information " $i_6: [\leq 0]$ " and " $i_{10}: \mathbb{Z}$ "

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



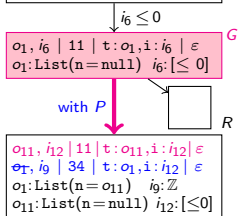
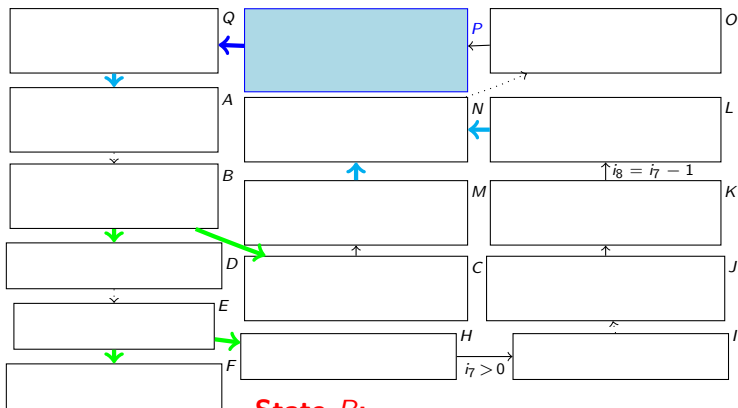
State R:

- o_1 and o_4 are identified to o_{11}
 - intersect information
“ $o_1: \text{List}(n = \text{null})$ ” and
“ $o_4: \text{List}(n = o_5), o_5: \text{List}(?)$ ”
- i_6 and i_{10} are identified to i_{12}
 - intersect information
“ $i_6: [\leq 0]$ ” and “ $i_{10}: \mathbb{Z}$ ”

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



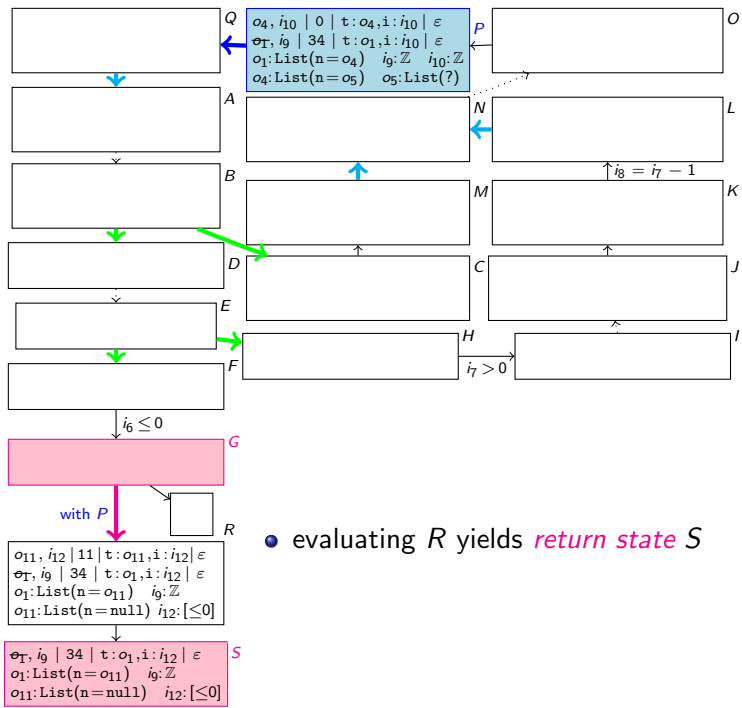
State R :

- σ_1 and σ_4 are identified to σ_{11}
- intersect information
“ $\sigma_1: \text{List}(n = \text{null})$ ” and
“ $\sigma_4: \text{List}(n = \sigma_5), \sigma_5: \text{List}(?)$ ”
- i_6 and i_{10} are identified to i_{12}
- intersect information
“ $i_6: [\leq 0]$ ” and “ $i_{10}: \mathbb{Z}$ ”

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```

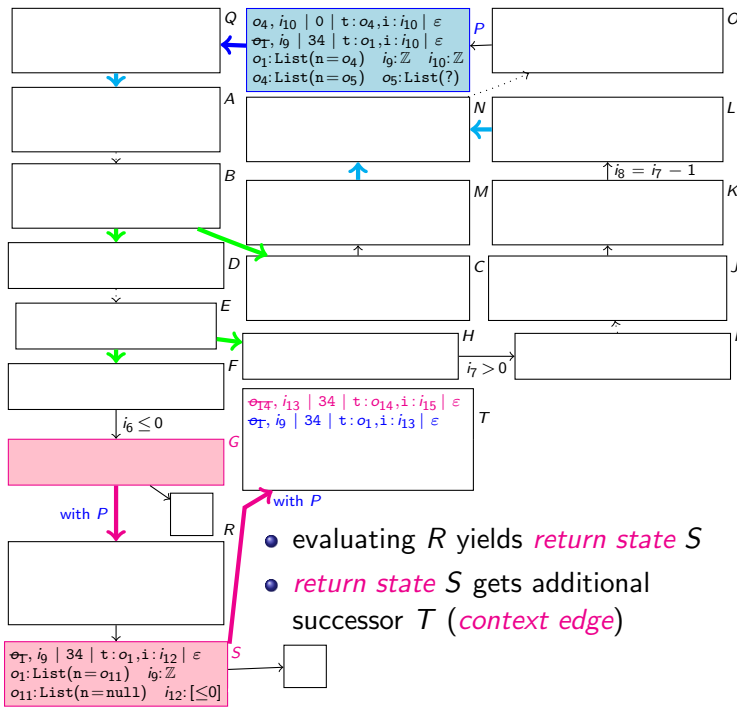


• evaluating R yields *return state S*


```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```

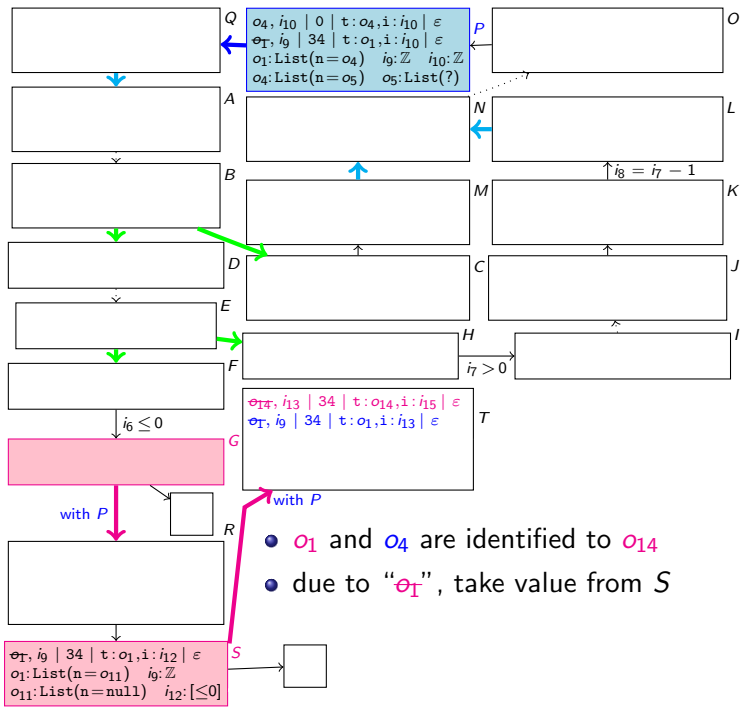


- evaluating R yields *return state* S
- *return state* S gets additional successor T (*context edge*)

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```

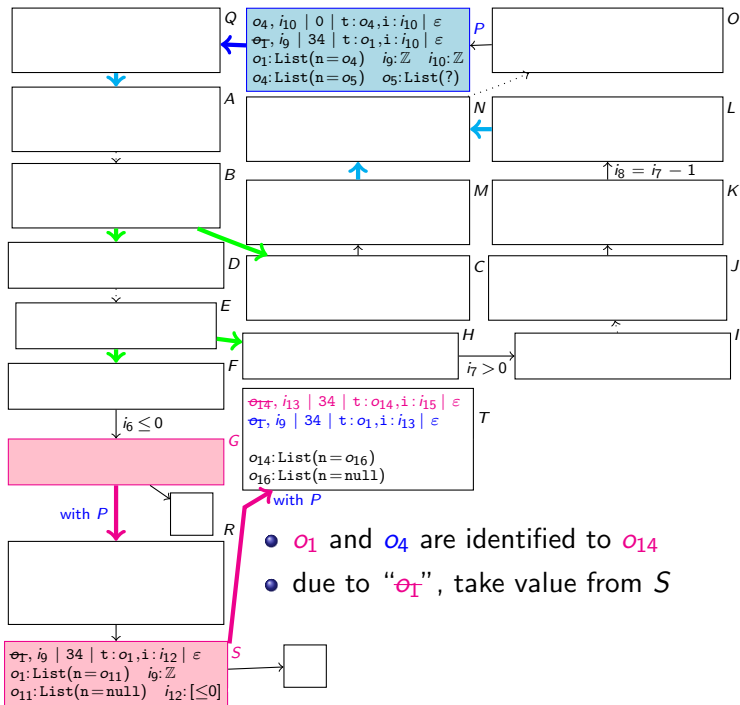


- o_1 and o_4 are identified to o_{14}
- due to " θ_T ", take value from S

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```

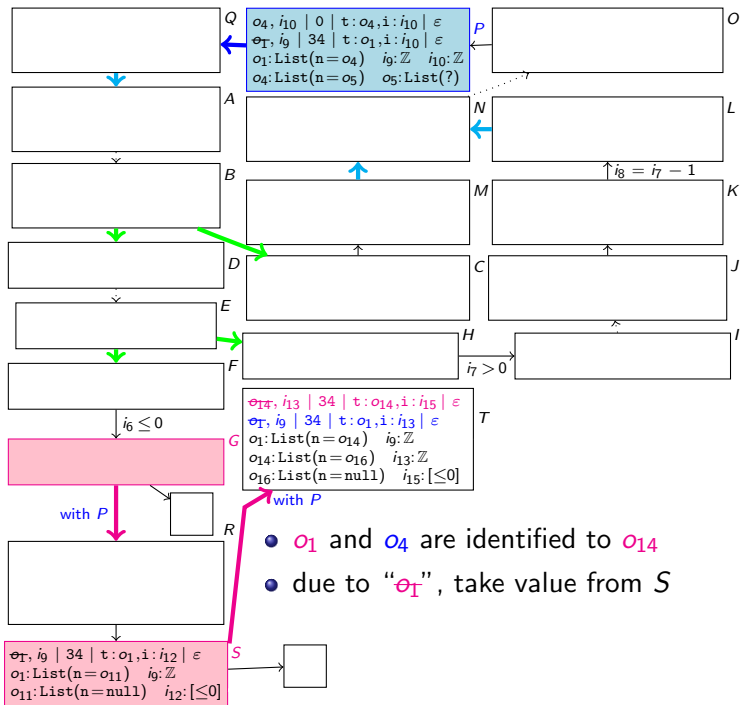


- σ_1 and σ_4 are identified to σ_{14}
- due to " σ_T ", take value from S

```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```

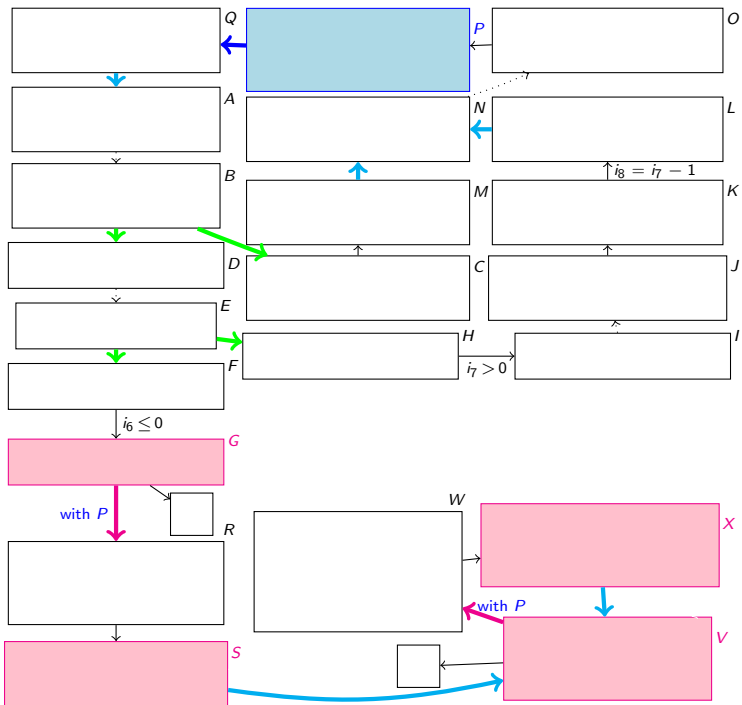


- σ_1 and σ_4 are identified to σ_{14}
- due to " θ_T ", take value from S

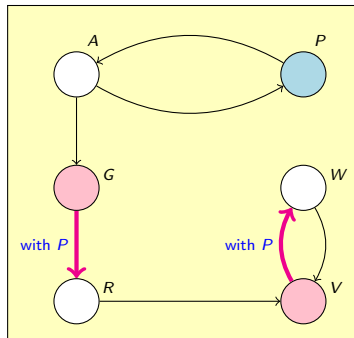
```

00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return

```



```
public void appE(int i) {  
    if (n == null) {  
        if (i <= 0) return;  
        n = new List();  
        i--;  
    }  
    n.appE(i);  
}
```



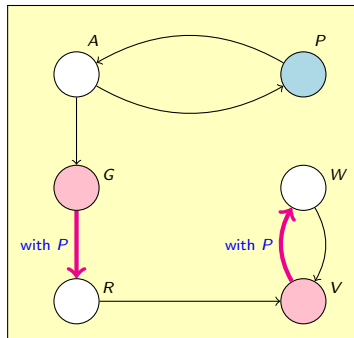
```

public void appE(int i) {
    if (n == null) {
        if (i <= 0) return;
        n = new List();
        i--;
    }
    n.appE(i);
} }

```

Termination Graphs

- expand nodes until all leaves correspond to program ends



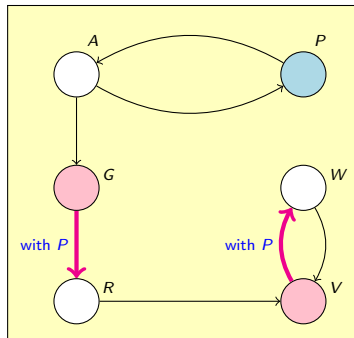
```

public void appE(int i) {
    if (n == null) {
        if (i <= 0) return;
        n = new List();
        i--;
    }
    n.appE(i);
} }

```

Termination Graphs

- expand nodes until all leaves correspond to program ends
- by appropriate generalization steps, one always reaches a *finite* termination graph

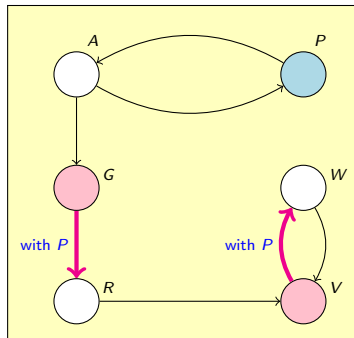



```

public void appE(int i) {
    if (n == null) {
        if (i <= 0) return;
        n = new List();
        i--;
    }
    n.appE(i);
} }

```

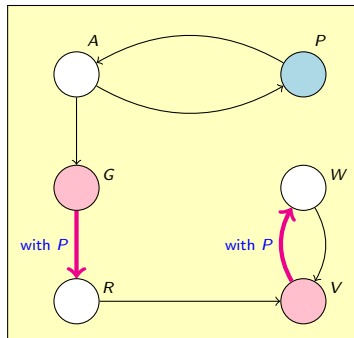
Termination Graphs



- expand nodes until all leaves correspond to program ends
- by appropriate generalization steps, one always reaches a *finite* termination graph
- termination graphs for a method can be re-used whenever the method is called

```
public void appE(int i) {  
    if (n == null) {  
        if (i <= 0) return;  
        n = new List();  
        i--;  
    }  
    n.appE(i);  
}
```

```
static void cappE(int j) {  
    List a = new List();  
    if (j > 0) {  
        a.appE(j);  
        while (a.n == null) {}  
    }  
}
```



Method `cappE`

- creates new list `a`
- calls `appE` to append $j > 0$ elements to `a`
- enters non-terminating loop if `a.n` is `null`

```

public void appE(int i) {
    if (n == null) {
        if (i <= 0) return;
        n = new List();
        i--;
    }
    n.appE(i);
} }

```

$i_1 > 0$

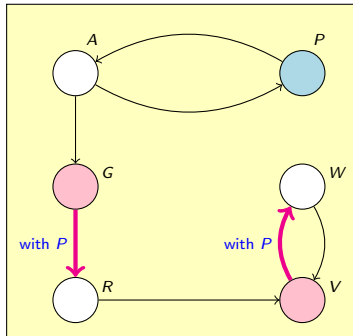
$i_1 \mid 14 \mid j: i_1, a: o_2 \mid i_1, o_2$
 $o_2: \text{List}(n=\text{null}) \quad i_1: [> 0]$

 A'

```

static void cappE(int j) {
    List a = new List();
    if (j > 0) {
        a.appE(j);
        while (a.n == null) {}
    } }

```



Method cappE

- creates new list a
- calls appE to append $j > 0$ elements to a
- enters non-terminating loop if a.n is null

```

public void appE(int i) {
  if (n == null) {
    if (i <= 0) return;
    n = new List();
    i--;
  }
  n.appE(i);
}

```

$i_1 > 0$

$i_1 \mid 14 \mid j: i_1, a: o_2 \mid i_1, o_2$
 $o_2: \text{List}(n=\text{null}) \quad i_1: [> 0]$

A'

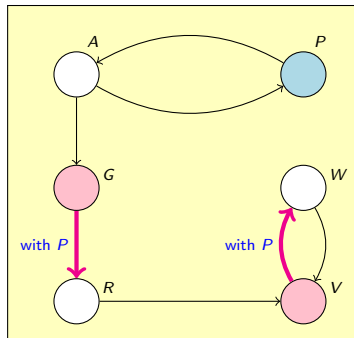
$o_2, i_1 \mid 0 \mid t: o_2, i: i_1 \mid \varepsilon$
 $i_1 \mid 17 \mid j: i_1, a: o_2 \mid \varepsilon$
 $o_2: \text{List}(n=\text{null}) \quad i_1: [> 0]$

B'

```

static void cappE(int j) {
  List a = new List();
  if (j > 0) {
    a.appE(j);
    while (a.n == null) {}
  }
}

```



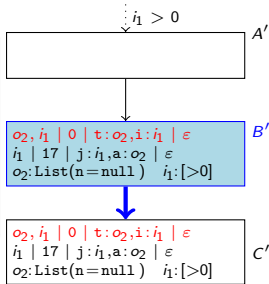
State B' :

- call of appE on arguments o_2, i_1
- new *call state* B'
- new stack frame on top of call stack, at position 0 of appE

```

public void appE(int i) {
  if (n == null) {
    if (i <= 0) return;
    n = new List();
    i--;
  }
  n.appE(i);
}

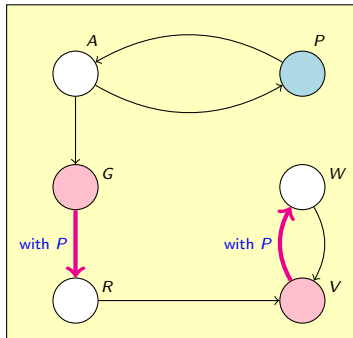
```



```

static void cappE(int j) {
  List a = new List();
  if (j > 0) {
    a.appE(j);
    while (a.n == null) {}
  }
}

```



State C' :

- *split* call stack, *call edge* to C' with top frame of B'

```

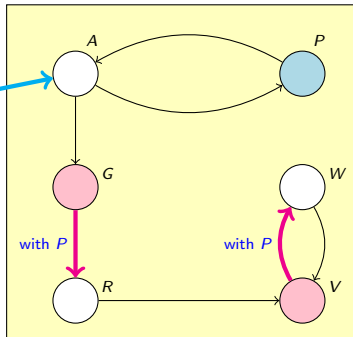
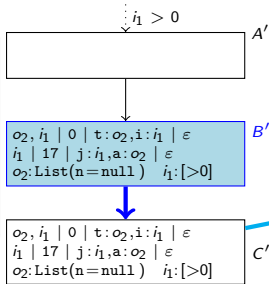
public void appE(int i) {
  if (n == null) {
    if (i <= 0) return;
    n = new List();
    i--;
  }
  n.appE(i);
}

```

```

static void cappE(int j) {
  List a = new List();
  if (j > 0) {
    a.appE(j);
    while (a.n == null) {}
  }
}

```



State C' :

- *split* call stack, *call edge* to C' with top frame of B'
- C' is *instance* of A (initial state of `appE`'s termination graph)

```

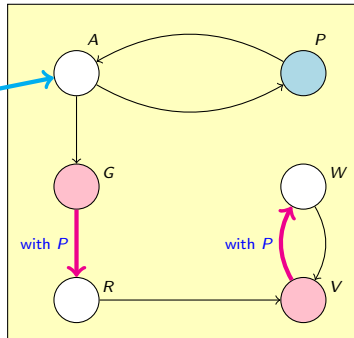
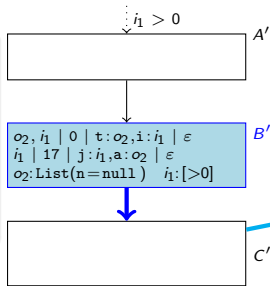
public void appE(int i) {
  if (n == null) {
    if (i <= 0) return;
    n = new List();
    i--;
  }
  n.appE(i);
}

```

```

static void cappE(int j) {
  List a = new List();
  if (j > 0) {
    a.appE(j);
    while (a.n == null) {}
  }
}

```



Every **return state** has **context edge** with every **call state** of appE

```

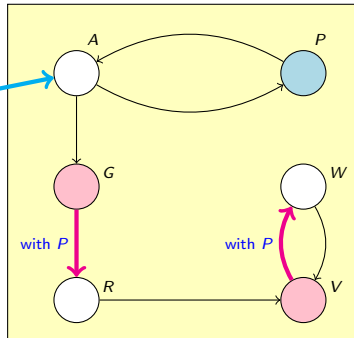
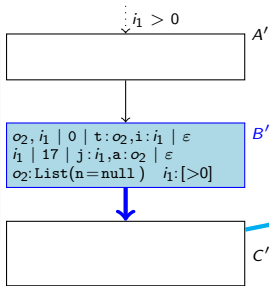
public void appE(int i) {
  if (n == null) {
    if (i <= 0) return;
    n = new List();
    i--;
  }
  n.appE(i);
}

```

```

static void cappE(int j) {
  List a = new List();
  if (j > 0) {
    a.appE(j);
    while (a.n == null) {}
  }
}

```



Every **return state** has **context edge** with every **call state** of appE

- **G** with **P** yields **R**
- **V** with **P** yields **W**


```

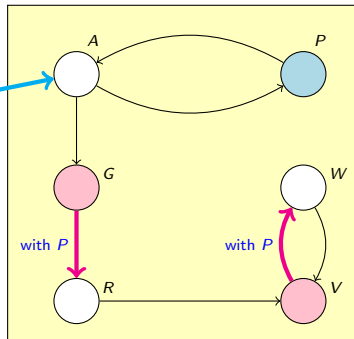
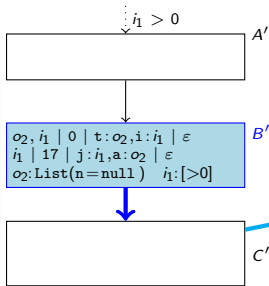
public void appE(int i) {
  if (n == null) {
    if (i <= 0) return;
    n = new List();
    i--;
  }
  n.appE(i);
}

```

```

static void cappE(int j) {
  List a = new List();
  if (j > 0) {
    a.appE(j);
    while (a.n == null) {}
  }
}

```



Every **return state** has **context edge** with every **call state** of appE

- **G** with **P** yields **R**
- **V** with **P** yields **W**
- **G** with **B'** not possible (intersection empty: $i \leq 0$ in **G**, $i > 0$ in **B'**)

```

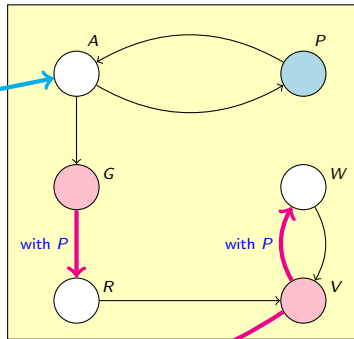
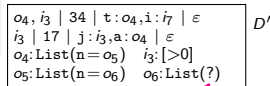
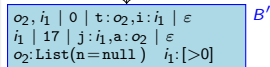
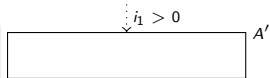
public void appE(int i) {
  if (n == null) {
    if (i <= 0) return;
    n = new List();
    i--;
  }
  n.appE(i);
}

```

```

static void cappE(int j) {
  List a = new List();
  if (j > 0) {
    a.appE(j);
    while (a.n == null) {}
  }
}

```



Every **return state** has **context edge** with every **call state** of appE

- **G** with **P** yields **R**
- **V** with **P** yields **W**
- **G** with **B'** not possible (intersection empty: $i \leq 0$ in **G**, $i > 0$ in **B'**)
- **V** with **B'** yields **D'**

```

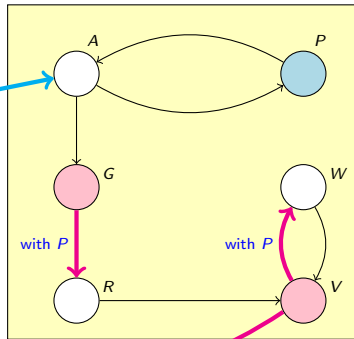
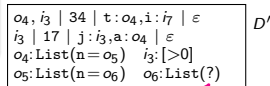
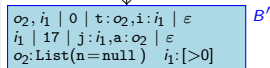
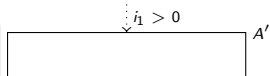
public void appE(int i) {
  if (n == null) {
    if (i <= 0) return;
    n = new List();
    i--;
  }
  n.appE(i);
}

```

```

static void cappE(int j) {
  List a = new List();
  if (j > 0) {
    a.appE(j);
    while (a.n == null) {}
  }
}

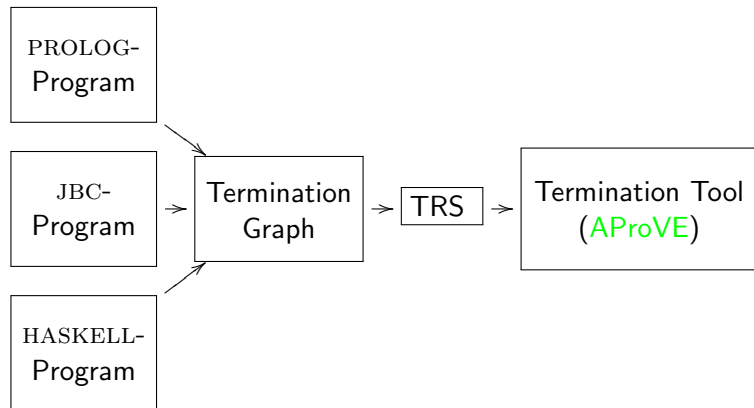
```



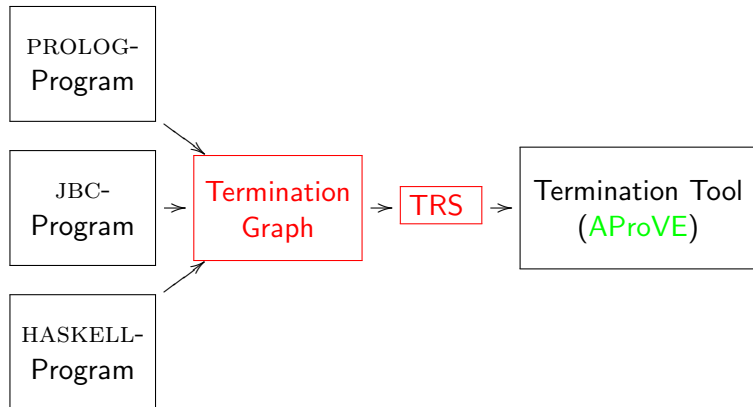
Every return state has context edge with every call state of `appE`

- G with P yields R
- V with P yields W
- G with B' not possible (intersection empty: $i \leq 0$ in G , $i > 0$ in B')
- V with B' yields D' ($a.n$ not null \Rightarrow while-loop not executed)

From Termination Graphs to TRSs



From Termination Graphs to TRSs



Transforming States to Terms

$$\left[\begin{array}{l} o_4, i_{10} \mid 0 \mid t:o_4, i:i_{10} \mid \varepsilon \\ o_1, i_9 \mid 34 \mid t:o_1, i:i_{10} \mid \varepsilon \\ o_1:\text{List}(n=o_4) \quad i_9:\mathbb{Z} \quad i_{10}:\mathbb{Z} \\ o_4:\text{List}(n=o_5) \quad o_5:\text{List}(?) \end{array} \right] P$$

- For every class C with n fields, introduce function symbol C with n arguments

$$\underbrace{L(o_5)}_{o_4}$$

Transforming States to Terms

$$\left[\begin{array}{l} o_4, i_{10} \mid 0 \mid t: o_4, i: i_{10} \mid \varepsilon \\ \sigma_1, i_9 \mid 34 \mid t: \sigma_1, i: i_{10} \mid \varepsilon \\ o_1: \text{List}(n = o_4) \quad i_9: \mathbb{Z} \quad i_{10}: \mathbb{Z} \\ o_4: \text{List}(n = o_5) \quad o_5: \text{List}(?) \end{array} \right] P$$

- For every class C with n fields, introduce function symbol C with n arguments

$$\underbrace{L(o_5)}_{o_4}, i_{10}$$

Transforming States to Terms

$$\left[\begin{array}{l} o_4, i_{10} \mid 0 \mid \mathbf{t}:o_4, \mathbf{i}:i_{10} \mid \varepsilon \\ \sigma_{\Gamma}, i_9 \mid 34 \mid \mathbf{t}:o_1, \mathbf{i}:i_{10} \mid \varepsilon \\ o_1:\text{List}(n=o_4) \quad i_9:\mathbb{Z} \quad i_{10}:\mathbb{Z} \\ \mathbf{o_4:\text{List}(n=o_5)} \quad \mathbf{o_5:\text{List}(?) } \end{array} \right]^P$$

- For every class C with n fields, introduce function symbol C with n arguments

$$\underbrace{L(o_5)}_{o_4}, i_{10}, \underbrace{L(o_5)}_{o_4}$$

Transforming States to Terms

$$\left[\begin{array}{l} o_4, i_{10} \mid 0 \mid t:o_4, i:i_{10} \mid \varepsilon \\ \sigma_T, i_9 \mid 34 \mid t:o_1, i:i_{10} \mid \varepsilon \\ o_1:\text{List}(n=o_4) \quad i_9:\mathbb{Z} \quad i_{10}:\mathbb{Z} \\ o_4:\text{List}(n=o_5) \quad o_5:\text{List}(?) \end{array} \right]^P$$

- For every class C with n fields, introduce function symbol C with n arguments

$$\underbrace{L(o_5)}_{o_4}, i_{10}, \underbrace{L(o_5)}_{o_4}, i_{10}$$

Transforming States to Terms

$$\begin{array}{l} o_4, i_{10} \mid 0 \mid t: o_4, i: i_{10} \mid \varepsilon \\ \sigma_T, i_9 \mid 34 \mid t: o_1, i: i_{10} \mid \varepsilon \\ o_1: \text{List}(n = o_4) \quad i_9: \mathbb{Z} \quad i_{10}: \mathbb{Z} \\ o_4: \text{List}(n = o_5) \quad o_5: \text{List}(?) \end{array} \quad P$$

- For every class C with n fields, introduce function symbol C with n arguments
- Extension for *class hierarchies* (nested constructor symbols)

$$\underbrace{L(o_5)}_{o_4}, i_{10}, \underbrace{L(o_5)}_{o_4}, i_{10}$$

Transforming States to Terms

$$\left. \begin{array}{l} o_4, i_{10} \mid 0 \mid t: o_4, i: i_{10} \mid \varepsilon \\ \sigma_1, i_9 \mid 34 \mid t: \sigma_1, i: i_{10} \mid \varepsilon \\ o_1: \text{List}(n = o_4) \quad i_9: \mathbb{Z} \quad i_{10}: \mathbb{Z} \\ o_4: \text{List}(n = o_5) \quad o_5: \text{List}(?) \end{array} \right\} P$$

- For every stack frame of state s at position pp , introduce function symbol $f_{s,pp}$.

$$f_{P,0}(\underbrace{L(o_5)}_{o_4}, i_{10}, \underbrace{L(o_5)}_{o_4}, i_{10})$$

Transforming States to Terms

$$\left[\begin{array}{l} o_4, i_{10} \mid 0 \mid t: o_4, i: i_{10} \mid \varepsilon \\ \sigma_1, i_9 \mid 34 \mid t: \sigma_1, i: i_{10} \mid \varepsilon \\ o_1: \text{List}(n = o_4) \quad i_9: \mathbb{Z} \quad i_{10}: \mathbb{Z} \\ o_4: \text{List}(n = o_5) \quad o_5: \text{List}(?) \end{array} \right] P$$

- For every stack frame of state s at position pp , introduce function symbol $f_{s,pp}$.
- Call stack: first argument encodes frame *above* the current one (nested f-symbols)

$$f_{P,0}(\text{eos}, \underbrace{L(o_5)}_{o_4}, i_{10}, \underbrace{L(o_5)}_{o_4}, i_{10})$$

Transforming States to Terms

$$\boxed{\begin{array}{l} o_4, i_{10} \mid 0 \mid \mathbf{t}: o_4, \mathbf{i}: i_{10} \mid \varepsilon \\ \mathbf{o}_1, i_9 \mid 34 \mid \mathbf{t}: o_1, \mathbf{i}: i_{10} \mid \varepsilon \\ o_1: \text{List}(n = o_4) \quad i_9: \mathbb{Z} \quad i_{10}: \mathbb{Z} \\ o_4: \text{List}(n = o_5) \quad o_5: \text{List}(?) \end{array}} \quad P$$

- For every stack frame of state s at position pp , introduce function symbol $f_{s,pp}$.
- Call stack: first argument encodes frame *above* the current one (nested f-symbols)

$$f_{P,34}(f_{P,0}(\text{eos}, \underbrace{L(o_5)}_{o_4}, i_{10}, \underbrace{L(o_5)}_{o_4}, i_{10}))$$

Transforming States to Terms

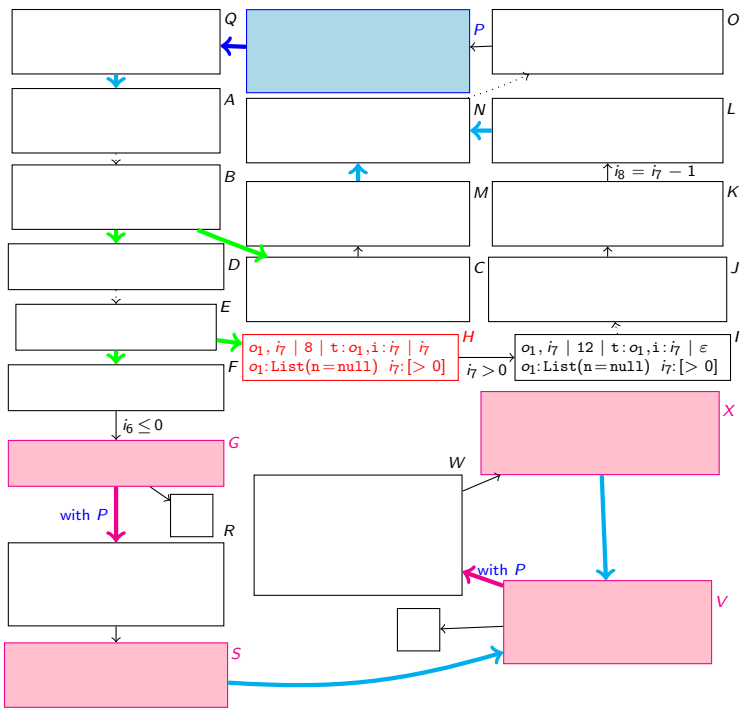
$$\left[\begin{array}{l} o_4, i_{10} \mid 0 \mid t:o_4, i:i_{10} \mid \varepsilon \\ \sigma_1, i_9 \mid 34 \mid t:\sigma_1, i:i_{10} \mid \varepsilon \\ o_1:\text{List}(n=o_4) \quad i_9:\mathbb{Z} \quad i_{10}:\mathbb{Z} \\ o_4:\text{List}(n=o_5) \quad o_5:\text{List}(?) \end{array} \right] P$$

- For every stack frame of state s at position pp , introduce function symbol $f_{s,pp}$.
- Call stack: first argument encodes frame *above* the current one (nested f-symbols)

$$f_{P,34}(f_{P,0}(\text{eos}, \underbrace{L(o_5)}_{o_4}, i_{10}, \underbrace{L(o_5)}_{o_4}, i_{10}), \underbrace{L(L(o_5))}_{o_1}, i_9, \underbrace{L(L(o_5))}_{o_1}, i_{10})$$

Transforming Evaluation Edges

$f_{H,8}(\text{eos},$
 $L(\text{null}),$
 $i_7,$
 $L(\text{null}),$
 $i_7,$
 $i_7)$



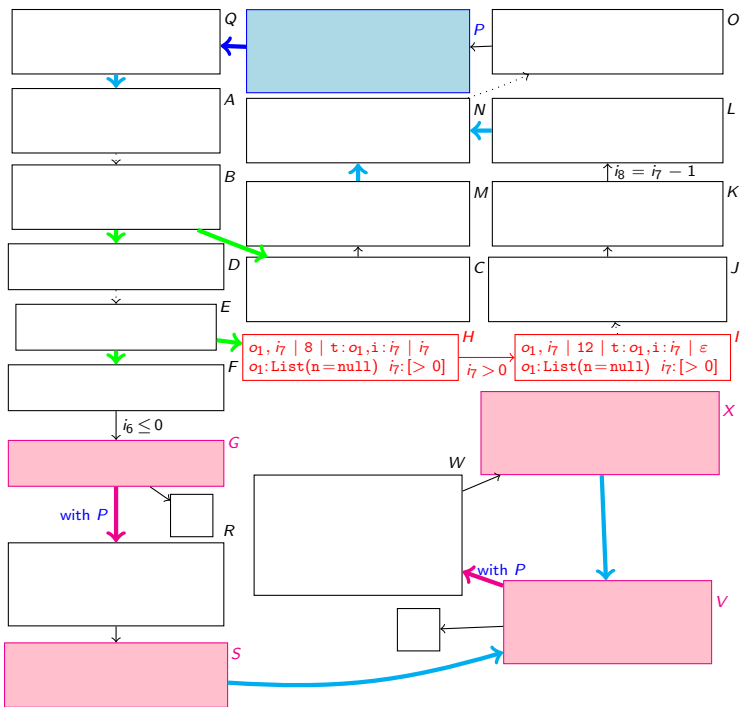
Transforming Evaluation Edges

$f_{H,8}(\text{eos},$
 $L(\text{null}),$
 $i_7,$
 $L(\text{null}),$
 $i_7,$
 $i_7)$

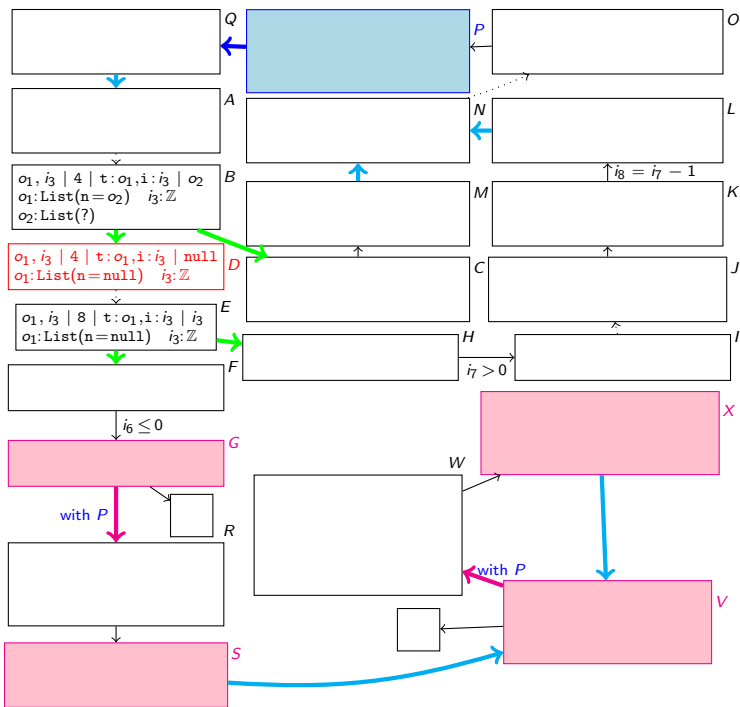


$f_{I,12}(\text{eos},$
 $L(\text{null}),$
 $i_7,$
 $L(\text{null}),$
 $i_7)$

$| i_7 > 0$



Transforming Refinement Edges

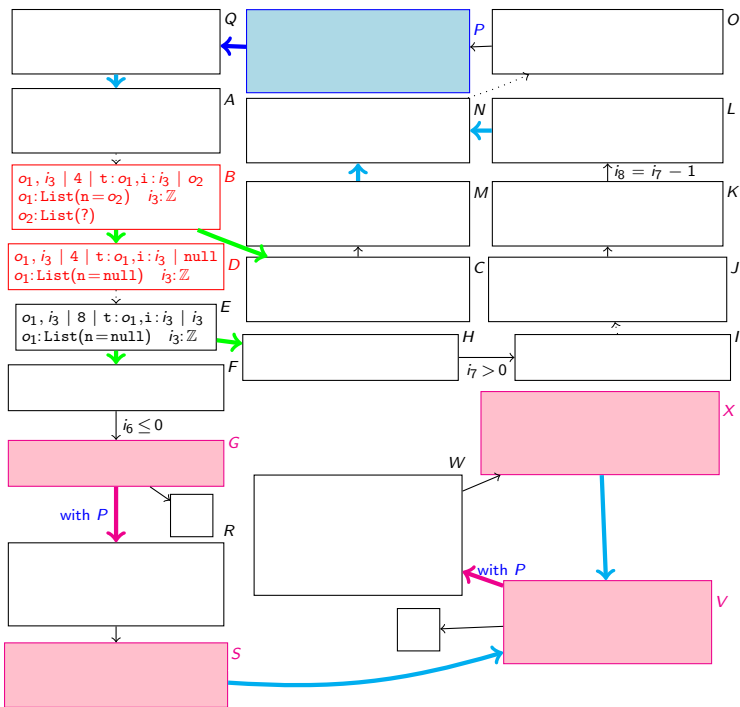


$f_{D,4}(\text{eos},$
 $L(\text{null}),$
 $i_3,$
 $L(\text{null}),$
 $i_3,$
 $\text{null})$

Transforming Refinement Edges

$f_{B,4}(\text{eos},$
 $L(\text{null}),$
 $i_3,$
 $L(\text{null}),$
 $i_3,$
 $\text{null})$

$f_{D,4}(\text{eos},$
 $L(\text{null}),$
 $i_3,$
 $L(\text{null}),$
 $i_3,$
 $\text{null})$

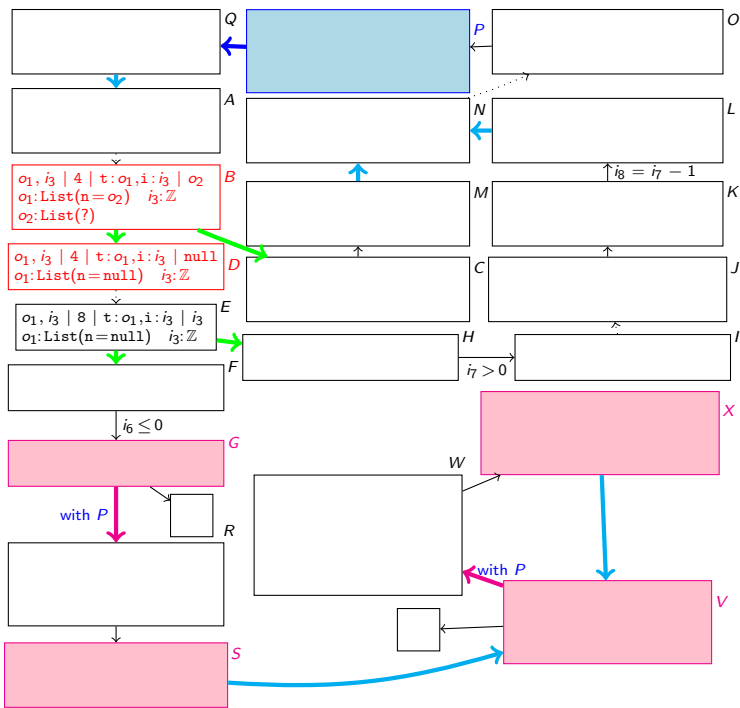


Transforming Refinement Edges

$f_{B,4}(\text{eos},$
 $L(\text{null}),$
 $i_3,$
 $L(\text{null}),$
 $i_3,$
 $\text{null})$

→

$f_{D,4}(\text{eos},$
 $L(\text{null}),$
 $i_3,$
 $L(\text{null}),$
 $i_3,$
 $\text{null})$



Merging Rewrite Rules

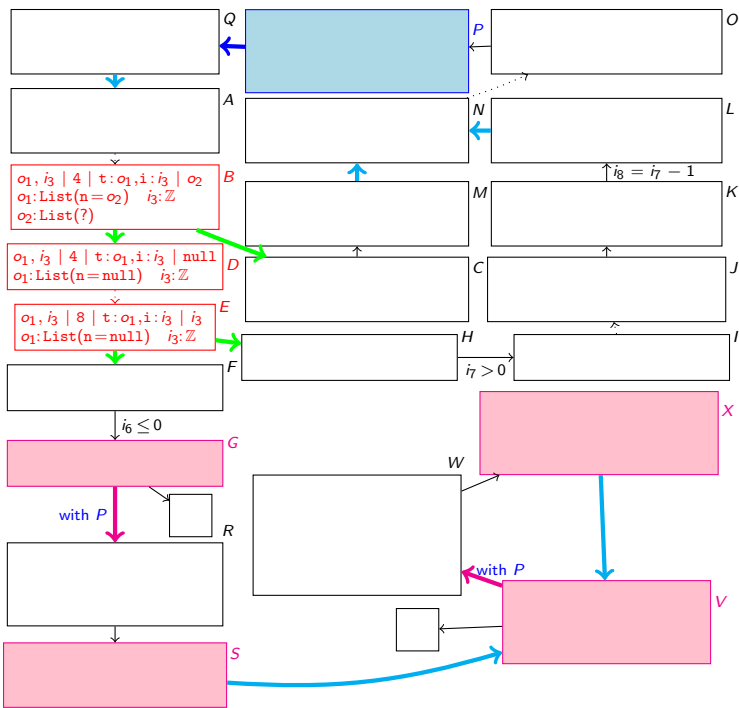
$f_{B,4}$ (eos,
 L(null),
 i_3 ,
 L(null),
 i_3 ,
 null)

→

$f_{D,4}$ (eos,
 L(null),
 i_3 ,
 L(null),
 i_3 ,
 null)

→

$f_{E,8}$ (eos,
 L(null),
 i_3 ,
 L(null),
 i_3 ,
 i_3)



$$f_A(L(\text{null}), i_6) \rightarrow f_G(L(\text{null}), i_6) \quad | i_6 \leq 0 \quad (1)$$

$$f_A(L(\text{null}), i_7) \rightarrow f_P(f_A(L(\text{null}), i_7 - 1), L(L(\text{null})), i_7) \quad | i_7 > 0 \quad (2)$$

$$f_A(L(L(o_5)), i_3) \rightarrow f_P(f_A(L(o_5), i_3), L(L(o_5)), i_3) \quad (3)$$

$$f_P(f_G(L(\text{null}), i_{12}), L(L(\text{null})), i_9) \rightarrow f_V(L(L(\text{null})), i_9) \quad (4)$$

$$f_P(f_V(L(L(o_{20})), i_{19}), L(L(o_5)), i_9) \rightarrow f_V(L(L(L(o_{20}))), i_9) \quad (5)$$

TRS is natural

```
public void appE(int i) {  
    if (n == null) {  
        if (i <= 0) return;  
        n = new List();  
        i--;  
    }  
    n.appE(i);  
}
```

$$f_A(L(\text{null}), i_6) \rightarrow f_G(L(\text{null}), i_6) \quad | i_6 \leq 0 \quad (1)$$

$$f_A(L(\text{null}), i_7) \rightarrow f_P(f_A(L(\text{null}), i_7 - 1), L(L(\text{null})), i_7) \quad | i_7 > 0 \quad (2)$$

$$f_A(L(L(o_5)), i_3) \rightarrow f_P(f_A(L(o_5), i_3), L(L(o_5)), i_3) \quad (3)$$

$$f_P(f_G(L(\text{null}), i_{12}), L(L(\text{null})), i_9) \rightarrow f_V(L(L(\text{null})), i_9) \quad (4)$$

$$f_P(f_V(L(L(o_{20})), i_{19}), L(L(o_5)), i_9) \rightarrow f_V(L(L(L(o_{20}))), i_9) \quad (5)$$

TRS is natural

- 1 If $n == \text{null}$ and $i \leq 0$, then return.

```
public void appE(int i) {  
    if (n == null) {  
        if (i <= 0) return;  
        n = new List();  
        i--;  
    }  
    n.appE(i);  
}
```


$$f_A(L(\text{null}), i_6) \rightarrow f_G(L(\text{null}), i_6) \quad | i_6 \leq 0 \quad (1)$$

$$f_A(L(\text{null}), i_7) \rightarrow f_P(f_A(L(\text{null}), i_7 - 1), L(L(\text{null})), i_7) \quad | i_7 > 0 \quad (2)$$

$$f_A(L(L(o_5)), i_3) \rightarrow f_P(f_A(L(o_5), i_3), L(L(o_5)), i_3) \quad (3)$$

$$f_P(f_G(L(\text{null}), i_{12}), L(L(\text{null})), i_9) \rightarrow f_V(L(L(\text{null})), i_9) \quad (4)$$

$$f_P(f_V(L(L(o_{20})), i_{19}), L(L(o_5)), i_9) \rightarrow f_V(L(L(L(o_{20}))), i_9) \quad (5)$$

TRS is natural

- 1 If $n == \text{null}$ and $i \leq 0$, then return.
- 2 If $n == \text{null}$ and $i > 0$, then attach new element to list. Recursive call with tail of list and $i-1$.

```
public void appE(int i) {  
    if (n == null) {  
        if (i <= 0) return;  
        n = new List();  
        i--;  
    }  
    n.appE(i);  
}
```

$$f_A(L(\text{null}), i_6) \rightarrow f_G(L(\text{null}), i_6) \quad | i_6 \leq 0 \quad (1)$$

$$f_A(L(\text{null}), i_7) \rightarrow f_P(f_A(L(\text{null}), i_7 - 1), L(L(\text{null})), i_7) \quad | i_7 > 0 \quad (2)$$

$$f_A(L(L(o_5)), i_3) \rightarrow f_P(f_A(L(o_5), i_3), L(L(o_5)), i_3) \quad (3)$$

$$f_P(f_G(L(\text{null}), i_{12}), L(L(\text{null})), i_9) \rightarrow f_V(L(L(\text{null})), i_9) \quad (4)$$

$$f_P(f_V(L(L(o_{20})), i_{19}), L(L(o_5)), i_9) \rightarrow f_V(L(L(L(o_{20}))), i_9) \quad (5)$$

TRS is natural

- 1 If $n == \text{null}$ and $i \leq 0$, then return.
- 2 If $n == \text{null}$ and $i > 0$,
then attach new element to list.
Recursive call with tail of list and $i-1$.
- 3 If $n \neq \text{null}$,
then recursive call with tail of list and i .

```
public void appE(int i) {  
    if (n == null) {  
        if (i <= 0) return;  
        n = new List();  
        i--;  
    }  
    n.appE(i);  
}
```

$$f_A(L(\text{null}), i_6) \rightarrow f_G(L(\text{null}), i_6) \quad | i_6 \leq 0 \quad (1)$$

$$f_A(L(\text{null}), i_7) \rightarrow f_P(f_A(L(\text{null}), i_7 - 1), L(L(\text{null})), i_7) \quad | i_7 > 0 \quad (2)$$

$$f_A(L(L(o_5)), i_3) \rightarrow f_P(f_A(L(o_5), i_3), L(L(o_5)), i_3) \quad (3)$$

$$f_P(f_G(L(\text{null}), i_{12}), L(L(\text{null})), i_9) \rightarrow f_V(L(L(\text{null})), i_9) \quad (4)$$

$$f_P(f_V(L(L(o_{20})), i_{19}), L(L(o_5)), i_9) \rightarrow f_V(L(L(L(o_{20}))), i_9) \quad (5)$$

TRS is natural

- 1 If `n == null` and `i <= 0`, then return.
- 2 If `n == null` and `i > 0`,
then attach new element to list.
Recursive call with tail of list and `i-1`.
- 3 If `n != null`,
then recursive call with tail of list and `i`.
- 4 After recursive call, resulting list `L(null)` is written to field `n`.

```
public void appE(int i) {  
    if (n == null) {  
        if (i <= 0) return;  
        n = new List();  
        i--;  
    }  
    n.appE(i);  
}
```

$$f_A(L(\text{null}), i_6) \rightarrow f_G(L(\text{null}), i_6) \quad | i_6 \leq 0 \quad (1)$$

$$f_A(L(\text{null}), i_7) \rightarrow f_P(f_A(L(\text{null}), i_7 - 1), L(L(\text{null})), i_7) \quad | i_7 > 0 \quad (2)$$

$$f_A(L(L(o_5)), i_3) \rightarrow f_P(f_A(L(o_5), i_3), L(L(o_5)), i_3) \quad (3)$$

$$f_P(f_G(L(\text{null}), i_{12}), L(L(\text{null})), i_9) \rightarrow f_V(L(L(\text{null})), i_9) \quad (4)$$

$$f_P(f_V(L(L(o_{20})), i_{19}), L(L(o_5)), i_9) \rightarrow f_V(L(L(L(o_{20}))), i_9) \quad (5)$$

TRS is natural

- 1 If `n == null` and `i <= 0`, then return.
- 2 If `n == null` and `i > 0`,
then attach new element to list.
Recursive call with tail of list and `i-1`.
- 3 If `n != null`,
then recursive call with tail of list and `i`.
- 4 After recursive call, resulting list `L(null)` is written to field `n`.
- 5 After recursive call, resulting list `L(L(o20))` is written to field `n`.
Side effect replaces `L(L(o5))` by `L(L(L(o20)))`.

```
public void appE(int i) {  
    if (n == null) {  
        if (i <= 0) return;  
        n = new List();  
        i--;  
    }  
    n.appE(i);  
}
```

$$f_A(L(\text{null}), i_6) \rightarrow f_G(L(\text{null}), i_6) \quad | i_6 \leq 0 \quad (1)$$

$$f_A(L(\text{null}), i_7) \rightarrow f_P(f_A(L(\text{null}), i_7 - 1), L(L(\text{null})), i_7) \quad | i_7 > 0 \quad (2)$$

$$f_A(L(L(o_5)), i_3) \rightarrow f_P(f_A(L(o_5), i_3), L(L(o_5)), i_3) \quad (3)$$

$$f_P(f_G(L(\text{null}), i_{12}), L(L(\text{null})), i_9) \rightarrow f_V(L(L(\text{null})), i_9) \quad (4)$$

$$f_P(f_V(L(L(o_{20})), i_{19}), L(L(o_5)), i_9) \rightarrow f_V(L(L(L(o_{20}))), i_9) \quad (5)$$

TRS is natural and termination is easy

- 1 If $n == \text{null}$ and $i \leq 0$, then return.
- 2 If $n == \text{null}$ and $i > 0$, then attach new element to list. Recursive call with tail of list and $i-1$.
- 3 If $n \neq \text{null}$, then recursive call with tail of list and i .
- 4 After recursive call, resulting list $L(\text{null})$ is written to field n .
- 5 After recursive call, resulting list $L(L(o_{20}))$ is written to field n . Side effect replaces $L(L(o_5))$ by $L(L(L(o_{20})))$.

```
public void appE(int i) {
    if (n == null) {
        if (i <= 0) return;
        n = new List();
        i--;
    }
    n.appE(i);
}
```

$$f_A(L(\text{null}), i_6) \rightarrow f_G(L(\text{null}), i_6) \quad | i_6 \leq 0 \quad (1)$$

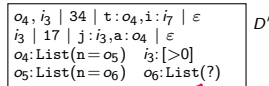
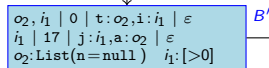
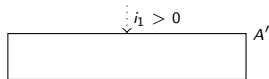
$$f_A(L(\text{null}), i_7) \rightarrow f_P(f_A(L(\text{null}), i_7 - 1), L(L(\text{null})), i_7) \quad | i_7 > 0 \quad (2)$$

$$f_A(L(L(o_5)), i_3) \rightarrow f_P(f_A(L(o_5), i_3), L(L(o_5)), i_3) \quad (3)$$

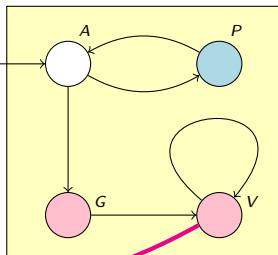
$$f_P(f_G(L(\text{null}), i_{12}), L(L(\text{null})), i_9) \rightarrow f_V(L(L(\text{null})), i_9) \quad (4)$$

$$f_P(f_V(L(L(o_{20})), i_{19}), L(L(o_5)), i_9) \rightarrow f_V(L(L(L(o_{20}))), i_9) \quad (5)$$

```
static void cappE(int j) {
    List a = new List();
    if (j > 0) {
        a.appE(j);
        while (a.n == null) {}
    }
}
```



with B'



$$f_A(L(\text{null}), i_6) \rightarrow f_G(L(\text{null}), i_6) \quad |i_6 \leq 0 \quad (1)$$

$$f_A(L(\text{null}), i_7) \rightarrow f_P(f_A(L(\text{null}), i_7 - 1), L(L(\text{null})), i_7) \quad |i_7 > 0 \quad (2)$$

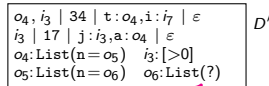
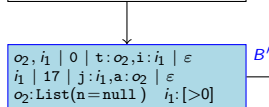
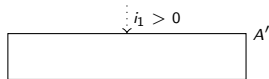
$$f_A(L(L(o_5)), i_3) \rightarrow f_P(f_A(L(o_5), i_3), L(L(o_5)), i_3) \quad (3)$$

$$f_P(f_G(L(\text{null}), i_{12}), L(L(\text{null})), i_9) \rightarrow f_V(L(L(\text{null})), i_9) \quad (4)$$

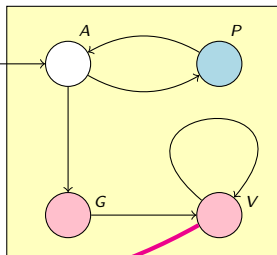
$$f_P(f_V(L(L(o_{20})), i_{19}), L(L(o_5)), i_9) \rightarrow f_V(L(L(L(o_{20}))), i_9) \quad (5)$$

$$f_{A'}(\dots) \rightarrow f_{B'}(f_A(\dots), \dots)$$

```
static void cappE(int j) {
    List a = new List();
    if (j > 0) {
        a.appE(j);
        while (a.n == null) {}
    }
}
```



with B'



$$f_A(L(\text{null}), i_6) \rightarrow f_G(L(\text{null}), i_6) \quad | i_6 \leq 0 \quad (1)$$

$$f_A(L(\text{null}), i_7) \rightarrow f_P(f_A(L(\text{null}), i_7 - 1), L(L(\text{null})), i_7) \quad | i_7 > 0 \quad (2)$$

$$f_A(L(L(o_5)), i_3) \rightarrow f_P(f_A(L(o_5), i_3), L(L(o_5)), i_3) \quad (3)$$

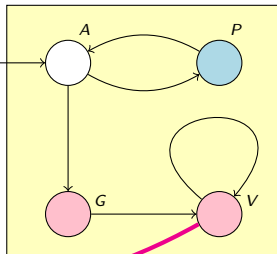
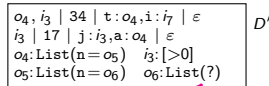
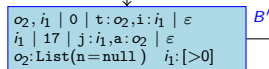
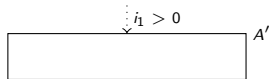
$$f_P(f_G(L(\text{null}), i_{12}), L(L(\text{null})), i_9) \rightarrow f_V(L(L(\text{null})), i_9) \quad (4)$$

$$f_P(f_V(L(L(o_{20})), i_{19}), L(L(o_5)), i_9) \rightarrow f_V(L(L(L(o_{20}))), i_9) \quad (5)$$

$$f_{A'}(\dots) \rightarrow f_{B'}(f_A(\dots), \dots)$$

$$f_{B'}(f_V(\dots), \dots) \rightarrow f_{D'}(f_{D'}(\dots), \dots)$$

```
static void cappE(int j) {
    List a = new List();
    if (j > 0) {
        a.appE(j);
        while (a.n == null) {}
    }
}
```



with B'

$$f_A(L(\text{null}), i_6) \rightarrow f_G(L(\text{null}), i_6) \quad | i_6 \leq 0 \quad (1)$$

$$f_A(L(\text{null}), i_7) \rightarrow f_P(f_A(L(\text{null}), i_7 - 1), L(L(\text{null})), i_7) \quad | i_7 > 0 \quad (2)$$

$$f_A(L(L(o_5)), i_3) \rightarrow f_P(f_A(L(o_5), i_3), L(L(o_5)), i_3) \quad (3)$$

$$f_P(f_G(L(\text{null}), i_{12}), L(L(\text{null})), i_9) \rightarrow f_V(L(L(\text{null})), i_9) \quad (4)$$

$$f_P(f_V(L(L(o_{20})), i_{19}), L(L(o_5)), i_9) \rightarrow f_V(L(L(L(o_{20}))), i_9) \quad (5)$$

$$f_{A'}(\dots) \rightarrow f_{B'}(f_A(\dots), \dots)$$

$$f_{B'}(f_V(\dots), \dots) \rightarrow f_{D'}(f_{D'}(\dots), \dots)$$

```
static void cappE(int j) {  
    List a = new List();  
    if (j > 0) {  
        a.appE(j);  
        while (a.n == null) {}  
    }  
}
```

- termination graphs and TRSs for a method can be re-used whenever the method is called

$$f_A(L(\text{null}), i_6) \rightarrow f_G(L(\text{null}), i_6) \quad | i_6 \leq 0 \quad (1)$$

$$f_A(L(\text{null}), i_7) \rightarrow f_P(f_A(L(\text{null}), i_7 - 1), L(L(\text{null})), i_7) \quad | i_7 > 0 \quad (2)$$

$$f_A(L(L(o_5)), i_3) \rightarrow f_P(f_A(L(o_5), i_3), L(L(o_5)), i_3) \quad (3)$$

$$f_P(f_G(L(\text{null}), i_{12}), L(L(\text{null})), i_9) \rightarrow f_V(L(L(\text{null})), i_9) \quad (4)$$

$$f_P(f_V(L(L(o_{20})), i_{19}), L(L(o_5)), i_9) \rightarrow f_V(L(L(L(o_{20}))), i_9) \quad (5)$$

$$f_{A'}(\dots) \rightarrow f_{B'}(f_A(\dots), \dots)$$

$$f_{B'}(f_V(\dots), \dots) \rightarrow f_{D'}(f_{D'}(\dots), \dots)$$

```
static void cappE(int j) {  
    List a = new List();  
    if (j > 0) {  
        a.appE(j);  
        while (a.n == null) {}  
    }  
}
```

- termination graphs and TRSs for a method can be re-used whenever the method is called

- *modular* termination proofs

⇒ termination of cappE follows from termination of appE

$$f_A(L(\text{null}), i_6) \rightarrow f_G(L(\text{null}), i_6) \quad | i_6 \leq 0 \quad (1)$$

$$f_A(L(\text{null}), i_7) \rightarrow f_P(f_A(L(\text{null}), i_7 - 1), L(L(\text{null})), i_7) \quad | i_7 > 0 \quad (2)$$

$$f_A(L(L(o_5)), i_3) \rightarrow f_P(f_A(L(o_5), i_3), L(L(o_5)), i_3) \quad (3)$$

$$f_P(f_G(L(\text{null}), i_{12}), L(L(\text{null})), i_9) \rightarrow f_V(L(L(\text{null})), i_9) \quad (4)$$

$$f_P(f_V(L(L(o_{20})), i_{19}), L(L(o_5)), i_9) \rightarrow f_V(L(L(L(o_{20}))), i_9) \quad (5)$$

$$f_{A'}(\dots) \rightarrow f_{B'}(f_A(\dots), \dots)$$

$$f_{B'}(f_V(\dots), \dots) \rightarrow f_{D'}(f_{D'}(\dots), \dots)$$

```
static void cappE(int j) {  
    List a = new List();  
    if (j > 0) {  
        a.appE(j);  
        while (a.n == null) {}  
    }  
}
```

- termination graphs and TRSs for a method can be re-used whenever the method is called

- *modular* termination proofs

⇒ termination of cappE follows from termination of appE

- modularity is crucial for scalability

From Termination Graphs to TRSs

Theorem 1

Every JBC-computation of concrete states corresponds to a *computation path* in the termination graph

From Termination Graphs to TRSs

Theorem 1

Every JBC-computation of concrete states corresponds to a *computation path* in the termination graph

Theorem 2

TRS corresponding to termination graph is terminating \Rightarrow

From Termination Graphs to TRSs

Theorem 1

Every JBC-computation of concrete states corresponds to a *computation path* in the termination graph

Theorem 2

TRS corresponding to termination graph is terminating \Rightarrow

termination graph has no infinite computation path \Rightarrow

From Termination Graphs to TRSs

Theorem 1

Every JBC-computation of concrete states corresponds to a *computation path* in the termination graph

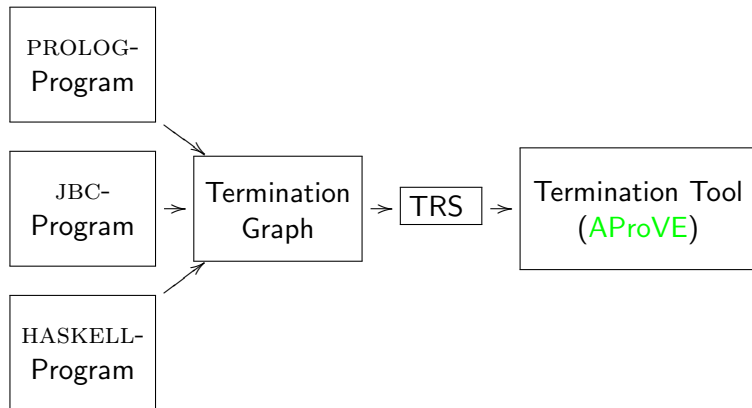
Theorem 2

TRS corresponding to termination graph is terminating \Rightarrow

termination graph has no infinite computation path \Rightarrow

JBC-program terminates for all states represented in termination graph

Modular Termination Analysis for JAVA BYTECODE by Term Rewriting



Modular Termination Analysis for JAVA BYTECODE by Term Rewriting

- implemented in AProVE and evaluated on collection of 216 JBC-programs (including the *Termination Problem Data Base*)

Modular Termination Analysis for JAVA BYTECODE by Term Rewriting

- implemented in AProVE and evaluated on collection of 216 JBC-programs (including the *Termination Problem Data Base*)

	Success	Failure	Timeout	Runtime
AProVE 2011	175	0	41	28.2
AProVE 2010	118	16	82	51.1
Julia	153	63	0	2.4
COSTA	120	95	1	5.4

Modular Termination Analysis for JAVA BYTECODE by Term Rewriting

- implemented in AProVE and evaluated on collection of 216 JBC-programs (including the *Termination Problem Data Base*)

	Success	Failure	Timeout	Runtime
AProVE 2011	175	0	41	28.2
AProVE 2010	118	16	82	51.1
Julia	153	63	0	2.4
COSTA	120	95	1	5.4

- improvement over AProVE 2010: [modularity](#) & [recursion](#)

Modular Termination Analysis for JAVA BYTECODE by Term Rewriting

- implemented in AProVE and evaluated on collection of 216 JBC-programs (including the *Termination Problem Data Base*)

	Success	Failure	Timeout	Runtime
AProVE 2011	175	0	41	28.2
AProVE 2010	118	16	82	51.1
Julia	153	63	0	2.4
COSTA	120	95	1	5.4

- improvement over AProVE 2010: [modularity](#) & [recursion](#)
- AProVE at the *International Termination Competition*:
powerful for [JBC](#), [HASKELL](#), [PROLOG](#), [term rewriting](#)

Modular Termination Analysis for JAVA BYTECODE by Term Rewriting

- implemented in AProVE and evaluated on collection of 216 JBC-programs (including the *Termination Problem Data Base*)

	Success	Failure	Timeout	Runtime
AProVE 2011	175	0	41	28.2
AProVE 2010	118	16	82	51.1
Julia	153	63	0	2.4
COSTA	120	95	1	5.4

- improvement over AProVE 2010: **modularity & recursion**
- AProVE at the *International Termination Competition*:
powerful for **JBC, HASKELL, PROLOG, term rewriting**
- <http://aprove.informatik.rwth-aachen.de>

Modular Termination Analysis for JAVA BYTECODE by Term Rewriting

- implemented in AProVE and evaluated on collection of 216 JBC-programs (including the *Termination Problem Data Base*)

	Success	Failure	Timeout	Runtime
AProVE 2011	175	0	41	28.2
AProVE 2010	118	16	82	51.1
Julia	153	63	0	2.4
COSTA	120	95	1	5.4

- improvement over AProVE 2010: **modularity & recursion**
- AProVE at the *International Termination Competition*:
powerful for **JBC, HASKELL, PROLOG, term rewriting**
- <http://aprove.informatik.rwth-aachen.de>
- termination of “real” languages can be analyzed automatically,
term rewriting is a suitable approach