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)*
- MultumNonMulta *(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)*
- Termptation *(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?

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)

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 ⇒ 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 ⇒ termination graph

      termination graph captures side effects, sharing, cyclic data objects etc.
    - transform termination graph ⇒ TRS

Termination of Imperative Programs

- **New approach** (RTA '10)
  - Frontend
    - evaluate JBC a few steps ⇒ termination graph

      termination graph captures side effects, sharing, cyclic data objects etc.
    - transform termination graph ⇒ 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 ⇒ termination graph

      termination graph captures side effects, sharing, cyclic data objects etc.
    - transform termination graph ⇒ 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 ⇒ termination graph

      termination graph captures side effects, sharing, cyclic data objects etc.
    - transform termination graph ⇒ TRS
  - Backend
    - prove termination of the resulting TRS
      (using existing techniques & tools)
- implemented in **AProVE**
  - modular termination graphs (separate graphs for separate methods)
    ⇒ scalability & recursion

Termination of Imperative Programs

- **New approach** (RTA '10, RTA '11)
  - Frontend
    - evaluate JBC a few steps ⇒ termination graph

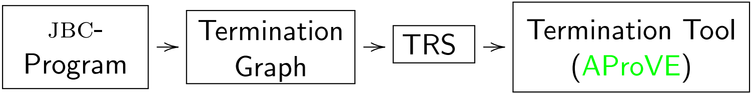
      termination graph captures side effects, sharing, cyclic data objects etc.
    - transform termination graph ⇒ TRS
  - Backend
    - prove termination of the resulting TRS
      (using existing techniques & tools)
- implemented in **AProVE**
  - modular termination graphs (separate graphs for separate methods)
    ⇒ scalability & recursion
  - successfully evaluated on JBC-collection

Termination of Imperative Programs

- **New approach** (RTA '10, RTA '11)
  - Frontend
    - evaluate JBC a few steps ⇒ termination graph

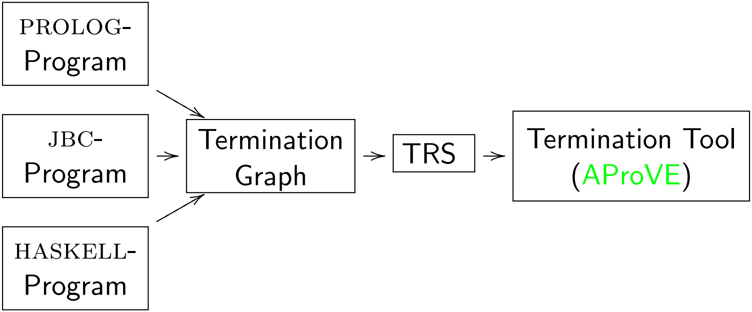
      termination graph captures side effects, sharing, cyclic data objects etc.
    - transform termination graph ⇒ TRS
  - Backend
    - prove termination of the resulting TRS
      (using existing techniques & tools)
- implemented in **AProVE**
  - modular termination graphs (separate graphs for separate 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) $\Rightarrow$ 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) $\Rightarrow$ 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: $\mathsf{List}(0,\ \mathsf{List}(1,\ \mathsf{List}(2, \mathsf{null})\,)\,)$

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: $\mathsf{List}(0,\ \mathsf{List}(1,\ \mathsf{List}(2, \mathsf{null})\,)\,)$
  - 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: $\mathsf{List}(0,\ \mathsf{List}(1,\ \mathsf{List}(2, \mathsf{null})\,)\,)$
  - TRS-techniques generate suitable orders to compare arbitrary terms
  - particularly powerful on user-defined data types

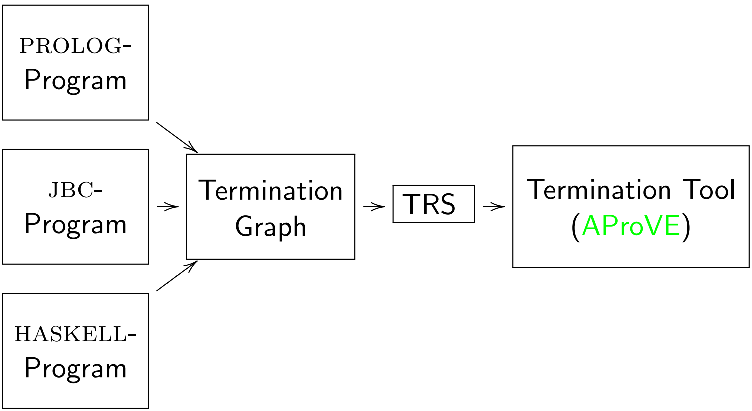
Termination of Imperative Programs

- other techniques:
  abstract objects to numbers
  - `List`-object representing $[0, 1, 2]$
    is abstracted to length 3

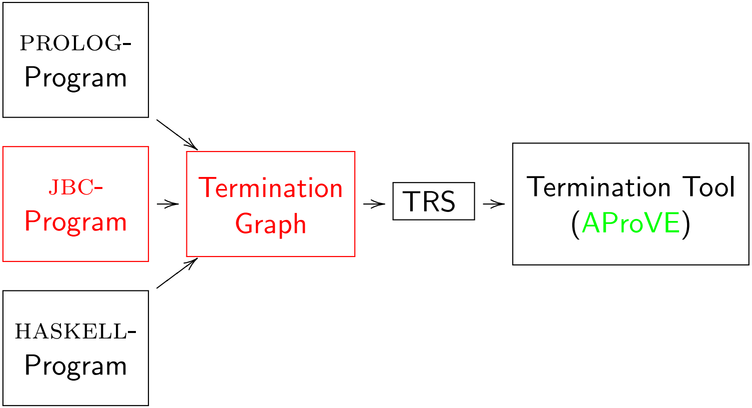
```
public class List {
  int value;
  List next;
}
```

- our technique:
  abstract objects to terms
  - introduce function symbol for every class
  - `List`-object representing $[0, 1, 2]$
    is abstracted to term: $\mathsf{List}(0,\ \mathsf{List}(1,\ \mathsf{List}(2, \mathsf{null})\,)\,)$
  - 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)
```

$o_1, i_3 \mid 0 \mid \texttt{t}: o_1, \texttt{i}: i_3 \mid \varepsilon$

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

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

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)
```

$o_1, i_3 \mid 0 \mid \texttt{t}: o_1, \texttt{i}: i_3 \mid \varepsilon$

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

o_2: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)
```

$o_1, i_3 \mid 0 \mid \mathtt{t}: o_1, \mathtt{i}: i_3 \mid \varepsilon$

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

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

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)
```

$o_1, i_3 \mid 0 \mid \texttt{t}: o_1, \texttt{i}: i_3 \mid \varepsilon$

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

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

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)
```

$o_1, i_3 \mid 0 \mid \texttt{t}: o_1, \texttt{i}: i_3 \mid \varepsilon$

o_1: List(n $= o_2$) 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)

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)
```

$o_1, i_3 \mid 0 \mid \texttt{t}: o_1, \texttt{i}: i_3 \mid \varepsilon$

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

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

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)
```

$o_1, i_3 \mid 0 \mid \mathtt{t}: o_1, \mathtt{i}: i_3 \mid \varepsilon$

o_1:List(n$=o_2$) 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)
```

$o_1, i_3 \mid 0 \mid \texttt{t}: o_1, \texttt{i}: i_3 \mid \varepsilon$

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

- 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)
```

$o_1, i_3 \mid 0 \mid \mathtt{t}: o_1, \mathtt{i}: i_3 \mid \varepsilon$

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

- 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	0	$\mathtt{t}: o_1, \mathtt{i}: i_3$	ε

o_1: List(n $= o_2$) 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

- 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)
```

$o_1, i_3 \mid 0 \mid \mathtt{t}: o_1, \mathtt{i}: i_3 \mid \varepsilon$

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

- 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

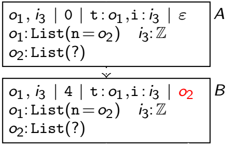
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```

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

State A:

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

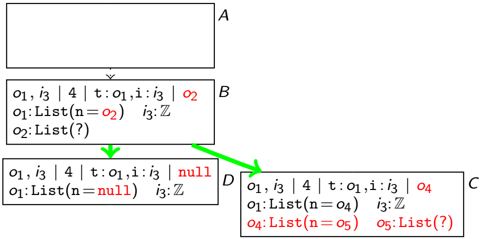
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



State *B*:

- "aload_0" loads value o_1 of this on opstack
- "getfield n" replaces o_1 by o_2 on opstack (value of its n-field)
- *A* connected to *B* by *evaluation edge*

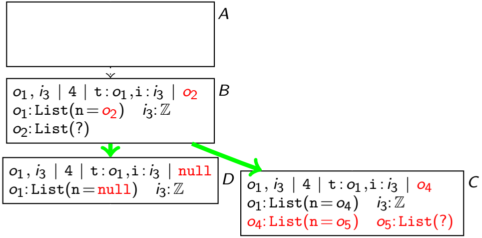
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



States C and D:

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

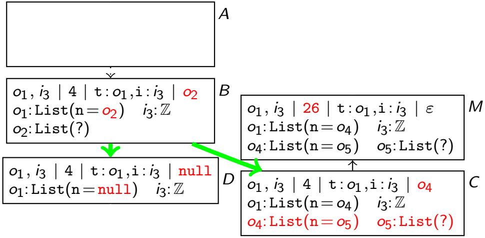
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



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 "$o_4 : \texttt{List}(\texttt{n} = o_5)$"

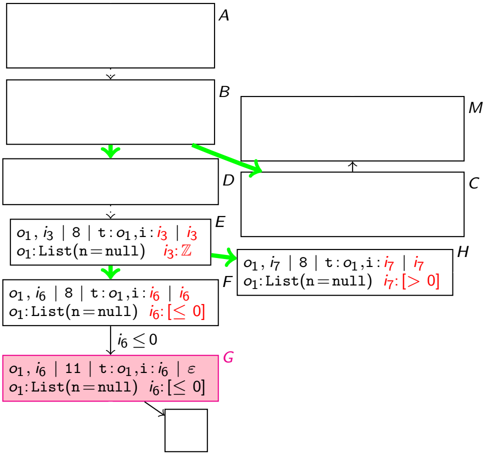
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



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 : List(n = o_5)", *evaluation* to M

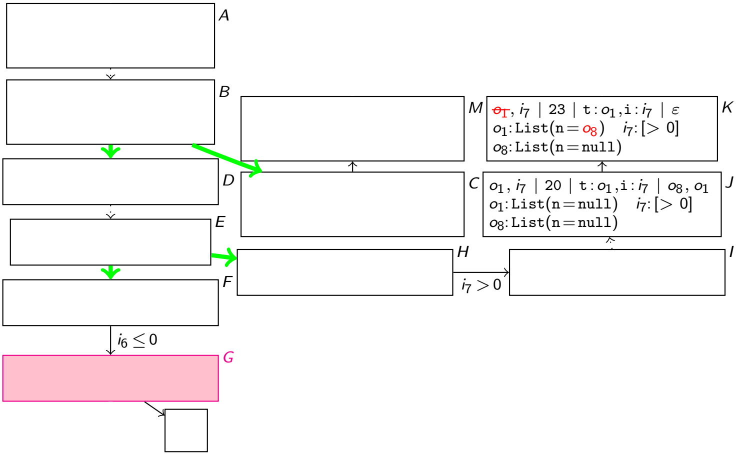
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



State G:

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

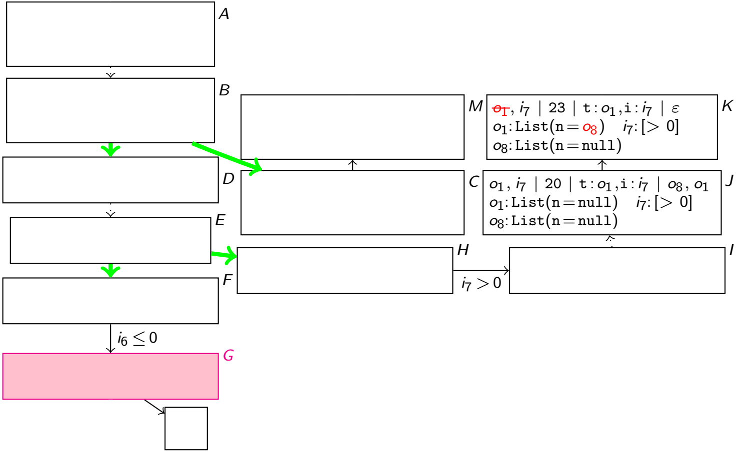
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



State K:

- "putfield n" writes o_8 to n-field of o_1

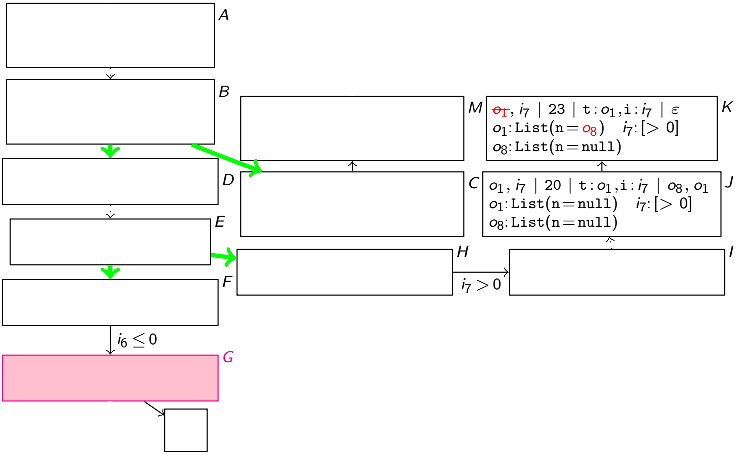
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



State K:

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

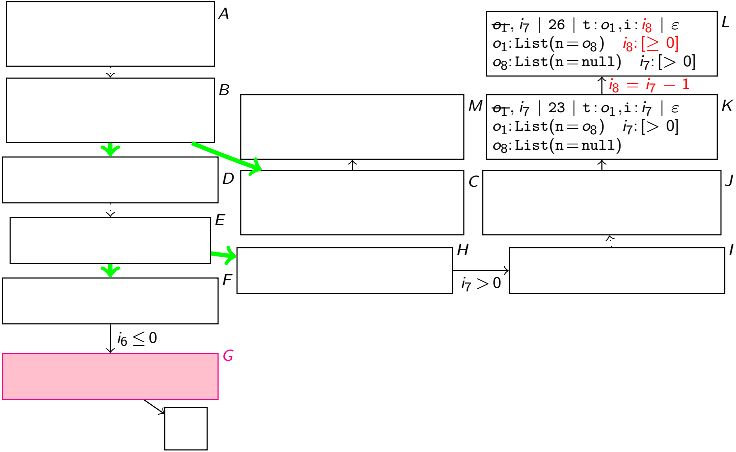
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



State K:

- "putfield n" writes o_8 to n-field of o_1
- *side effect* which changes original *input argument* o_1
- switch boolean flag of input argument o_1 to *false*

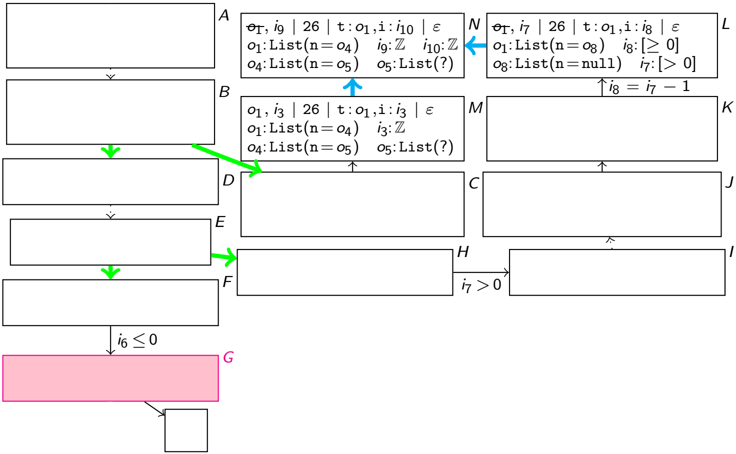
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



State L:

- decrement i_7 by 1

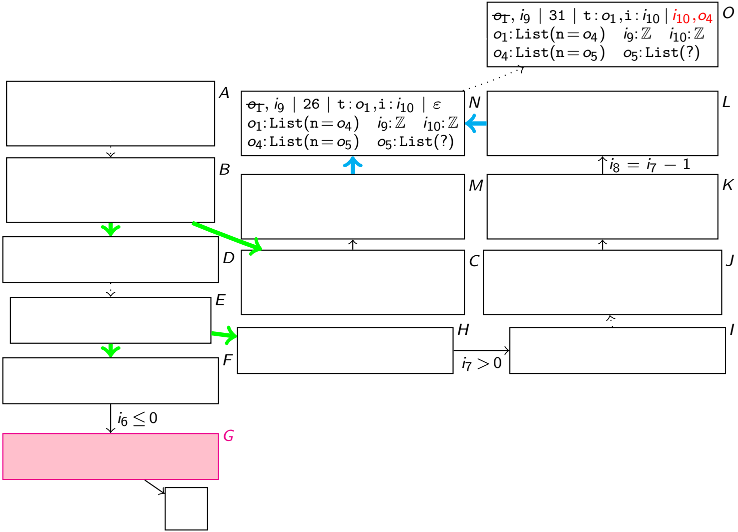
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



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*)

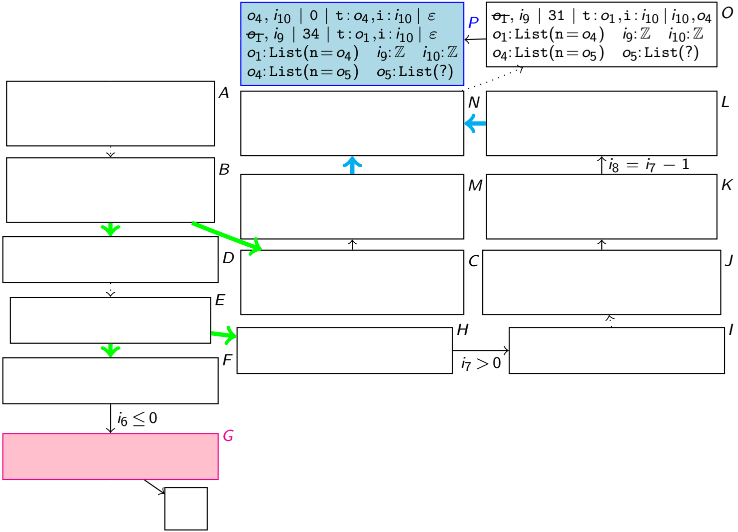
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



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

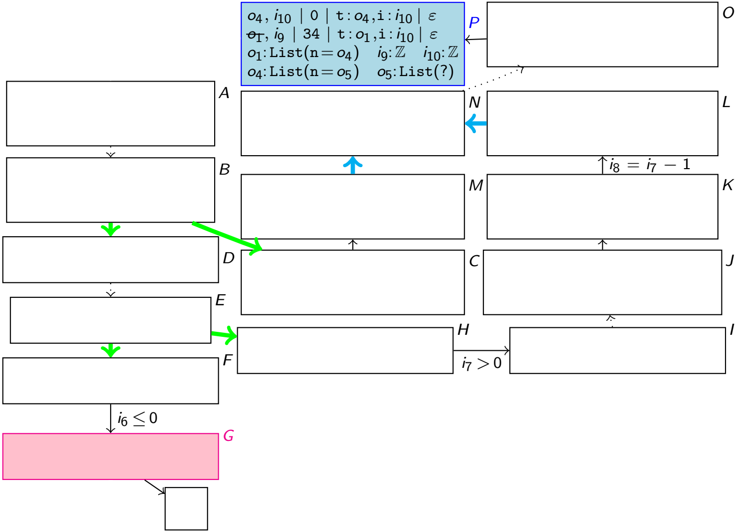
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



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

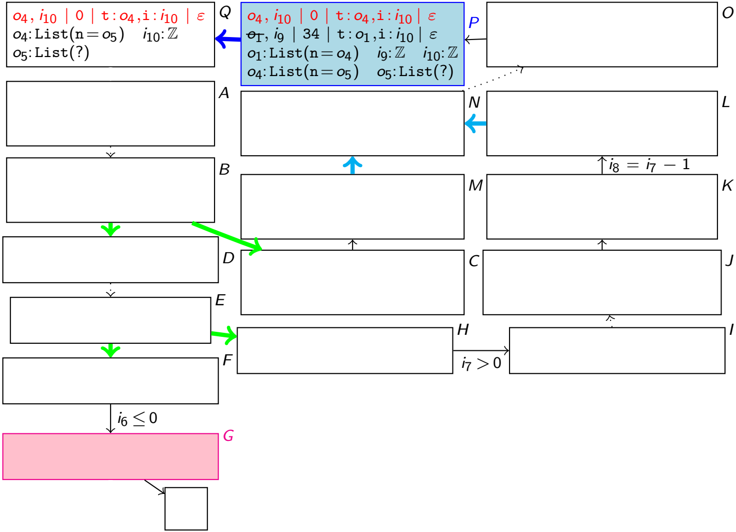
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



State Q:

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

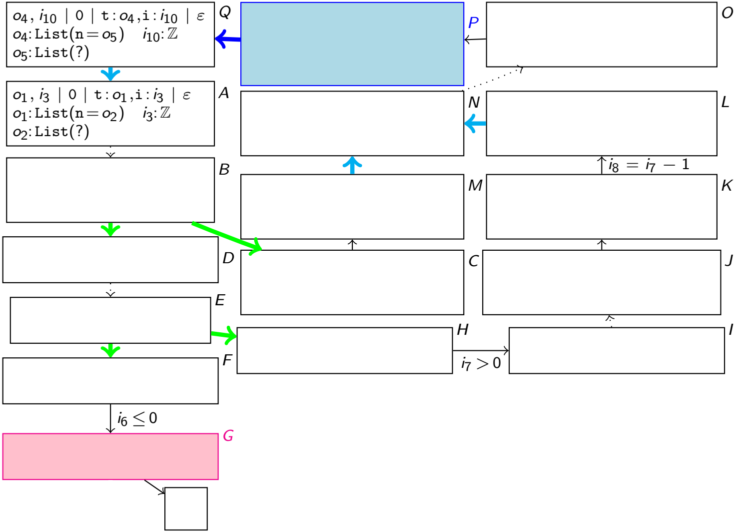
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



State Q:

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

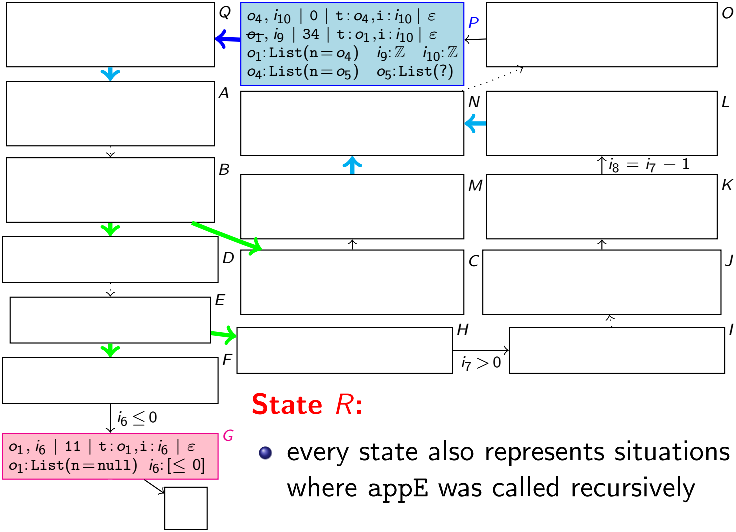
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



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*)

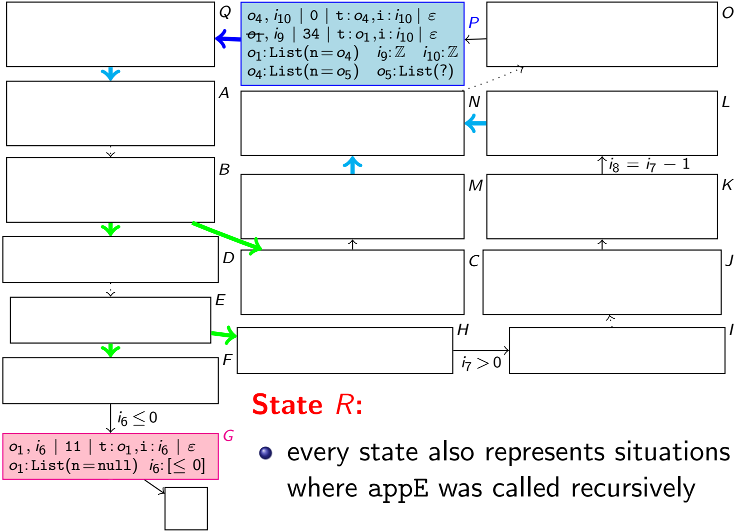
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



State R:

- every state also represents situations where appE was called recursively

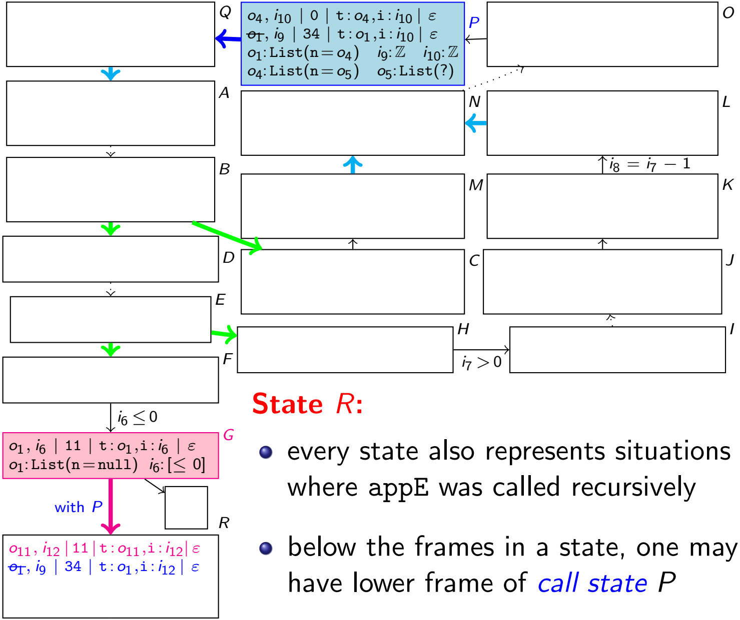
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



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*

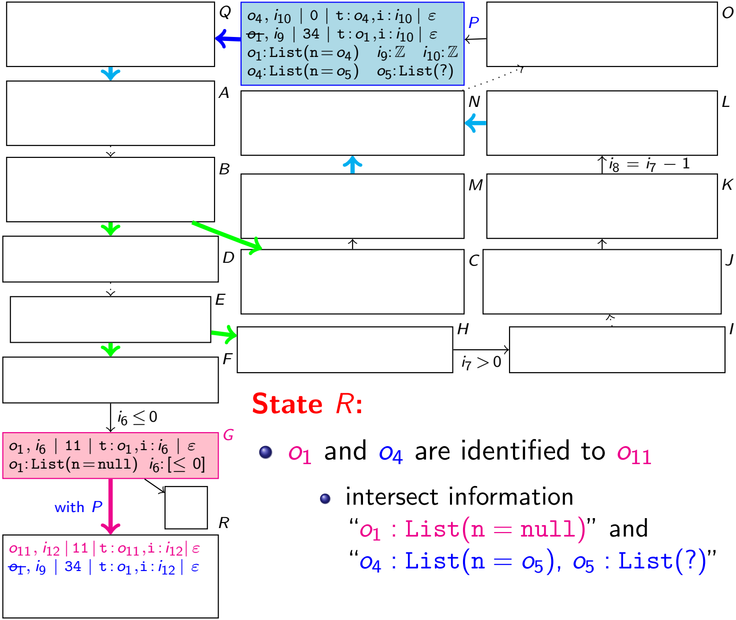
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



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*)

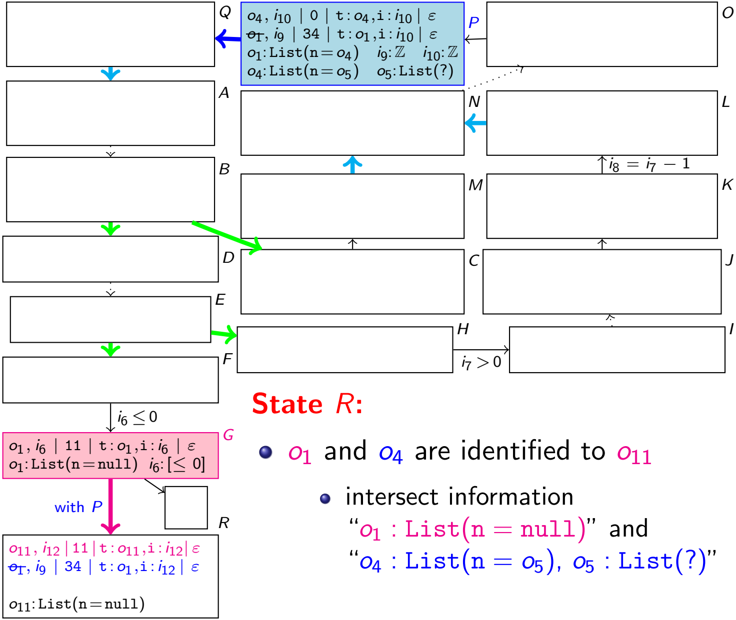
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



State R:

- o_1 and o_4 are identified to o_{11}
  - intersect information "o_1 : List(n = null)" and "o_4 : List(n = o_5), o_5 : List(?)"

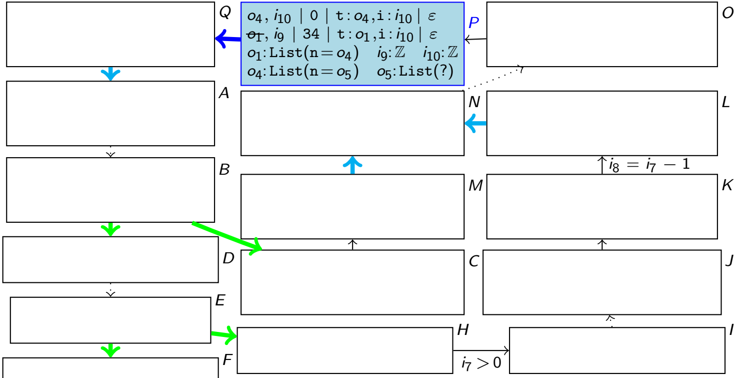
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```

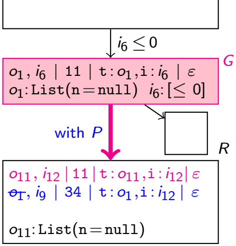


State R:

- o_1 and o_4 are identified to o_{11}
  - intersect information
    "o_1 : List(n = null)" and
    "o_4 : List(n = o_5), o_5 : 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
```

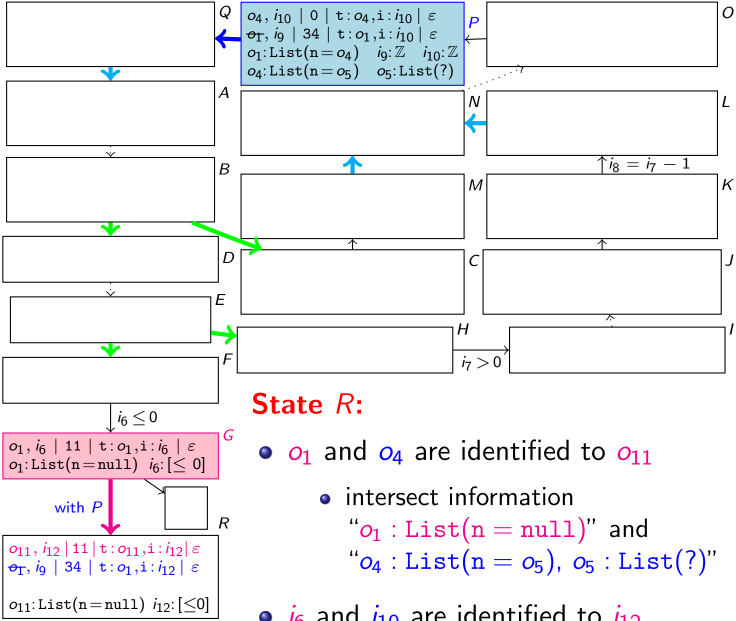




State R:

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

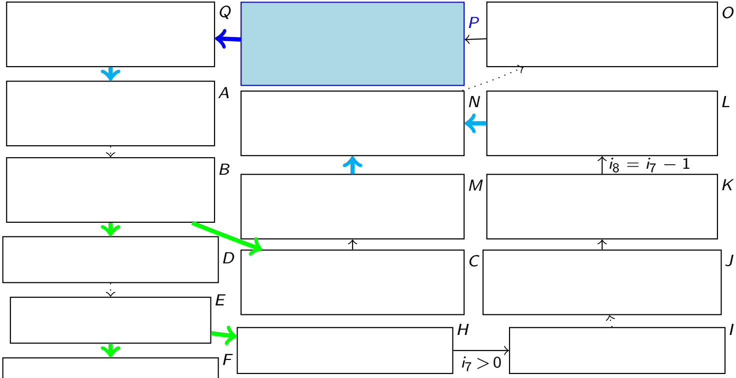
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```

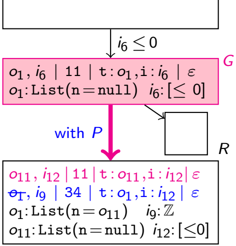


State R:

- o_1 and o_4 are identified to o_{11}
  - intersect information "o_1 : List(n = null)" and "o_4 : List(n = o_5), o_5 : 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
```

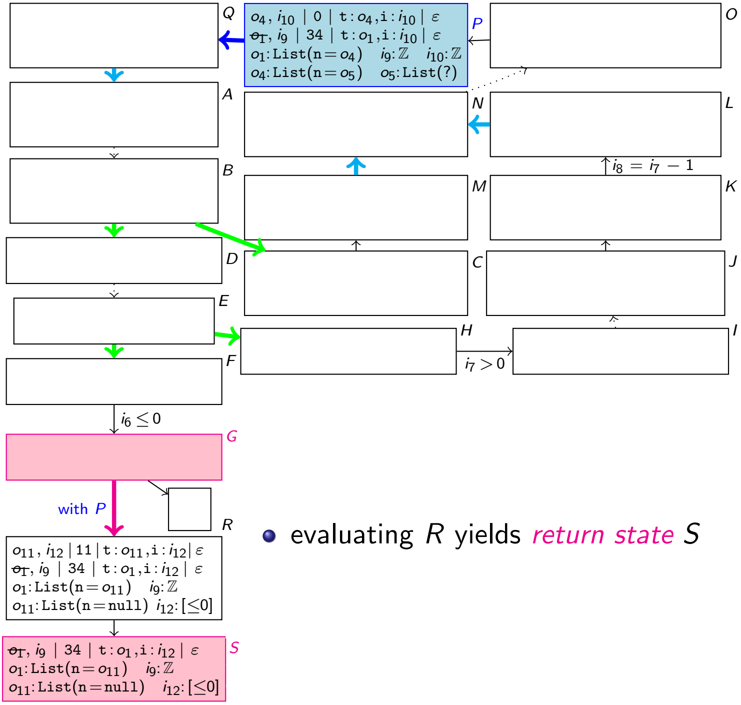




State R:

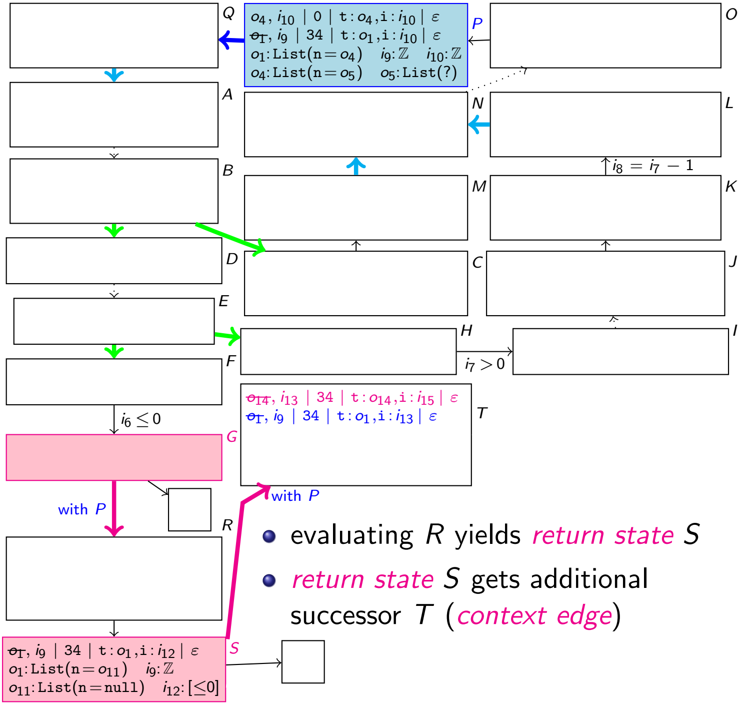
- o_1 and o_4 are identified to o_{11}
  - intersect information "o_1 : List(n = null)" and "o_4 : List(n = o_5), o_5 : List(?)"
- i_6 and i_{10} are identified to i_{12}
  - intersect information "i_6 : [≤ 0]" and "i_{10} : $\mathbb{Z}$"

```
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
```



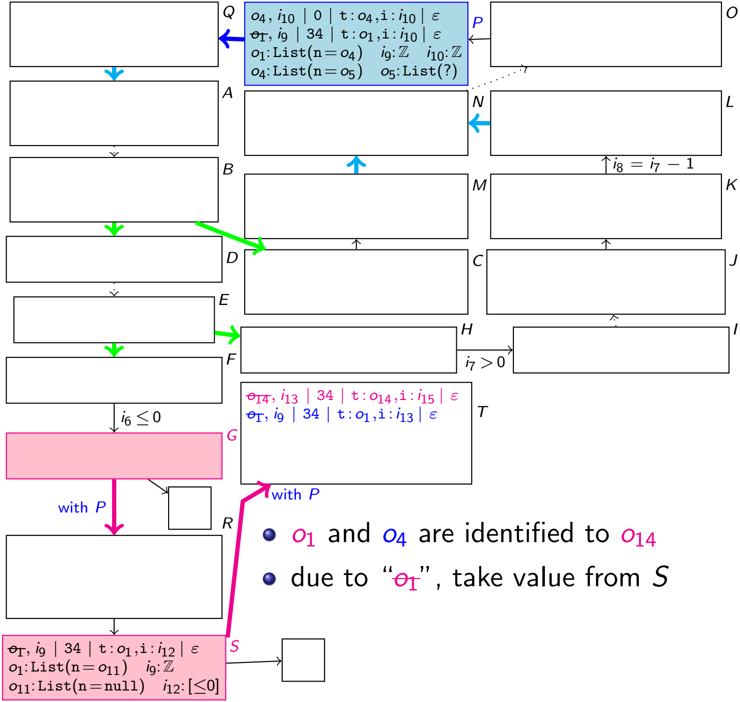
- evaluating R yields *return state* S

```
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
```



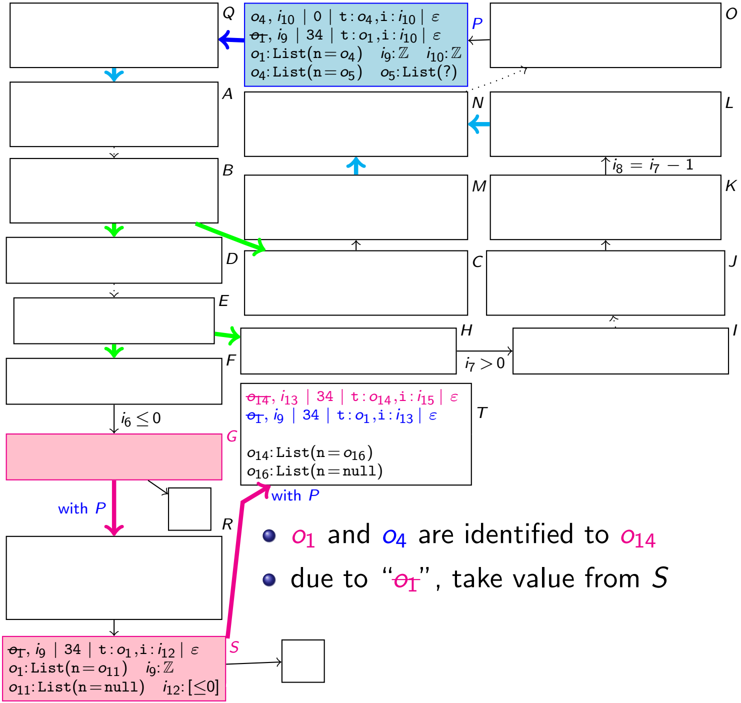
- evaluating R yields *return state* S
- *return state* S gets additional successor T (*context edge*)

```
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
```



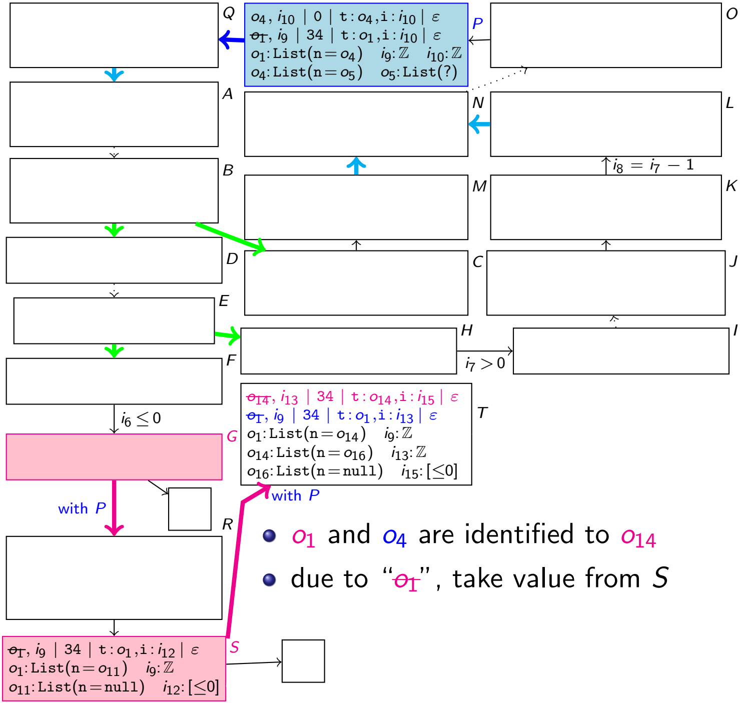
- o_1 and o_4 are identified to o_{14}
- due to "$o_{\top}$", take value from S

```
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
```



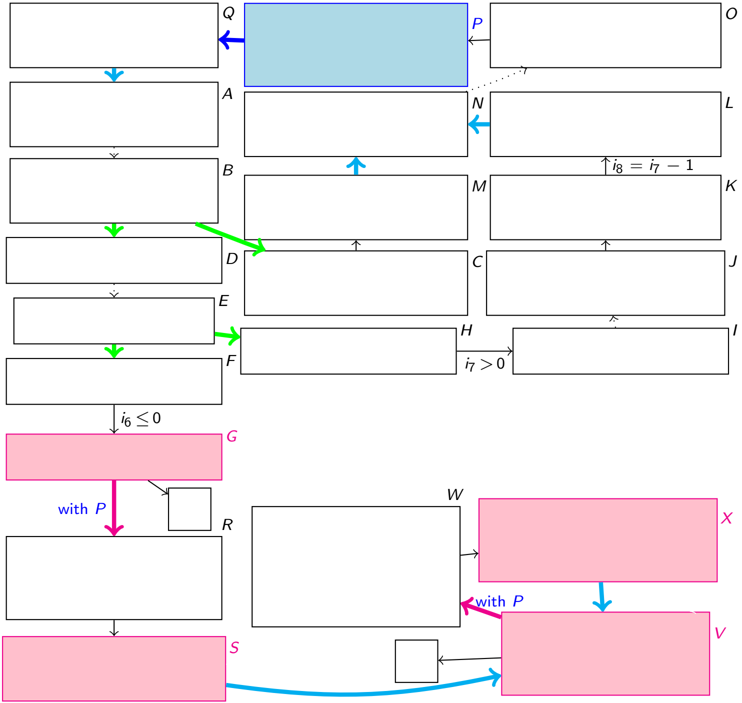
- o_1 and o_4 are identified to o_{14}
- due to "o_1", take value from S

```
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
```

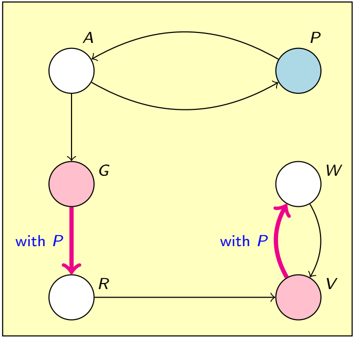


- o_1 and o_4 are identified to o_{14}
- due to "~~o_T~~", take value from S

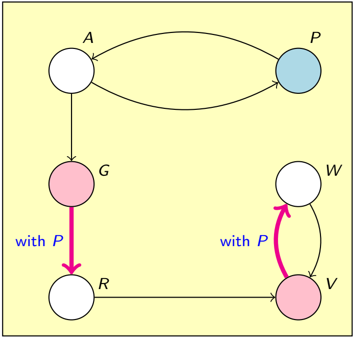
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



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



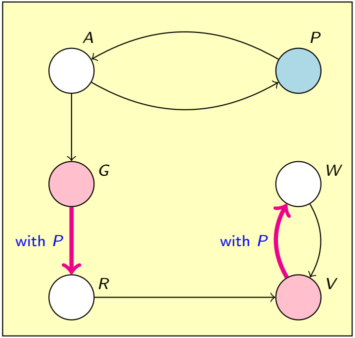
```
public void appE(int i) {
  if (n == null) {
   if (i <= 0) return;
   n = new List();
   i--;
  }
  n.appE(i);
} }
```



Termination Graphs

- expand nodes until all leaves correspond to program ends

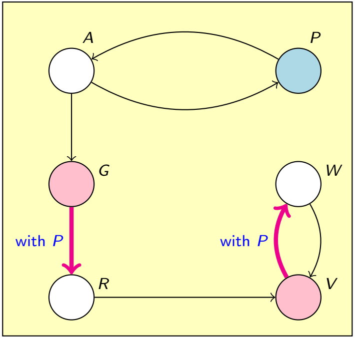
```
public void appE(int i) {
  if (n == null) {
   if (i <= 0) return;
   n = new List();
   i--;
  }
  n.appE(i);
} }
```



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);
} }
```

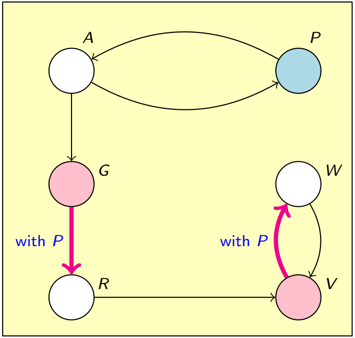


Termination Graphs

- 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);
} }
```

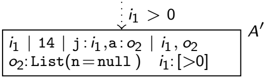
```
static void cappE(int j) {
  List a = new List();
  if (j > 0) {
   a.appE(j);
   while (a.n == null) {}
} }
```



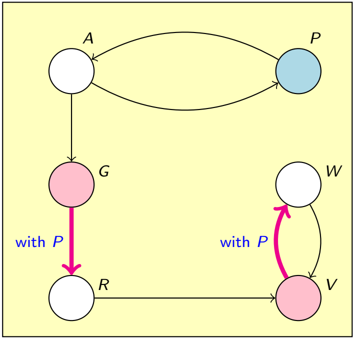
Method `cappE`

- creates new list `a`
- calls `appE` to append $\texttt{j} > 0$ 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);
} }
```




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

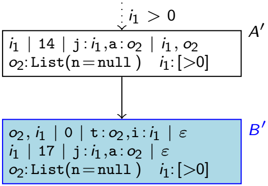


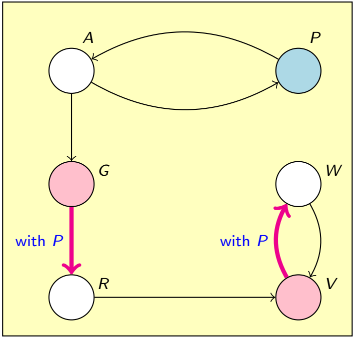
Method cappE

- 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);
} }
```

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



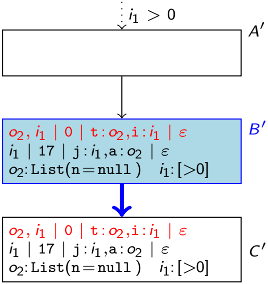


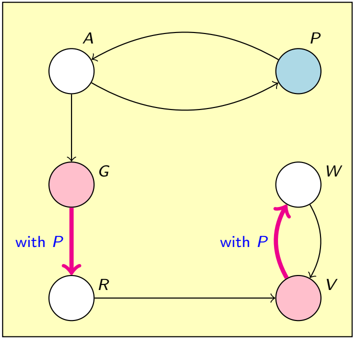
State B':

- 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

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

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



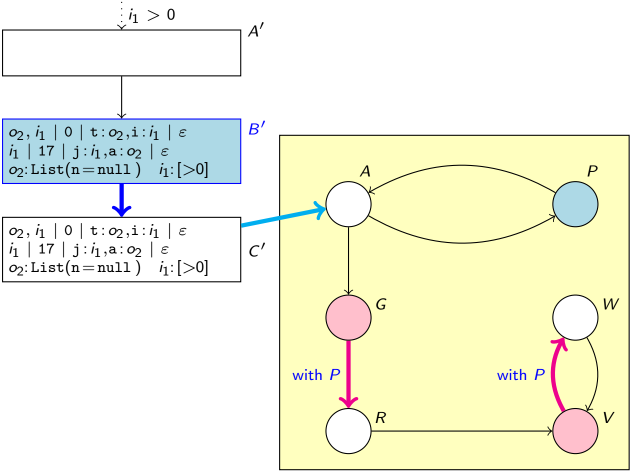


State C':

- *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) {}
} }
```

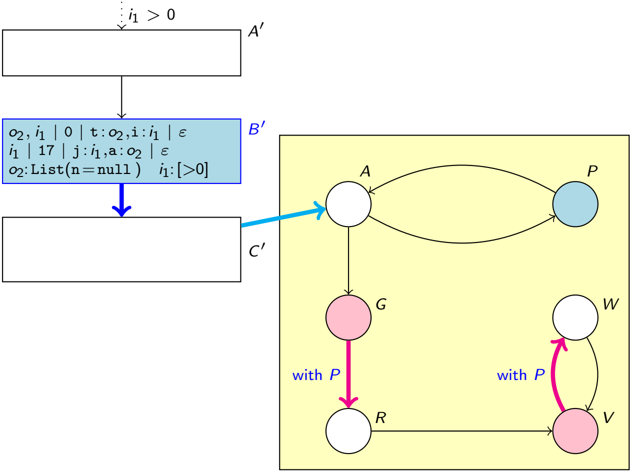


State C':

- *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);
} }
```

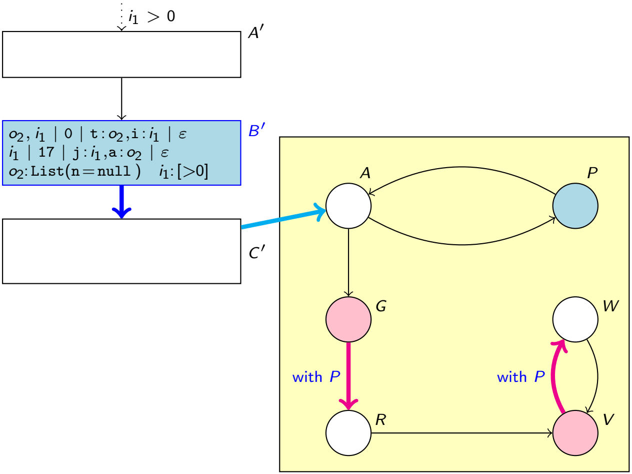
```
static void cappE(int j) {
  List a = new List();
  if (j > 0) {
   a.appE(j);
   while (a.n == null) {}
} }
```



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) {}
} }
```

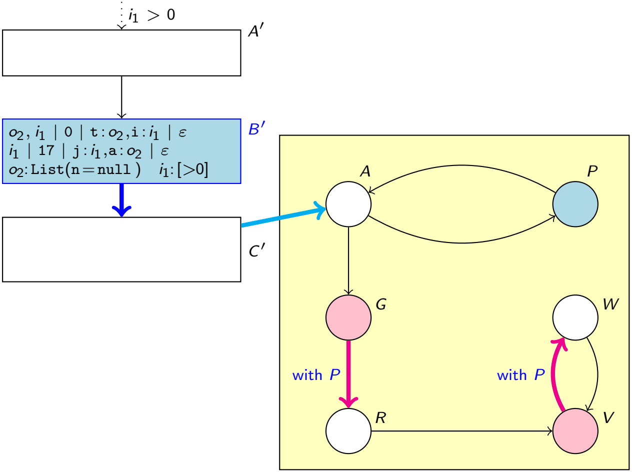


Every return state has context edge with every call state of appE

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

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

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

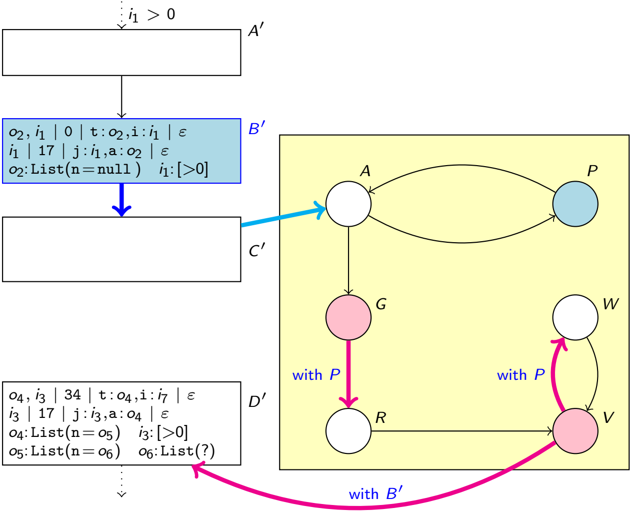


Every return state has context edge with every call state of appE

- 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')

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

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

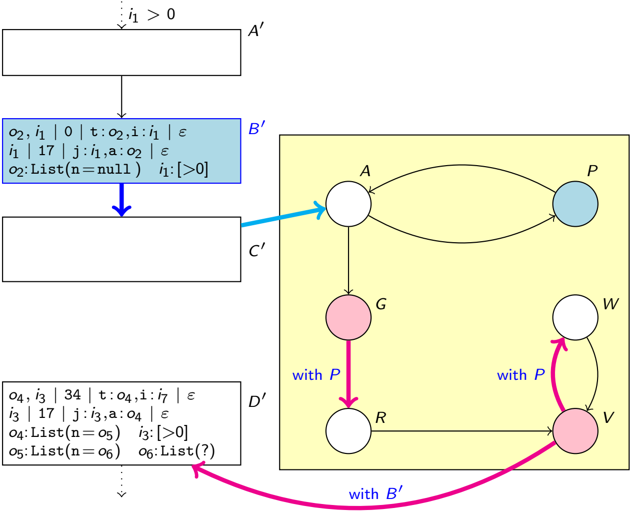


Every return state has context edge with every call state of appE

- 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);
} }
```

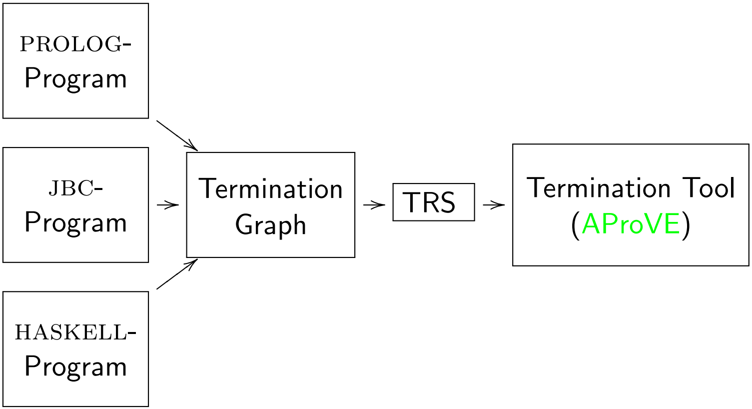
```
static void cappE(int j) {
  List a = new List();
  if (j > 0) {
   a.appE(j);
   while (a.n == null) {}
} }
```



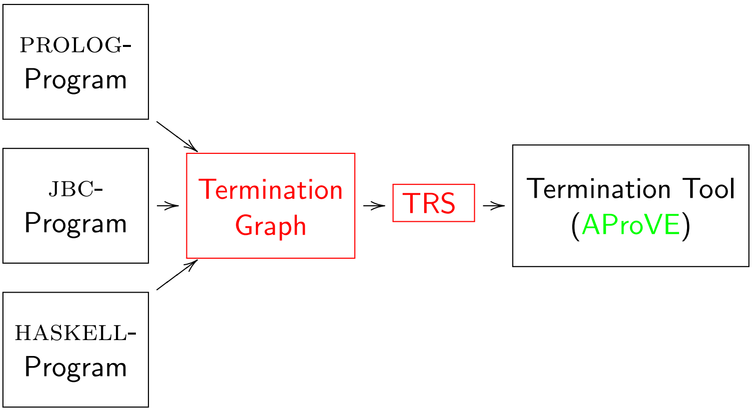
Every return state has context edge with every call state of `appE`

- 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

$$\begin{array}{|l|}\hline
{\color{red}o_4}, i_{10} \mid 0 \mid \mathtt{t}: o_4, \mathtt{i}: i_{10} \mid \varepsilon \\
\not{o_1}, i_9 \mid 34 \mid \mathtt{t}: o_1, \mathtt{i}: i_{10} \mid \varepsilon \\
o_1{:}\mathtt{List}(\mathtt{n}{=}o_4) \quad i_9{:}\mathbb{Z} \quad i_{10}{:}\mathbb{Z} \\
{\color{red}o_4{:}\mathtt{List}(\mathtt{n}{=}o_5) \quad o_5{:}\mathtt{List}(?)} \\ \hline
\end{array}\; P$$

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

$${\color{red}\underbrace{\mathsf{L}(o_5)}_{o_4}}$$

Transforming States to Terms

$o_4, i_{10} \mid 0 \mid \mathtt{t}: o_4, \mathtt{i}: i_{10} \mid \varepsilon$	P
~~o_1~~$, i_9 \mid 34 \mid \mathtt{t}: o_1, \mathtt{i}: i_{10} \mid \varepsilon$	
$o_1{:}\mathtt{List}(\mathtt{n}{=}o_4) \quad i_9{:}\mathbb{Z} \quad i_{10}{:}\mathbb{Z}$	
$o_4{:}\mathtt{List}(\mathtt{n}{=}o_5) \quad o_5{:}\mathtt{List(?)}$	

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

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

Transforming States to Terms

$o_4, i_{10} \mid 0 \mid \texttt{t}: o_4, \texttt{i}: i_{10} \mid \varepsilon$	
$\not{o_1}, i_9 \mid 34 \mid \texttt{t}: o_1, \texttt{i}: i_{10} \mid \varepsilon$	P
$o_1{:}\texttt{List}(\texttt{n}{=}o_4) \quad i_9{:}\mathbb{Z} \quad i_{10}{:}\mathbb{Z}$	
$o_4{:}\texttt{List}(\texttt{n}{=}o_5) \quad o_5{:}\texttt{List(?)}$	

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

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

Transforming States to Terms

$$\boxed{\begin{array}{l} o_4, i_{10} \mid 0 \mid \mathtt{t}: o_4, \mathtt{i}: i_{10} \mid \varepsilon \\ \not{o_1}, i_9 \mid 34 \mid \mathtt{t}: o_1, \mathtt{i}: i_{10} \mid \varepsilon \\ o_1{:}\mathtt{List}(\mathtt{n}{=}o_4) \quad i_9{:}\mathbb{Z} \quad i_{10}{:}\mathbb{Z} \\ o_4{:}\mathtt{List}(\mathtt{n}{=}o_5) \quad o_5{:}\mathtt{List}(?) \end{array}} \; P$$

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

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

Transforming States to Terms

$o_4, i_{10} \mid 0 \mid \texttt{t}: o_4, \texttt{i}: i_{10} \mid \varepsilon$		
$o_1, i_9 \mid 34 \mid \texttt{t}: o_1, \texttt{i}: i_{10} \mid \varepsilon$		
$o_1{:}\texttt{List}(\texttt{n}=o_4)$	$i_9{:}\mathbb{Z}$	$i_{10}{:}\mathbb{Z}$
$o_4{:}\texttt{List}(\texttt{n}=o_5)$	$o_5{:}\texttt{List}(?)$	

P

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

- Extension for *class hierarchies*
  (nested constructor symbols)

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

Transforming States to Terms

$$\begin{array}{|l|}\hline \color{red}{o_4, i_{10} \mid 0 \mid \mathtt{t}: o_4, \mathtt{i}: i_{10} \mid \varepsilon} \\ \not{o_1}, i_9 \mid 34 \mid \mathtt{t}: o_1, \mathtt{i}: i_{10} \mid \varepsilon \\ o_1{:}\mathtt{List}(\mathtt{n}{=}o_4) \quad i_9{:}\mathbb{Z} \quad i_{10}{:}\mathbb{Z} \\ o_4{:}\mathtt{List}(\mathtt{n}{=}o_5) \quad o_5{:}\mathtt{List}(?) \\ \hline \end{array}\; P$$

- For every stack frame of state s at position pp, introduce function symbol $\mathsf{f}_{s,pp}$.

$$\color{red}{\mathsf{f}_{P,0}(}\quad \underbrace{\mathsf{L}(o_5)}_{o_4},\ i_{10},\ \underbrace{\mathsf{L}(o_5)}_{o_4},\ i_{10}\color{red}{)}$$

Transforming States to Terms

$$\boxed{\begin{array}{l} o_4, i_{10} \mid 0 \mid \mathtt{t}: o_4, \mathtt{i}: i_{10} \mid \varepsilon \\ \not{o_1}, i_9 \mid 34 \mid \mathtt{t}: o_1, \mathtt{i}: i_{10} \mid \varepsilon \\ o_1{:}\mathtt{List}(\mathtt{n}{=}o_4) \quad i_9{:}\mathbb{Z} \quad i_{10}{:}\mathbb{Z} \\ o_4{:}\mathtt{List}(\mathtt{n}{=}o_5) \quad o_5{:}\mathtt{List}(?) \end{array}} \; P$$

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

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

Transforming States to Terms

$o_4, i_{10} \mid 0 \mid \mathtt{t}: o_4, \mathtt{i}: i_{10} \mid \varepsilon$	
$\sout{o_1}, i_9 \mid 34 \mid \mathtt{t}: o_1, \mathtt{i}: i_{10} \mid \varepsilon$	P
$o_1{:}\,\mathtt{List}(\mathtt{n}{=}o_4) \quad i_9{:}\,\mathbb{Z} \quad i_{10}{:}\,\mathbb{Z}$	
$o_4{:}\,\mathtt{List}(\mathtt{n}{=}o_5) \quad o_5{:}\,\mathtt{List}(?)$	

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

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

Transforming States to Terms

o_4, i_{10} | 0 | $\texttt{t}: o_4, \texttt{i}: i_{10}$ | ε			
~~o_1~~$, i_9$ | 34 | $\texttt{t}: o_1, \texttt{i}: i_{10}$ | ε			
$o_1: \texttt{List}(\texttt{n} = o_4)$	$i_9: \mathbb{Z}$	$i_{10}: \mathbb{Z}$	
$o_4: \texttt{List}(\texttt{n} = o_5)$	$o_5: \texttt{List}(?)$		

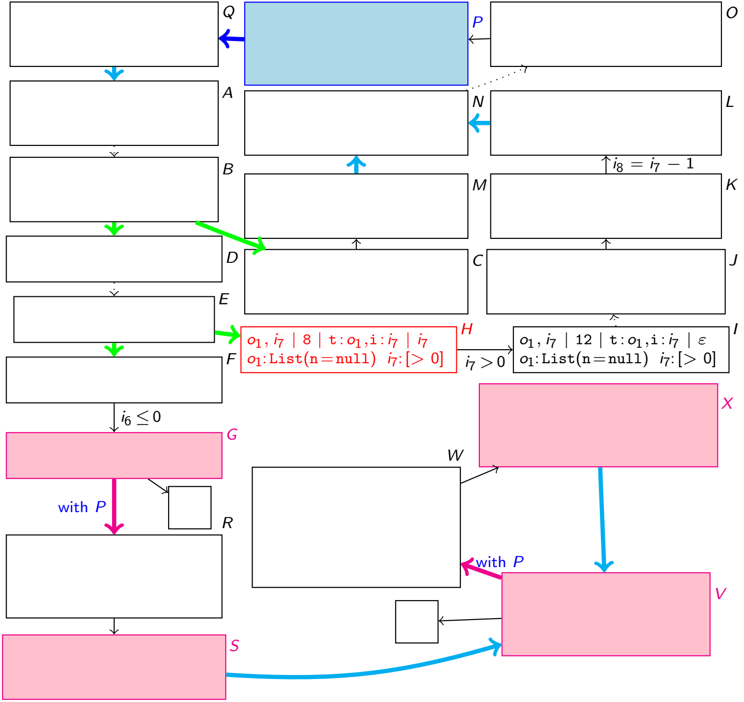
P

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

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

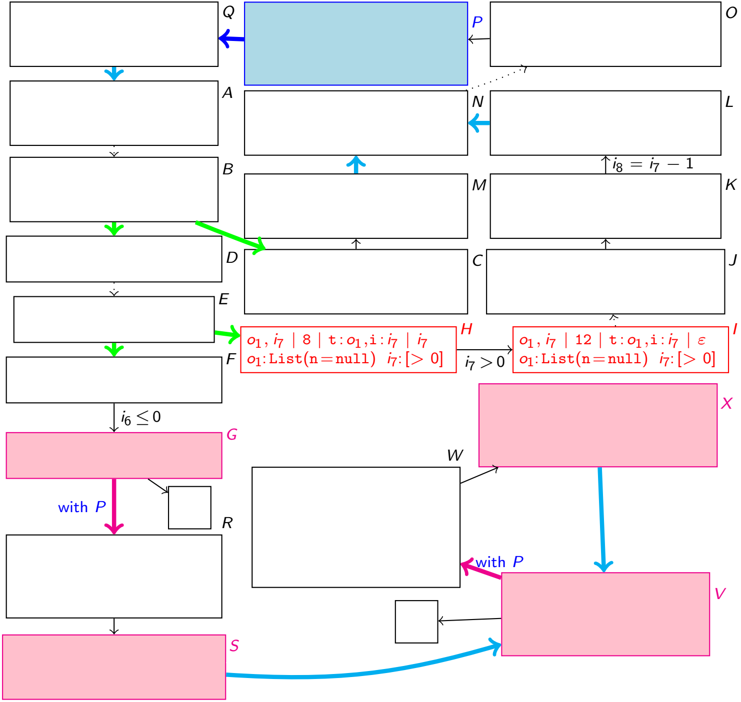
Transforming Evaluation Edges

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



Transforming Evaluation Edges

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



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

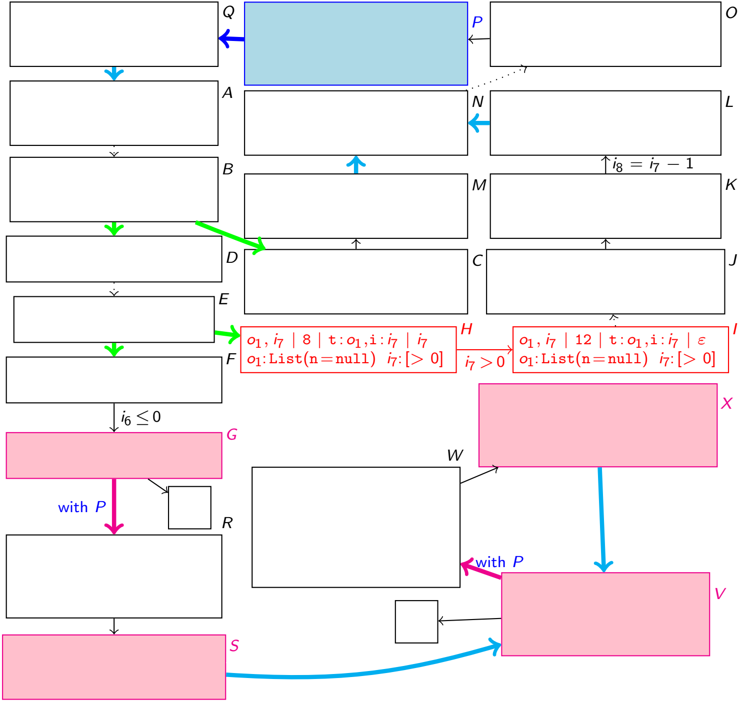
Transforming Evaluation Edges

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

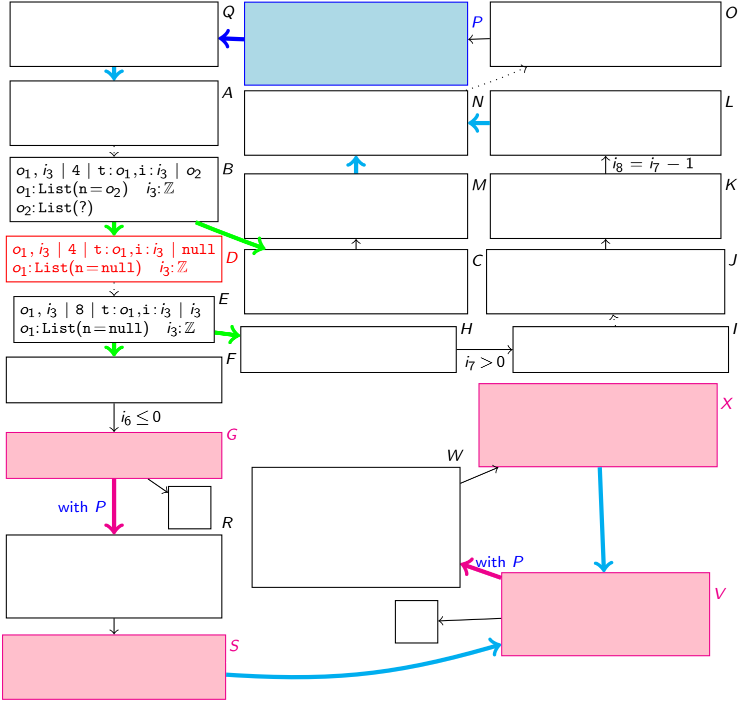
$$\rightarrow$$

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

$$\mid i_7 > 0$$



Transforming Refinement Edges

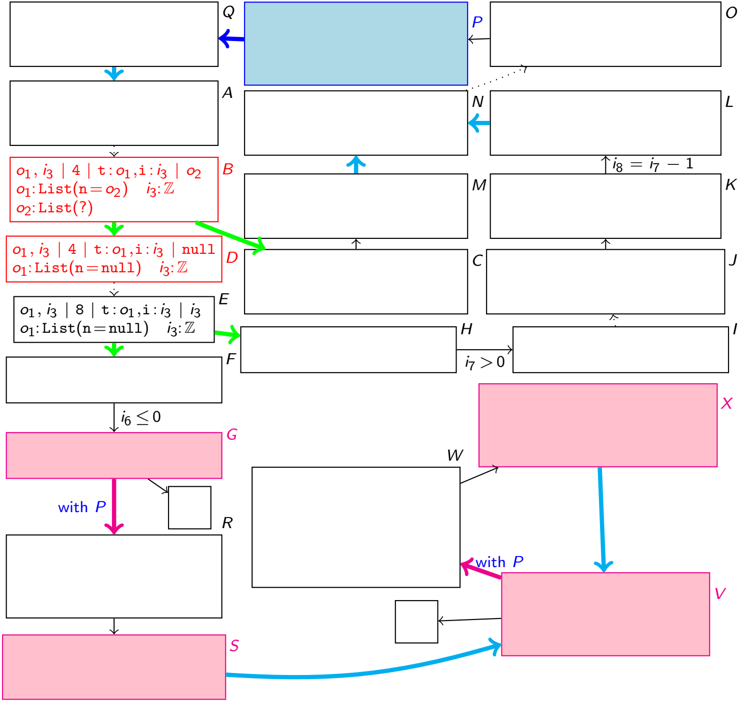


$f_{D,4}($ eos,
L(null),
i_3,
L(null),
i_3,
null)

Transforming Refinement Edges

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

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

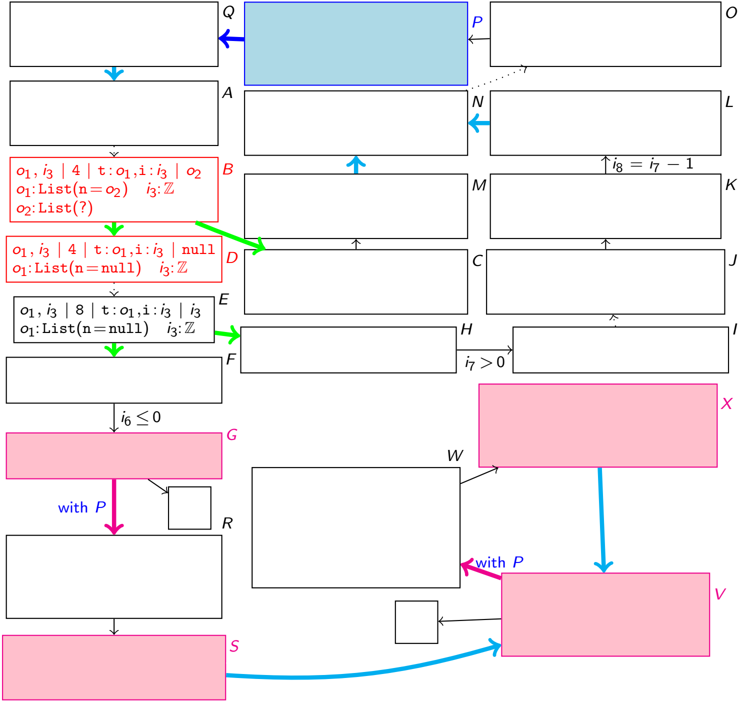


Transforming Refinement Edges

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

$\rightarrow$

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



Merging Rewrite Rules

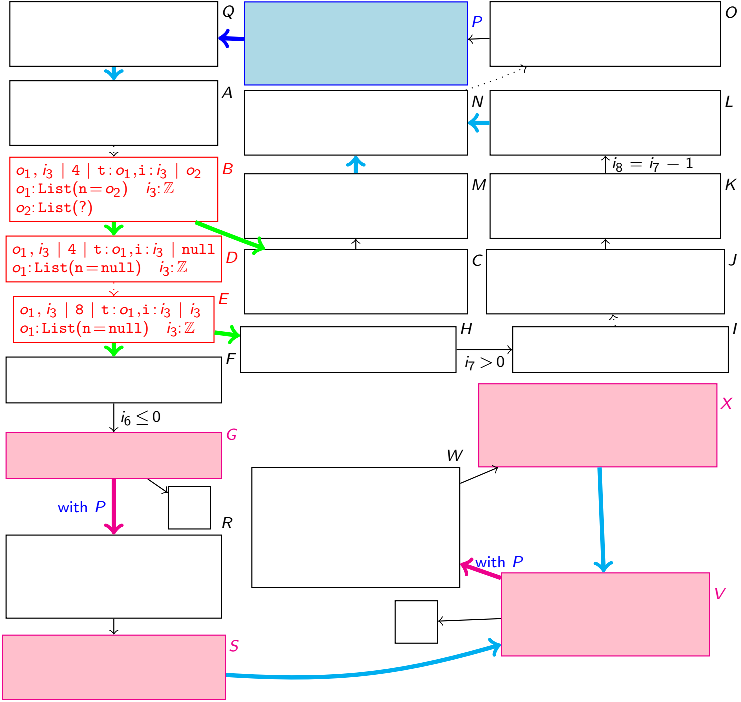
$f_{B,4}($ eos,
L(null),
i_3,
L(null),
i_3,
null)

$\rightarrow$

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

$\rightarrow$

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

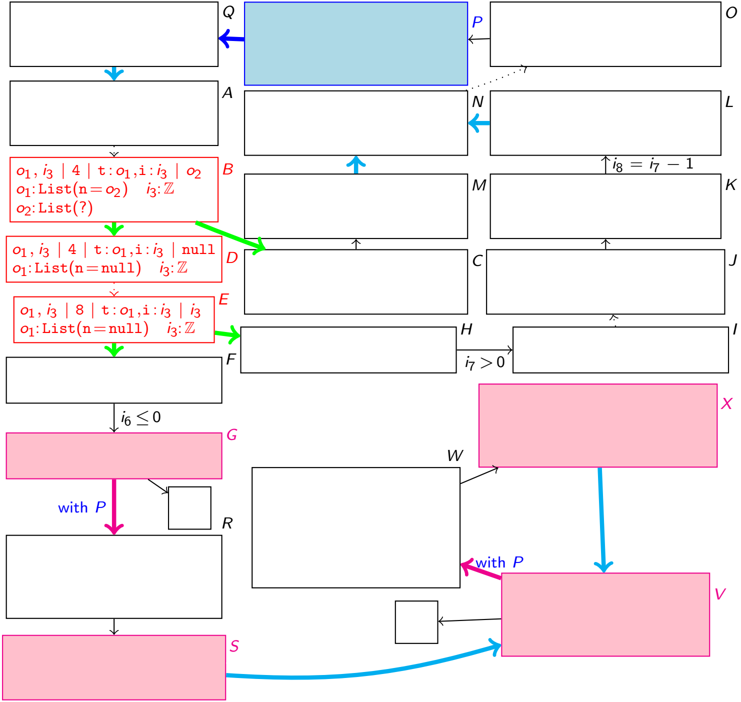


Merging Rewrite Rules

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

$$\rightarrow$$

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



$$
\begin{array}{rcll}
f_A(L(null),\ i_6) & \rightarrow & f_G(L(null),\ i_6) & |\, i_6 \leq 0 \quad (1) \\
f_A(L(null),\ i_7) & \rightarrow & f_P(\ f_A(L(null), i_7 - 1),\ L(L(null)),\ i_7) & |\, i_7 > 0 \quad (2) \\
f_A(L(L(o_5)),\ i_3) & \rightarrow & f_P(\ f_A(L(o_5), i_3),\ L(L(o_5)),\ i_3) & (3) \\
f_P(\ f_G(L(null), i_{12}),\ L(L(null)),\ i_9) & \rightarrow & f_V(L(L(null)),\ i_9) & (4) \\
f_P(\ f_V(L(L(o_{20})), i_{19}),\ L(L(o_5)),\ i_9) & \rightarrow & f_V(L(L(L(o_{20}))),\ i_9) & (5)
\end{array}
$$

TRS is natural

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

$$\mathsf{f}_A(\mathsf{L}(\mathsf{null}),\ i_6) \to \mathsf{f}_G(\mathsf{L}(\mathsf{null}),\ i_6) \quad | i_6 \leq 0 \quad (1)$$
$$\mathsf{f}_A(\mathsf{L}(\mathsf{null}),\ i_7) \to \mathsf{f}_P(\ \mathsf{f}_A(\mathsf{L}(\mathsf{null}), i_7 - 1),\ \mathsf{L}(\mathsf{L}(\mathsf{null})),\ i_7) \,|\, i_7 > 0 \quad (2)$$
$$\mathsf{f}_A(\mathsf{L}(\mathsf{L}(o_5)),\ i_3) \to \mathsf{f}_P(\ \mathsf{f}_A(\mathsf{L}(o_5), i_3),\ \mathsf{L}(\mathsf{L}(o_5)),\ i_3) \quad (3)$$
$$\mathsf{f}_P(\ \mathsf{f}_G(\mathsf{L}(\mathsf{null}), i_{12}),\ \mathsf{L}(\mathsf{L}(\mathsf{null})),\ i_9) \to \mathsf{f}_V(\mathsf{L}(\mathsf{L}(\mathsf{null})),\ i_9) \quad (4)$$
$$\mathsf{f}_P(\ \mathsf{f}_V(\mathsf{L}(\mathsf{L}(o_{20})), i_{19}),\ \mathsf{L}(\mathsf{L}(o_5)),\ i_9) \to \mathsf{f}_V(\mathsf{L}(\mathsf{L}(\mathsf{L}(o_{20}))),\ i_9) \quad (5)$$

TRS is natural

1. If `n == null` and `i <= 0`, then `return`.

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

$$
\begin{array}{rll}
f_A(L(null),\ i_6) \to f_G(L(null),\ i_6) & |\, i_6 \leq 0 & (1)\\
f_A(L(null),\ i_7) \to f_P(\ f_A(L(null), i_7 - 1),\ L(L(null)),\ i_7) & |\, i_7 > 0 & (2)\\
f_A(L(L(o_5)),\ i_3) \to f_P(\ f_A(L(o_5), i_3),\ L(L(o_5)),\ i_3) & & (3)\\
f_P(\ f_G(L(null), i_{12}),\ L(L(null)),\ i_9) \to f_V(L(L(null)),\ i_9) & & (4)\\
f_P(\ f_V(L(L(o_{20})), i_{19}),\ L(L(o_5)),\ i_9) \to f_V(L(L(L(o_{20}))),\ i_9) & & (5)
\end{array}
$$

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`.

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

$$
\begin{array}{rlr}
f_A(L(null),\ i_6) \to f_G(L(null),\ i_6) & |\, i_6 \leq 0 & (1) \\
f_A(L(null),\ i_7) \to f_P(\ f_A(L(null), i_7 - 1),\ L(L(null)),\ i_7) & |\, i_7 > 0 & (2) \\
f_A(L(L(o_5)),\ i_3) \to f_P(\ f_A(L(o_5), i_3),\ L(L(o_5)),\ i_3) & & (3) \\
f_P(\ f_G(L(null), i_{12}),\ L(L(null)),\ i_9) \to f_V(L(L(null)),\ i_9) & & (4) \\
f_P(\ f_V(L(L(o_{20})), i_{19}),\ L(L(o_5)),\ i_9) \to f_V(L(L(L(o_{20}))),\ i_9) & & (5)
\end{array}
$$

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`.

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

$$
\begin{array}{rll}
f_A(L(null),\ i_6) \to f_G(L(null),\ i_6) & |\, i_6 \leq 0 & (1) \\
f_A(L(null),\ i_7) \to f_P(\ f_A(L(null), i_7 - 1),\ L(L(null)),\ i_7) & |\, i_7 > 0 & (2) \\
f_A(L(L(o_5)),\ i_3) \to f_P(\ f_A(L(o_5), i_3),\ L(L(o_5)),\ i_3) & & (3) \\
f_P(\ f_G(L(null), i_{12}),\ L(L(null)),\ i_9) \to f_V(L(L(null)),\ i_9) & & (4) \\
f_P(\ f_V(L(L(o_{20})), i_{19}),\ L(L(o_5)),\ i_9) \to f_V(L(L(L(o_{20}))),\ i_9) & & (5)
\end{array}
$$

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);
}
```

$$
\begin{aligned}
f_A(L(\text{null}), i_6) &\to f_G(L(\text{null}), i_6) & | i_6 \le 0 \quad (1)\\
f_A(L(\text{null}), i_7) &\to f_P(f_A(L(\text{null}), i_7 - 1), L(L(\text{null})), i_7) & | i_7 > 0 \quad (2)\\
f_A(L(L(o_5)), i_3) &\to f_P(f_A(L(o_5), i_3), L(L(o_5)), i_3) & (3)\\
f_P(f_G(L(\text{null}), i_{12}), L(L(\text{null})), i_9) &\to f_V(L(L(\text{null})), i_9) & (4)\\
f_P(f_V(L(L(o_{20})), i_{19}), L(L(o_5)), i_9) &\to f_V(L(L(L(o_{20}))), i_9) & (5)
\end{aligned}
$$

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 $L(L(o_{20}))$ is written to field `n`.
   Side effect replaces $L(\underline{L(o_5)})$ by $L(\underline{L(L(o_{20}))})$.

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

$$
\begin{array}{rlr}
f_A(L(null),\ i_6) \to f_G(L(null),\ i_6) & |\, i_6 \le 0 & (1)\\
f_A(L(null),\ i_7) \to f_P(\ f_A(L(null), i_7 - 1),\ L(L(null)),\ i_7) & |\, i_7 > 0 & (2)\\
f_A(L(L(o_5)),\ i_3) \to f_P(\ f_A(L(o_5), i_3),\ L(L(o_5)),\ i_3) & & (3)\\
f_P(\ f_G(L(null), i_{12}),\ L(L(null)),\ i_9) \to f_V(L(L(null)),\ i_9) & & (4)\\
f_P(\ f_V(L(L(o_{20})), i_{19}),\ L(L(o_5)),\ i_9) \to f_V(L(L(L(o_{20}))),\ i_9) & & (5)
\end{array}
$$

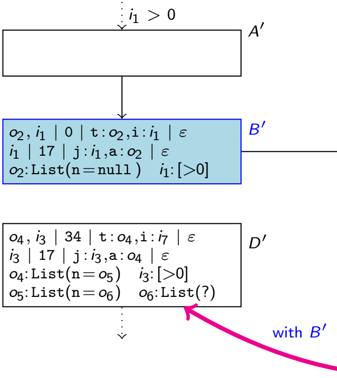
TRS is natural **and termination is easy**

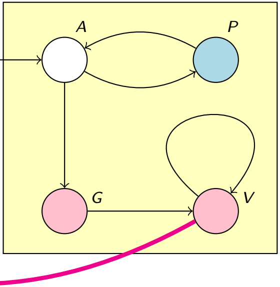
1. If `n == null` and `i <= 0`, then `return`.
2. If `n == null` and `i > 0`,
   then attach new element to list.
   Recursive call with tail of list and `i-1`.
3. If `n != null`,
   then recursive call with tail of list and `i`.
4. After recursive call, resulting list L(null) is written to field `n`.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

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

$$
\begin{array}{rcll}
f_A(L(null),\ i_6) & \to & f_G(L(null),\ i_6) & |\, i_6 \le 0 \quad (1)\\
f_A(L(null),\ i_7) & \to & f_P(\ f_A(L(null), i_7 - 1),\ L(L(null)),\ i_7) & |\, i_7 > 0 \quad (2)\\
f_A(L(L(o_5)),\ i_3) & \to & f_P(\ f_A(L(o_5), i_3),\ L(L(o_5)),\ i_3) & (3)\\
f_P(\ f_G(L(null), i_{12}),\ L(L(null)),\ i_9) & \to & f_V(L(L(null)),\ i_9) & (4)\\
f_P(\ f_V(L(L(o_{20})), i_{19}),\ L(L(o_5)),\ i_9) & \to & f_V(L(L(L(o_{20}))),\ i_9) & (5)
\end{array}
$$

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





$$\mathsf{f}_A(\mathsf{L}(\mathsf{null}),\ i_6) \to \mathsf{f}_G(\mathsf{L}(\mathsf{null}),\ i_6) \qquad | i_6 \leq 0 \qquad (1)$$

$$\mathsf{f}_A(\mathsf{L}(\mathsf{null}),\ i_7) \to \mathsf{f}_P(\ \mathsf{f}_A(\mathsf{L}(\mathsf{null}), i_7 - 1),\ \mathsf{L}(\mathsf{L}(\mathsf{null})),\ i_7) \ | i_7 > 0 \qquad (2)$$

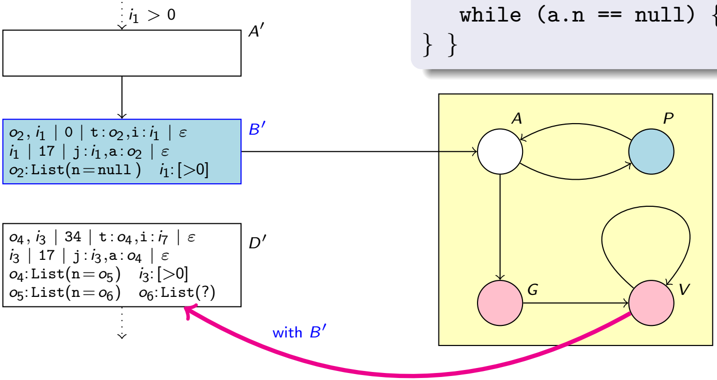
$$\mathsf{f}_A(\mathsf{L}(\mathsf{L}(o_5)),\ i_3) \to \mathsf{f}_P(\ \mathsf{f}_A(\mathsf{L}(o_5), i_3),\ \mathsf{L}(\mathsf{L}(o_5)),\ i_3) \qquad (3)$$

$$\mathsf{f}_P(\ \mathsf{f}_G(\mathsf{L}(\mathsf{null}), i_{12}),\ \mathsf{L}(\mathsf{L}(\mathsf{null})),\ i_9) \to \mathsf{f}_V(\mathsf{L}(\mathsf{L}(\mathsf{null})),\ i_9) \qquad (4)$$

$$\mathsf{f}_P(\ \mathsf{f}_V(\mathsf{L}(\mathsf{L}(o_{20})), i_{19}),\ \mathsf{L}(\mathsf{L}(o_5)),\ i_9) \to \mathsf{f}_V(\mathsf{L}(\mathsf{L}(\mathsf{L}(o_{20}))),\ i_9) \qquad (5)$$

$$\mathsf{f}_{A'}(\ldots) \to \mathsf{f}_{B'}(\mathsf{f}_A(\ldots), \ldots)$$

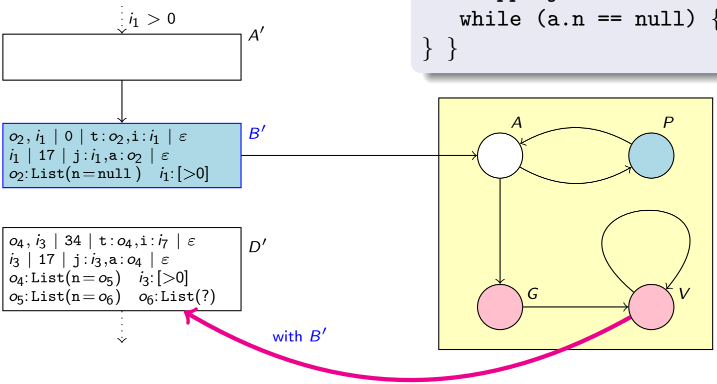
```
static void cappE(int j) {
  List a = new List();
  if (j > 0) {
   a.appE(j);
   while (a.n == null) {}
} }
```



$$
\begin{aligned}
f_A(L(\text{null}),\ i_6) &\to f_G(L(\text{null}),\ i_6) & \mid i_6 \leq 0 \quad (1)\\
f_A(L(\text{null}),\ i_7) &\to f_P(\ f_A(L(\text{null}), i_7 - 1),\ L(L(\text{null})),\ i_7) & \mid i_7 > 0 \quad (2)\\
f_A(L(L(o_5)),\ i_3) &\to f_P(\ f_A(L(o_5), i_3),\ L(L(o_5)),\ i_3) & (3)\\
f_P(\ f_G(L(\text{null}), i_{12}),\ L(L(\text{null})),\ i_9) &\to f_V(L(L(\text{null})),\ i_9) & (4)\\
f_P(\ f_V(L(L(o_{20})), i_{19}),\ L(L(o_5)),\ i_9) &\to f_V(L(L(L(o_{20}))),\ i_9) & (5)
\end{aligned}
$$

$$
\begin{aligned}
f_{A'}(\ldots) &\to f_{B'}(f_A(\ldots), \ldots)\\
f_{B'}(f_V(\ldots), \ldots) &\to f_{D'}(f_{D'}(\ldots), \ldots)
\end{aligned}
$$

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



$$
\begin{aligned}
f_A(L(\mathrm{null}),\ i_6) &\to f_G(L(\mathrm{null}),\ i_6) & |\, i_6 \le 0 \quad (1)\\
f_A(L(\mathrm{null}),\ i_7) &\to f_P(\ f_A(L(\mathrm{null}), i_7 - 1),\ L(L(\mathrm{null})),\ i_7) & |\, i_7 > 0 \quad (2)\\
f_A(L(L(o_5)),\ i_3) &\to f_P(\ f_A(L(o_5), i_3),\ L(L(o_5)),\ i_3) & (3)\\
f_P(\ f_G(L(\mathrm{null}), i_{12}),\ L(L(\mathrm{null})),\ i_9) &\to f_V(L(L(\mathrm{null})),\ i_9) & (4)\\
f_P(\ f_V(L(L(o_{20})), i_{19}),\ L(L(o_5)),\ i_9) &\to f_V(L(L(L(o_{20}))),\ i_9) & (5)
\end{aligned}
$$

$$
\begin{aligned}
f_{A'}(\ldots) &\to f_{B'}(f_A(\ldots), \ldots)\\
f_{B'}(f_V(\ldots), \ldots) &\to f_{D'}(f_{D'}(\ldots), \ldots)
\end{aligned}
$$

```
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

$$
\begin{array}{rlr}
f_A(\mathsf{L}(\mathsf{null}),\, i_6) \to f_G(\mathsf{L}(\mathsf{null}),\, i_6) & \mid i_6 \leq 0 & (1)\\
f_A(\mathsf{L}(\mathsf{null}),\, i_7) \to f_P(\, f_A(\mathsf{L}(\mathsf{null}), i_7 - 1),\, \mathsf{L}(\mathsf{L}(\mathsf{null})),\, i_7) & \mid i_7 > 0 & (2)\\
f_A(\mathsf{L}(\mathsf{L}(o_5)),\, i_3) \to f_P(\, f_A(\mathsf{L}(o_5), i_3),\, \mathsf{L}(\mathsf{L}(o_5)),\, i_3) & & (3)\\
f_P(\, f_G(\mathsf{L}(\mathsf{null}), i_{12}),\, \mathsf{L}(\mathsf{L}(\mathsf{null})),\, i_9) \to f_V(\mathsf{L}(\mathsf{L}(\mathsf{null})),\, i_9) & & (4)\\
f_P(\, f_V(\mathsf{L}(\mathsf{L}(o_{20})), i_{19}),\, \mathsf{L}(\mathsf{L}(o_5)),\, i_9) \to f_V(\mathsf{L}(\mathsf{L}(\mathsf{L}(o_{20}))),\, i_9) & & (5)
\end{array}
$$

$$
\begin{array}{rl}
f_{A'}(\ldots) & \to f_{B'}(f_A(\ldots), \ldots)\\
f_{B'}(f_V(\ldots), \ldots) & \to f_{D'}(f_{D'}(\ldots), \ldots)
\end{array}
$$

```
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

  $\Rightarrow$ termination of `cappE` follows from termination of `appE`

$$
\begin{array}{rll}
f_A(L(null),\ i_6) \to f_G(L(null),\ i_6) & |\, i_6 \leq 0 & (1)\\
f_A(L(null),\ i_7) \to f_P(\ f_A(L(null), i_7 - 1),\ L(L(null)),\ i_7) & |\, i_7 > 0 & (2)\\
f_A(L(L(o_5)),\ i_3) \to f_P(\ f_A(L(o_5), i_3),\ L(L(o_5)),\ i_3) & & (3)\\
f_P(\ f_G(L(null), i_{12}),\ L(L(null)),\ i_9) \to f_V(L(L(null)),\ i_9) & & (4)\\
f_P(\ f_V(L(L(o_{20})), i_{19}),\ L(L(o_5)),\ i_9) \to f_V(L(L(L(o_{20}))),\ i_9) & & (5)
\end{array}
$$

$$
\begin{array}{rl}
f_{A'}(\ldots) & \to f_{B'}(f_A(\ldots),\ldots)\\
f_{B'}(f_V(\ldots),\ldots) & \to f_{D'}(f_{D'}(\ldots),\ldots)
\end{array}
$$

```
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

  $\Rightarrow$ 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

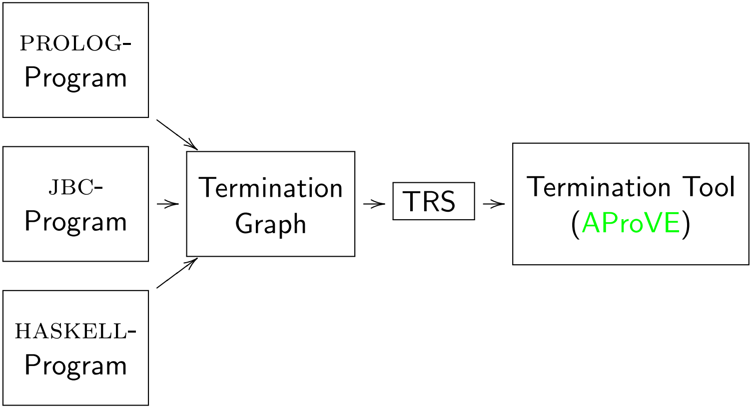
Theorem 2

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