Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting

M. Brockschmidt, C. Otto, J. Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

RTA 2011, Novi Sad

Termination Analysis for Imperative Programs

Termination Analysis for Imperative Programs

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

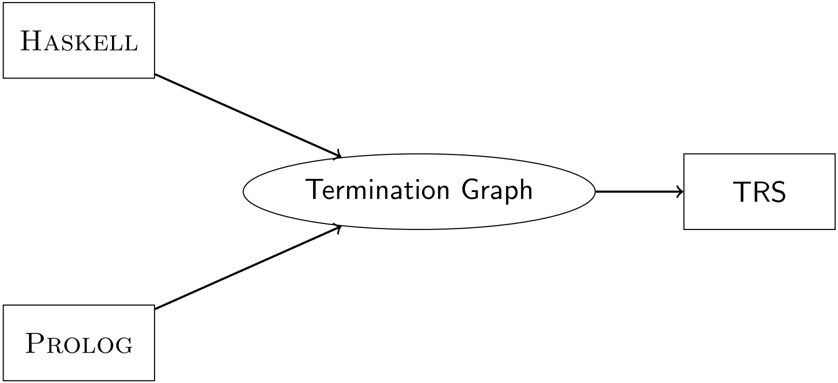
Termination Analysis for Imperative Programs

- 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)

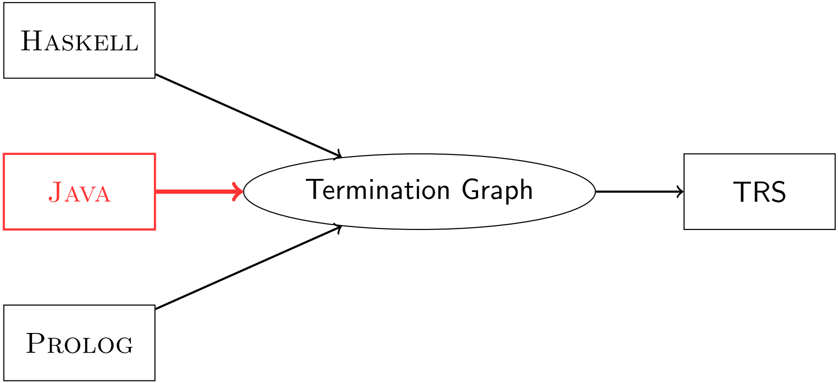
Termination Analysis for Imperative Programs

- 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 (JBC)
  Fixed abstraction, via Constraint Logic Programs
  (Spoto, Mesnard, Payet, 10)
  (Albert, Arenas, Codish, Genaim, Puebla, Zanardini, 08)

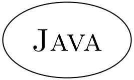
Rewriting-based approaches



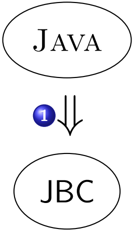
Rewriting-based approaches



Rewriting-based approach: Structure

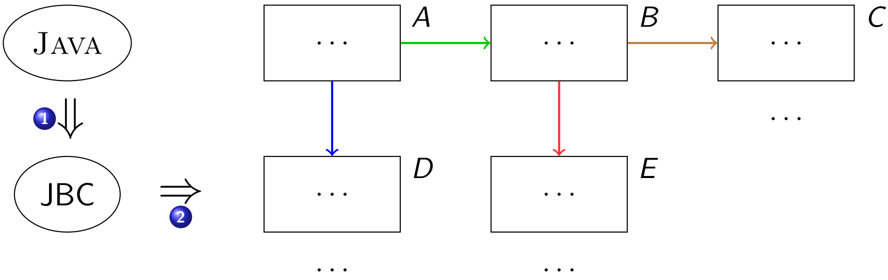


Rewriting-based approach: Structure



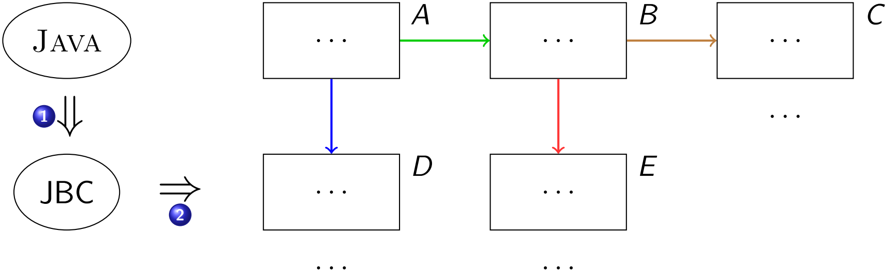
1. Sun/Oracle `javac`

Rewriting-based approach: Structure



1. Sun/Oracle `javac`
2. Contribution 1

Rewriting-based approach: Structure



1. Sun/Oracle `javac`
2. Contribution 1
3. Contribution 2

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

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

Rewriting-based approach: Advantages

Handling of user-defined data structures:

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

Rewriting-based approach: Advantages

Handling of user-defined data structures:

- Other techniques:
  Fixed abstraction to **number**
- List [2, 4, 6] abstracted to
  length 3

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

Rewriting-based approach: Advantages

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

Handling of user-defined data structures:

- Other techniques:
  Fixed abstraction to **number**
- List [2, 4, 6] abstracted to
  length 3
- Our technique:
  Abstraction to **terms**
- List [2, 4, 6] becomes
  List(2, List(4, List(6, null)))

Rewriting-based approach: Advantages

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

Handling of user-defined data structures:

- Other techniques:
  Fixed abstraction to **number**
- List [2, 4, 6] abstracted to
  length 3
- Our technique:
  Abstraction to **terms**
- List [2, 4, 6] becomes
  List(2, List(4, List(6, null)))

- **TRS techniques** search for suitable orders automatically

⇒ Complex orders for user-defined data structures possible

Rewriting-based approach: Challenges

- JAVA's syntactic sugar
  $\Rightarrow$ **Compile to Java Bytecode**, analyze that

Rewriting-based approach: Challenges

- JAVA's syntactic sugar
  $\Rightarrow$ **Compile to Java Bytecode**, analyze that
- Built-in data types (i.e., Integers)
  $\Rightarrow$ Use **Integer TRSs** (RTA '09)

Rewriting-based approach: Challenges

- JAVA's syntactic sugar
  $\Rightarrow$ **Compile to Java Bytecode**, analyze that
- Built-in data types (i.e., Integers)
  $\Rightarrow$ Use **Integer TRSs** (RTA '09)
- Object-oriented programming
  $\Rightarrow$ Handle in JBC-aware **graph construction**

Rewriting-based approach: Challenges

- JAVA's syntactic sugar
  $\Rightarrow$ **Compile to Java Bytecode**, analyze that
- Built-in data types (i.e., Integers)
  $\Rightarrow$ Use **Integer TRSs** (RTA '09)
- Object-oriented programming
  $\Rightarrow$ Handle in JBC-aware **graph construction**
- Aliasing, sharing, cyclicity, side-effects
  $\Rightarrow$ **Heap annotations** in **graph construction** (RTA '10)

Rewriting-based approach: Challenges

- JAVA's syntactic sugar
  ⇒ **Compile to Java Bytecode**, analyze that
- Built-in data types (i.e., Integers)
  ⇒ Use **Integer TRSs** (RTA '09)
- Object-oriented programming
  ⇒ Handle in JBC-aware **graph construction**
- Aliasing, sharing, cyclicity, side-effects
  ⇒ **Heap annotations** in **graph construction** (RTA '10)

Improvement over RTA '10:

- Recursion: Implies unbounded number of variables
- Modularity: Reusing termination proofs: How to handle side-effects?

The example

```
class List {
  List n;

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

1. Run to end of list.
2. Append `i` elements.

The 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);
  }
}
```

1. Run to end of list.
2. Append `i` elements.

Abstract JAVA virtual machine states

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

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

Abstract Java virtual machine states

```
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:

Abstract JAVA virtual machine states

```
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:

- Input arguments

Abstract JAVA virtual machine states

```
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)
```

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

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

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

stack frame:

- Input arguments
- Next program instruction

Abstract JAVA virtual machine states

```
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)
```

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

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

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

stack frame:

- Input arguments
- Next program instruction
- Local variables
- Operand stack

Abstract JAVA virtual machine states

```
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:

- Input arguments
- Next program instruction
- Local variables
- Operand stack

heap information:

Abstract JAVA virtual machine states

```
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:

- Input arguments
- Next program instruction
- Local variables
- Operand stack

heap information:

- at o_1 is List object field n points to o_2

Abstract JAVA virtual machine states

```
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:

- Input arguments
- Next program instruction
- Local variables
- Operand stack

heap information:

- at o_1 is List object field `n` points to o_2
- at o_2 is List object or `null`

Abstract JAVA virtual machine states

```
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($\texttt{n} = o_2$) i_3: $\mathbb{Z}$

o_2: List(?)

stack frame:

- Input arguments
- Next program instruction
- Local variables
- Operand stack

heap information:

- at o_1 is List object
  field `n` points to o_2
- at o_2 is List object or `null`
- at i_3 is unknown integer

Abstract JAVA virtual machine states

```
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)
```

Only explicit sharing

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

stack frame:

- Input arguments
- Next program instruction
- Local variables
- Operand stack

heap information:

- at o_1 is List object
  field n points to o_2
- at o_2 is List object or null
- at i_3 is unknown 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
```

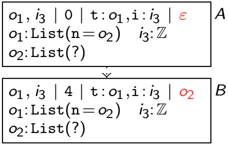
$o_1, i_3 \mid 0 \mid \mathtt{t}: o_1, \mathtt{i}: i_3 \mid \varepsilon$
$o_1: \mathtt{List}(\mathtt{n} = o_2) \quad i_3: \mathbb{Z}$
$o_2: \mathtt{List}(?)$

A

State A:

- Do all calls of appE terminate?
- this is some acyclic list
- i some integer

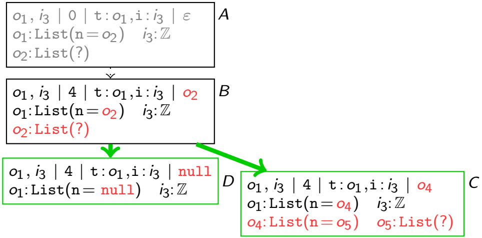
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



State *B*:

- aload_0 loads o_1 from this to opstack
- getfield n loads field n of o_1 to opstack
- *Evaluation* from *A* to *B*

```
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
```



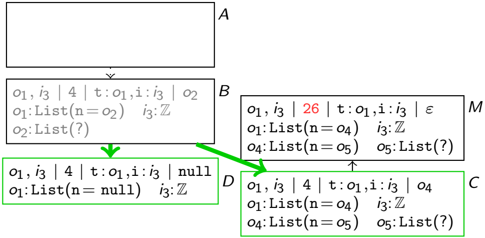
States C, D:

- `ifnonnull` branches depending on nullness of o_2
- Nullness of o_2 not known

$\Rightarrow$ Refine information:

- In C o_1 replaced by $o_4 : \texttt{List}(\texttt{n} = o_5)$
- In D o_2 replaced by `null`

```
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
```



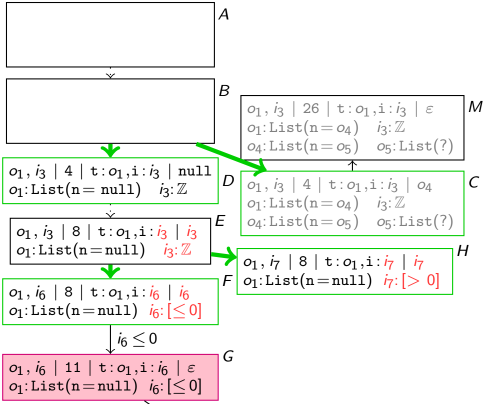
States C, D, E:

- `ifnonnull` branches depending on nullness of o_2
- Nullness of o_2 not known

$\Rightarrow$ Refine information:

- In C o_1 replaced by $o_4 : \texttt{List}(\texttt{n} = o_5)$, *evaluate to M*
- In D o_2 replaced by `null`

```
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
```



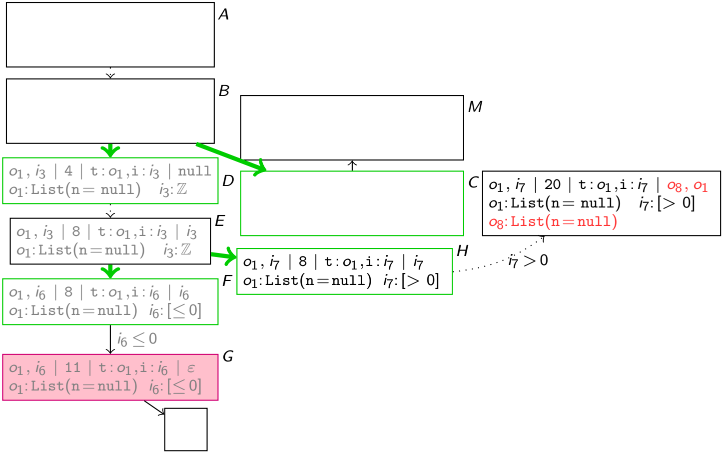
States *E*, *F*, *G*, *H*:

- *D* evaluates to *E*
- ifgt branches depending on relation of $i_3 > 0$

⇒ Refine information about i_3 (F, H)

- *Evaluation* to return state *G*

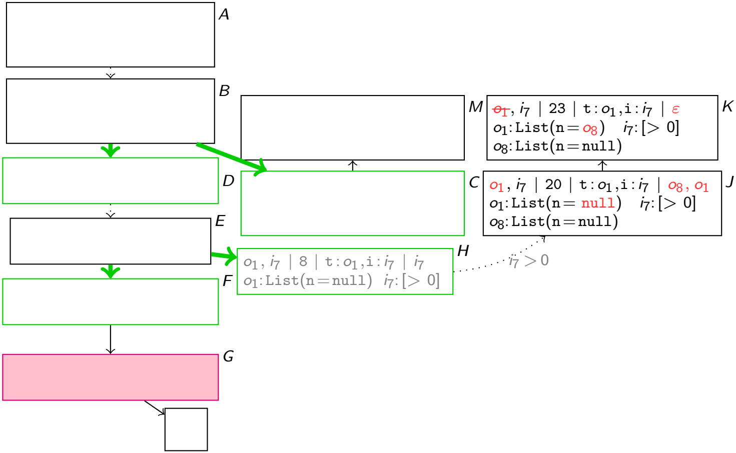
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



State *J*:

- Created new (empty) List object

```
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
```

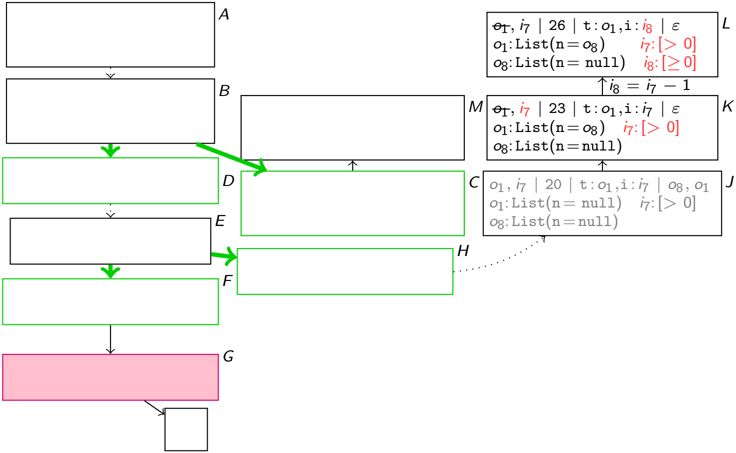


State K:

- putfield n changes field n of o_1
- *side effect* on input argument o_1

  ($\not{o_1}$ marks this)

```
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
```

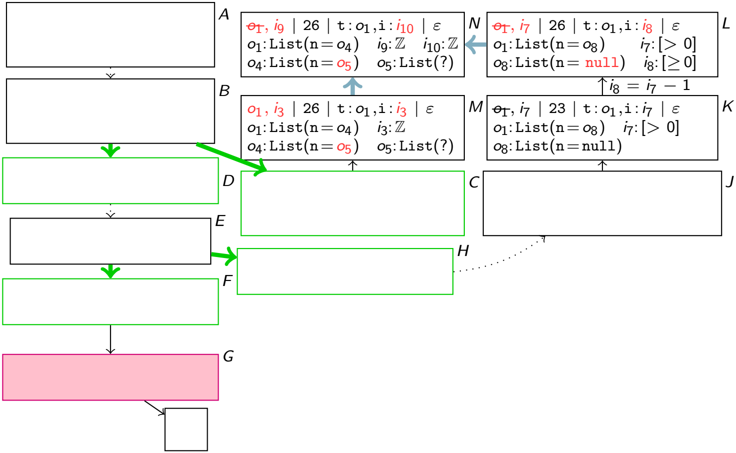


State L:

- Decrement i_7 by 1
- *No sharing* for primitive types!

  (copy on write to new value i_8)

```
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
```

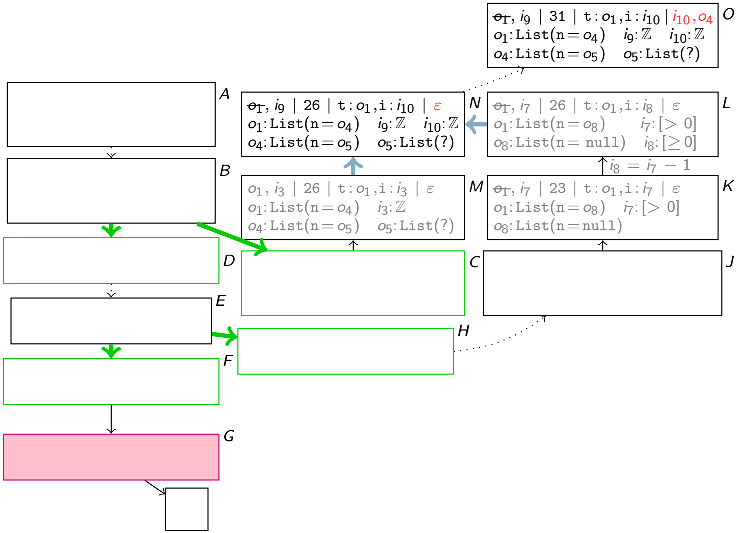


State N:

- States L and M very similar

$\Rightarrow$ N represents both ("L, M are instances of N")

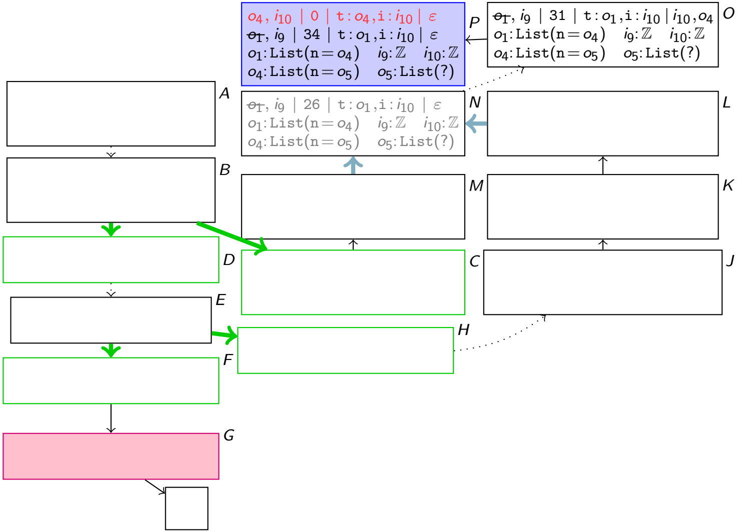
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



State *O*:

- Load `i` and `this.n` to opstack

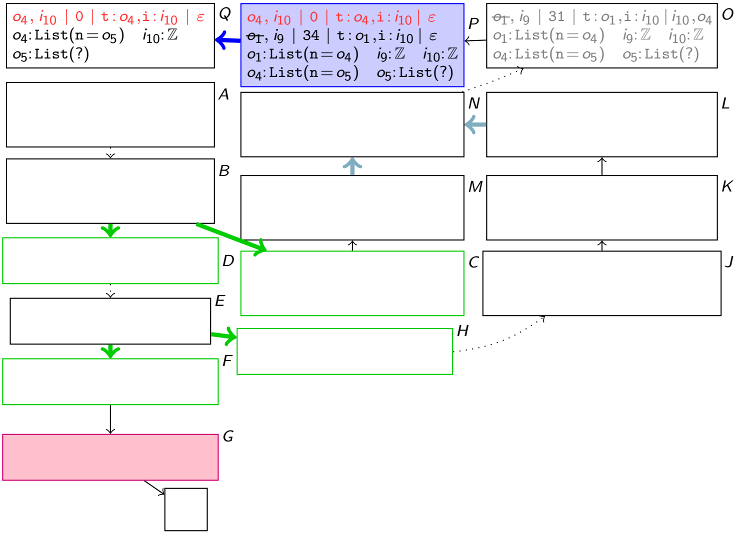
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



State *P*:

- Recursive call to appE
- New stackframe on top at position 0
- *P* call state

```
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
```

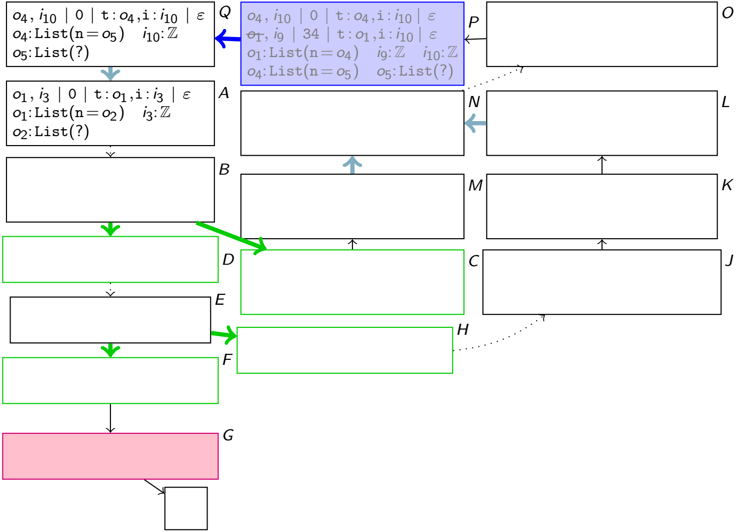


State Q:

- Avoid unbounded call stack growth

$\Rightarrow$ Split call stack (leave only top frame)

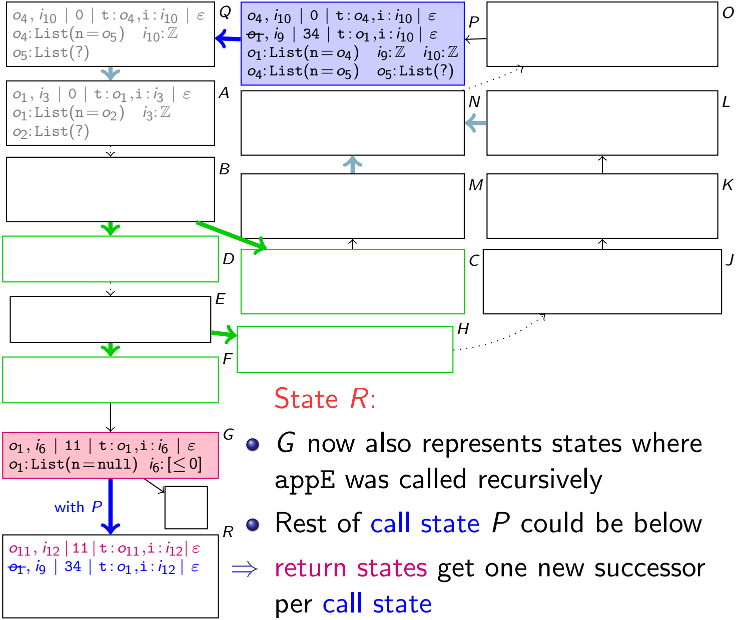
```
00: aload_0
01: getfield n
04: ifnonnull 26
07: iload_1
08: ifgt 12
11: return
12: aload_0
13: new List
16: dup
17: invoke <init>
20: putfield n
23: iinc 1, -1
26: aload_0
27: getfield n
30: iload_1
31: invoke appE
34: return
```



- Q renaming of A

$\Rightarrow$ Q instance of A

```
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
```

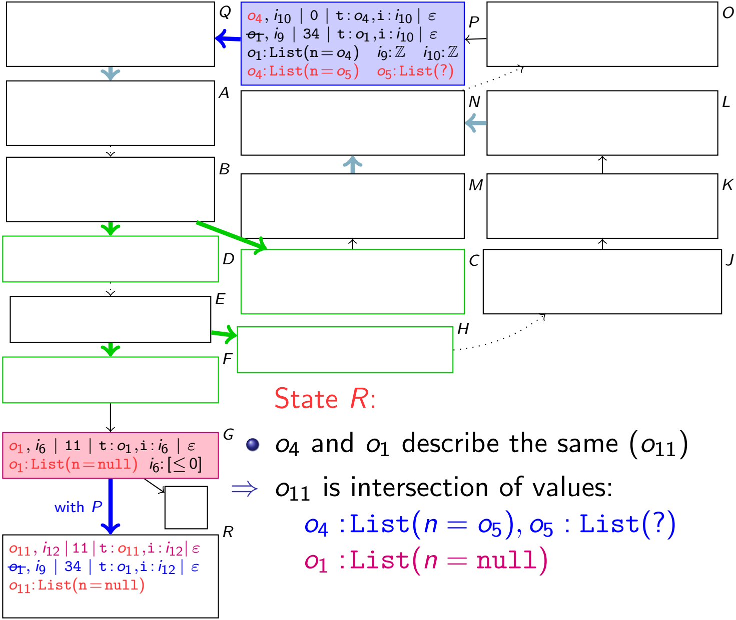


State *R*:

- *G* now also represents states where appE was called recursively
- Rest of call state *P* could be below

⇒ return states get one new successor per call state

```
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:

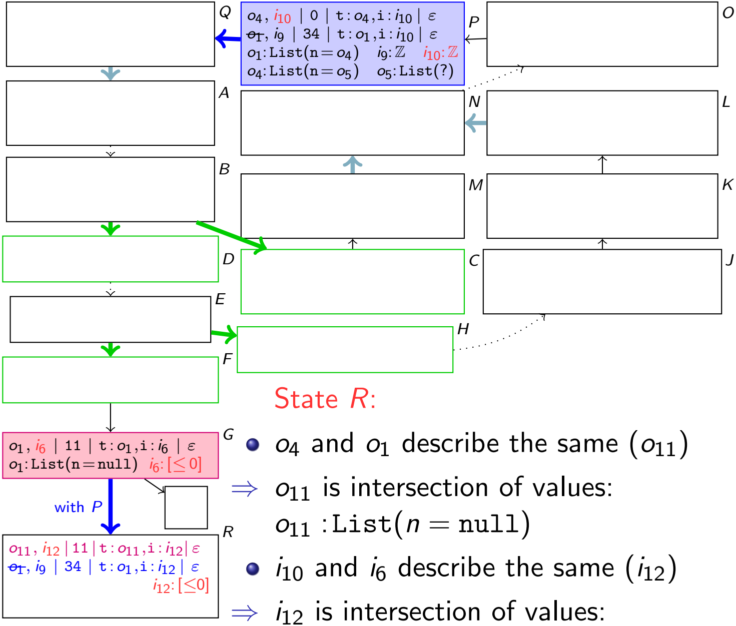
- o_4 and o_1 describe the same (o_{11})

⇒ o_{11} is intersection of values:

o_4 : List($n = o_5$), o_5 : List(?)

o_1 : List(n = null)

```
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:

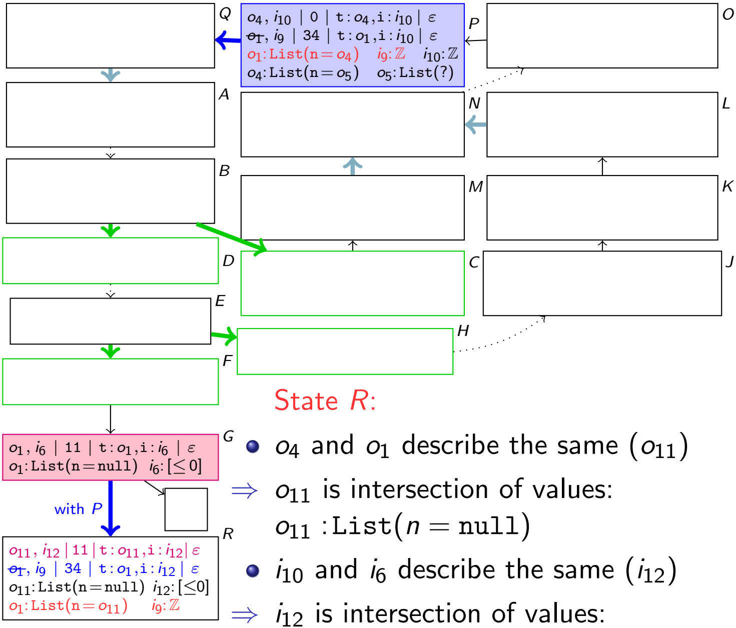
- o_4 and o_1 describe the same (o_{11})

$\Rightarrow$ o_{11} is intersection of values:
o_{11} : List(n = null)

- i_{10} and i_6 describe the same (i_{12})

$\Rightarrow$ i_{12} is intersection of values:
i_{10} : $\mathbb{Z}$
i_6 : $[\leq 0]$

```
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_4 and o_1 describe the same (o_{11})

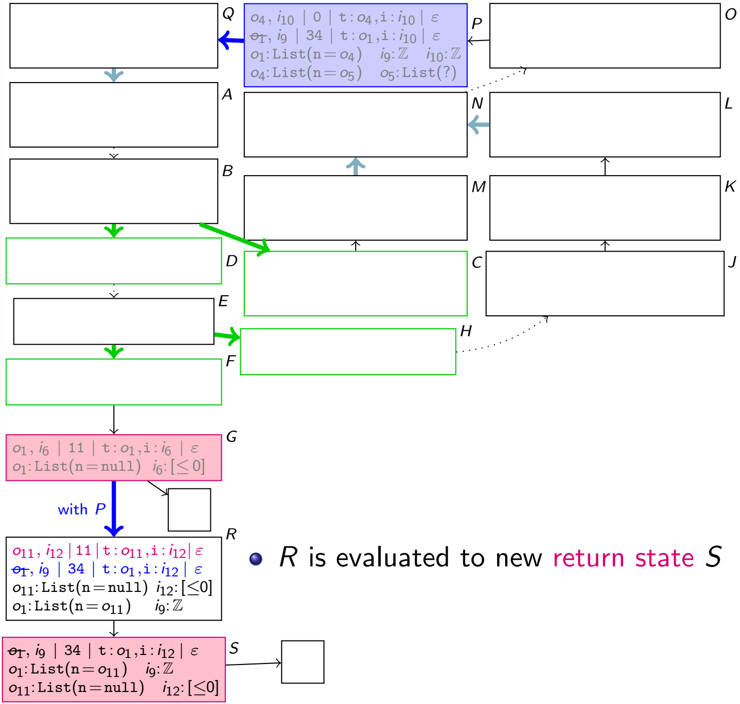
$\Rightarrow$ o_{11} is intersection of values:
o_{11} : List(n = null)

- i_{10} and i_6 describe the same (i_{12})

$\Rightarrow$ i_{12} is intersection of values:
i_{12} : [$\leq$ 0]

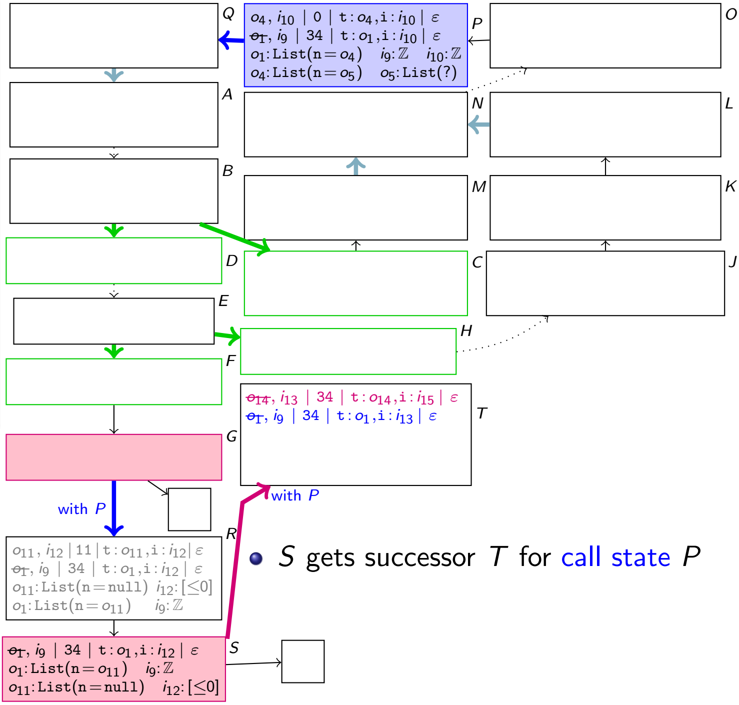
- Other values are copied

```
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
```



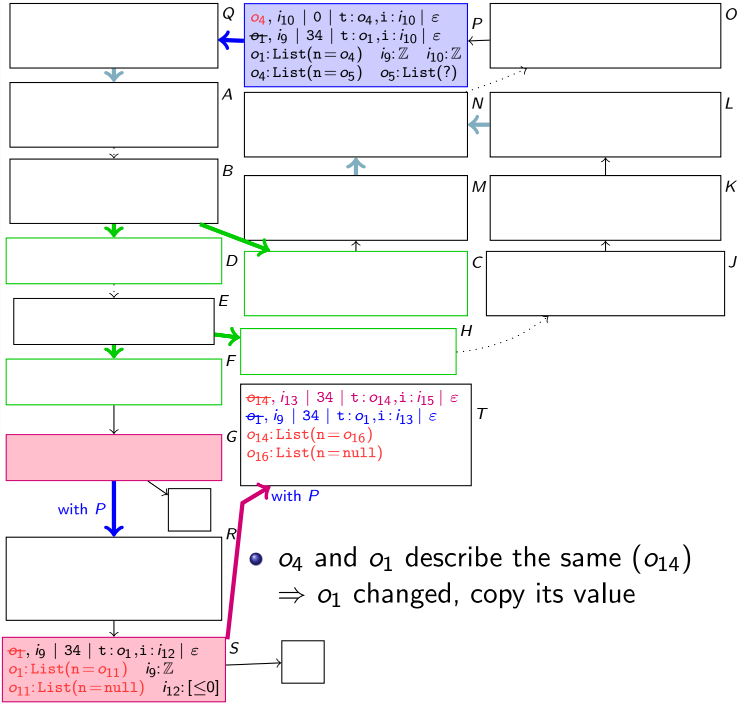
- *R* is evaluated to new return state *S*

```
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
```



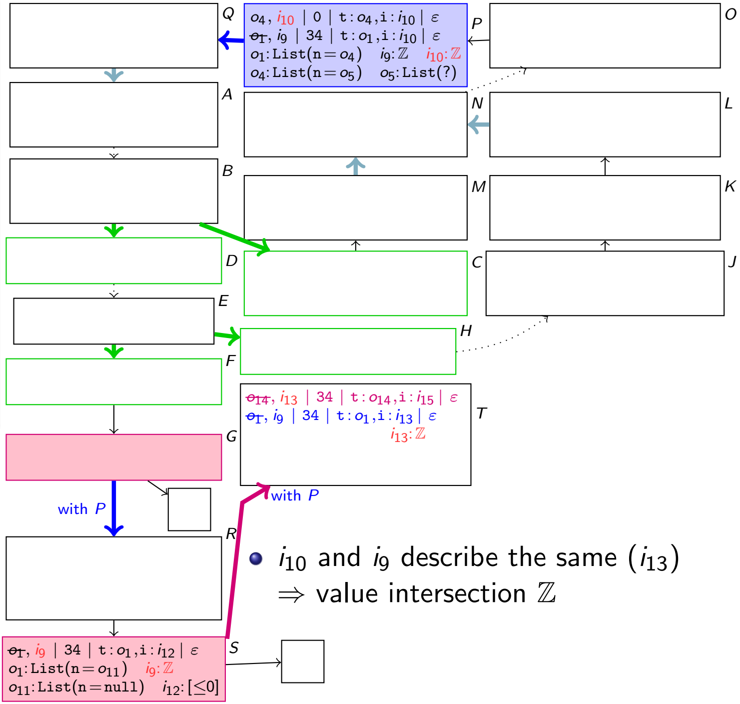
- S gets successor T for call state P

```
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
```



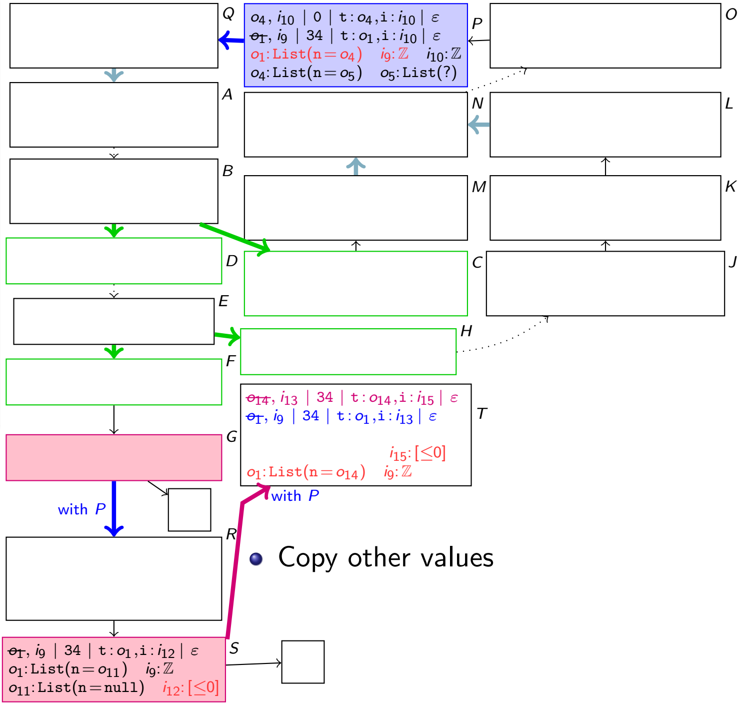
- o_4 and o_1 describe the same (o_{14})
  $\Rightarrow$ o_1 changed, copy its value

```
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
```



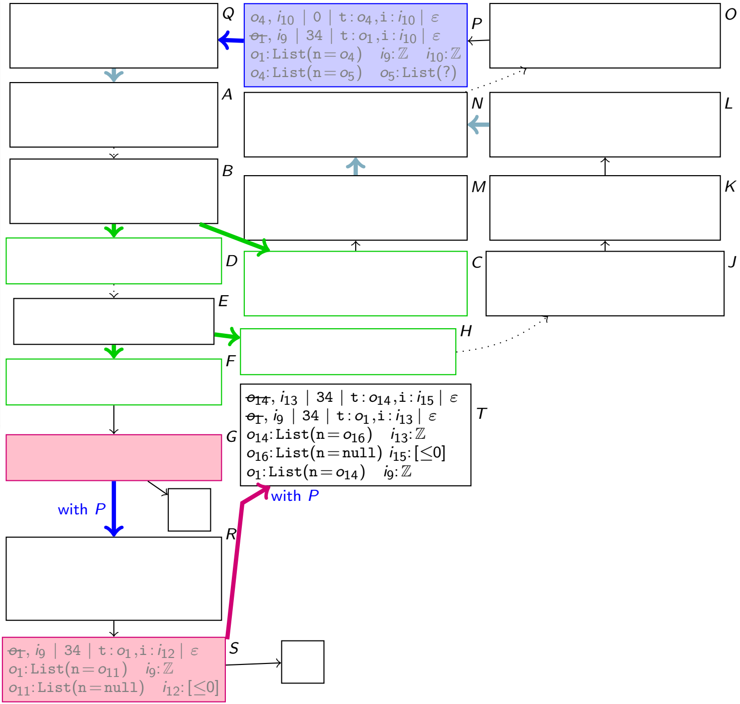
- i_{10} and i_9 describe the same (i_{13}) $\Rightarrow$ value intersection $\mathbb{Z}$

```
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
```

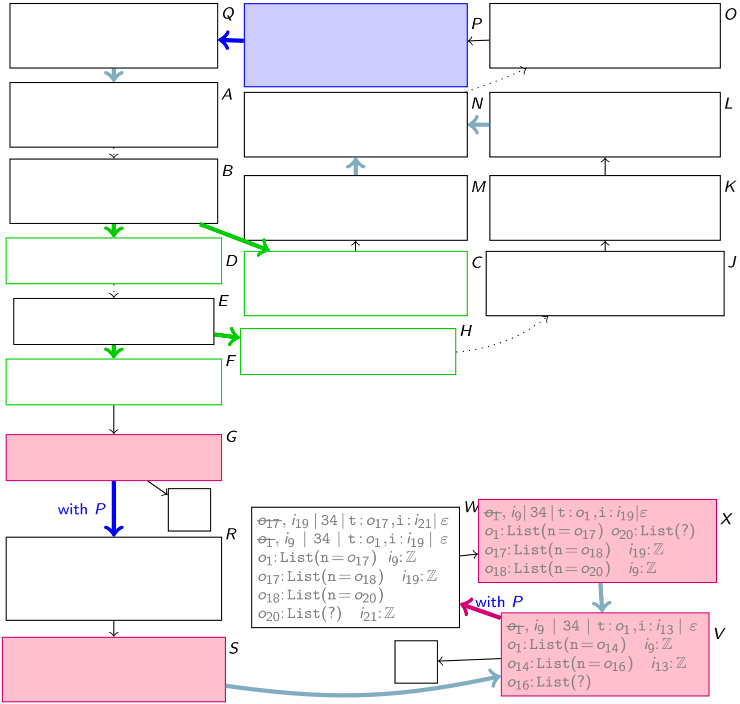


- Copy other values

```
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
```



Termination graphs

- Termination Graph from symbolic evaluation
- Repeating states are *generalized*

$\Rightarrow$ Graphs finite

Termination graphs

- Termination Graph from symbolic evaluation
- Repeating states are *generalized*

⇒ Graphs finite

- Reusal of Termination Graphs possible.

Termination graphs of several methods

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

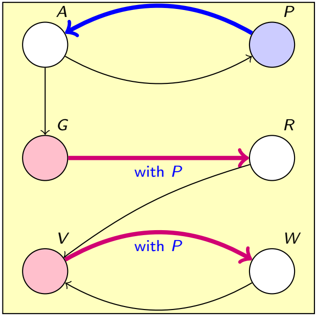
```
static void cappE(int j) {
  List a = new List();
  if (j > 0) {
   a.appE(j);
   while (a.n == null) {}
} }
```

- `cappE` creates a new `List a`
- If $j > 0$, j elements are appended
- Enters infinite loop if `a.n` is `null` afterwards

Termination graphs of several methods

```
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) {}
} }
```

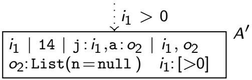


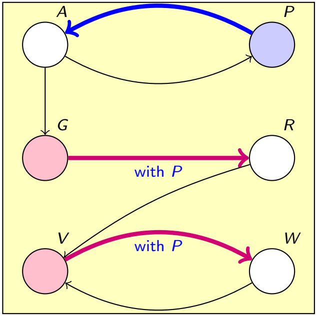
- cappE creates a new List a
- If $j > 0$, j elements are appended
- Enters infinite loop if a.n is null afterwards

Termination graphs of several methods

```
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) {}
} }
```



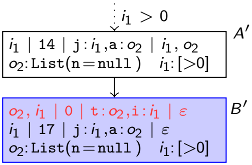


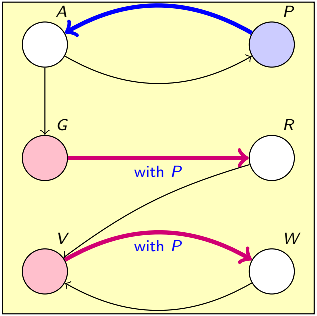
- cappE creates a new List a
- If $j > 0$, j elements are appended
- Enters infinite loop if a.n is null afterwards

Termination graphs of several methods

```
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) {}
} }
```



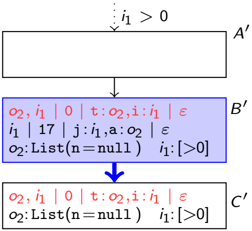


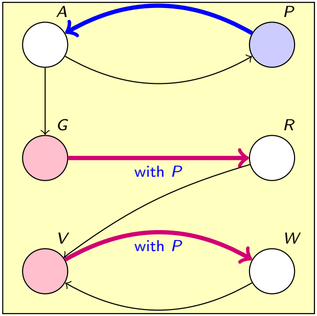
- State B' call state

Termination graphs of several methods

```
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) {}
} }
```



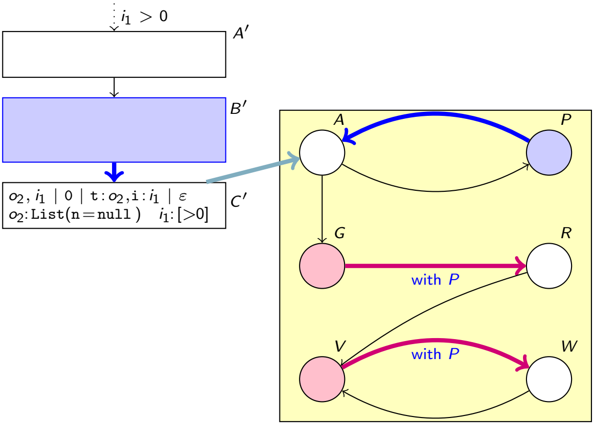


- State B' call state
- State C' call stack split result

Termination graphs of several methods

```
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) {}
} }
```

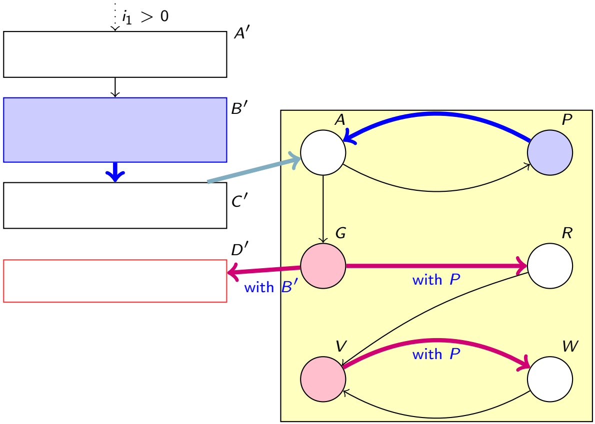


- State B' call state
- State C' call stack split result
- C' instance of A from appE

Termination graphs of several methods

```
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) {}
} }
```



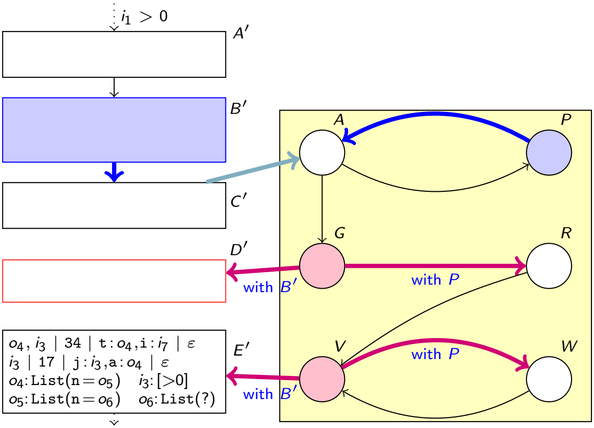
New successors for return states G, V:

- Successor D' of G does not exist:
  B' has $\texttt{i} > 0$, G has $\texttt{i} \leq 0$, intersection empty

Termination graphs of several methods

```
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) {}
} }
```



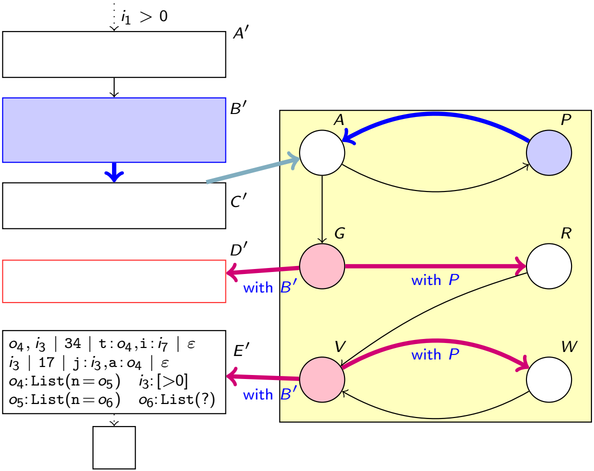
New successors for return states G, V:

- Successor D' of G does not exist
- Successor E' of V exists, a has length ≥ 1

Termination graphs of several methods

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

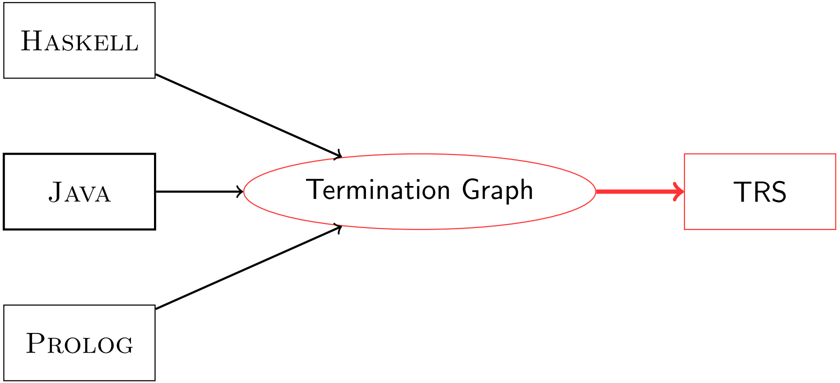
```
static void cappE(int j) {
  List a = new List();
  if (j > 0) {
   a.appE(j);
   while (a.n == null) {}
} }
```



New successors for return states G, V:

- Successor D' of G does not exist
- Successor E' of V exists, a has length ≥ 1
- Program terminates

From Termination Graphs to TRSs



Transforming values to terms

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

P

- Data structures transformed to nested constructor terms

$$\mathsf{List}(\,\mathsf{List}(o_5))$$

Transforming values to terms

$o_4, i_{10} \mid 0 \mid \texttt{t}: o_4, \texttt{i}: i_{10} \mid \varepsilon$

$\not{o_1}, i_9 \mid 34 \mid \texttt{t}: o_1, \texttt{i}: i_{10} \mid \varepsilon$

$o_1{:}\texttt{List}(\texttt{n}{=}o_4) \quad i_9{:}\mathbb{Z} \quad i_{10}{:}\mathbb{Z}$

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

P

- Data structures transformed to nested constructor terms
- Unknown values transformed to variable

$\text{List}(\text{List}(o_5))$

Transforming values to terms

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

- Data structures transformed to nested constructor terms
- Unknown values transformed to variable
- For each class `C` with n fields
  introduce one function symbol C of arity n

$$\mathrm{List}(\underbrace{\mathrm{List}(o_5)}_{o_4})$$

Transforming values to terms

$o_4, i_{10} \mid 0 \mid \texttt{t}: o_4, \texttt{i}: i_{10} \mid \varepsilon$

~~o_1~~$, i_9 \mid 34 \mid \texttt{t}: o_1, \texttt{i}: i_{10} \mid \varepsilon$

$o_1{:}\texttt{List}(\texttt{n} = o_4) \quad i_9{:}\mathbb{Z} \quad i_{10}{:}\mathbb{Z}$

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

P

- Data structures transformed to nested constructor terms
- Unknown values transformed to variable
- For each class `C` with n fields
  introduce one function symbol C of arity n
- Encode field values recursively

$$\underbrace{\mathsf{List}(\underbrace{\mathsf{List}(o_5)}_{o_4})}_{o_1}$$

Transforming states to terms

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

- For stack frame at position pp in s, introduce $\mathsf{f}_{s,pp}$

$$\mathsf{f}_{P,34}(\mathsf{f}_{P,0}(\mathrm{eos}, \mathrm{List}(o_5), i_{10}, \mathrm{List}(o_5), i_{10}), \mathrm{List}(\mathrm{List}(o_5)), i_9, \mathrm{List}(\mathrm{List}(o_5)), i_{10})$$

Transforming states to terms

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

- For stack frame at position pp in s, introduce $\mathsf{f}_{s,pp}$
- Nest terms for stack frames (topmost is innermost)
  Topmost has symbol eos to mark **e**nd **o**f **s**tack

$$\mathsf{f}_{P,34}(\mathsf{f}_{P,0}(\mathsf{eos}, \mathsf{List}(o_5), i_{10}, \mathsf{List}(o_5), i_{10}), \mathsf{List}(\mathsf{List}(o_5)), i_9, \mathsf{List}(\mathsf{List}(o_5)), i_{10})$$

Transforming states to terms

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

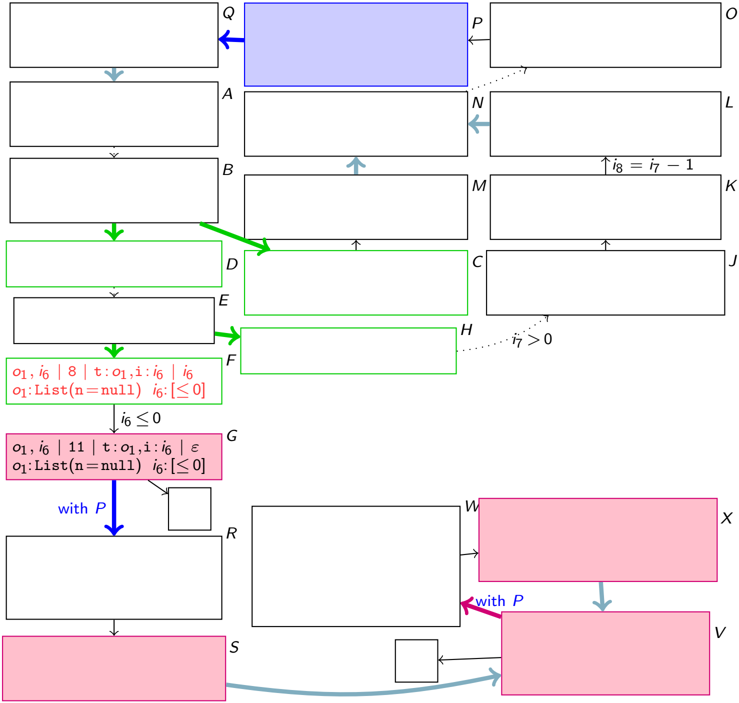
P

- For stack frame at position pp in s, introduce $\mathsf{f}_{s,pp}$
- Nest terms for stack frames (topmost is innermost)
  Topmost has symbol eos to mark **e**nd **o**f **s**tack
- Each input argument, local variable and opstack entry is encoded:

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

Converting Evaluation edges

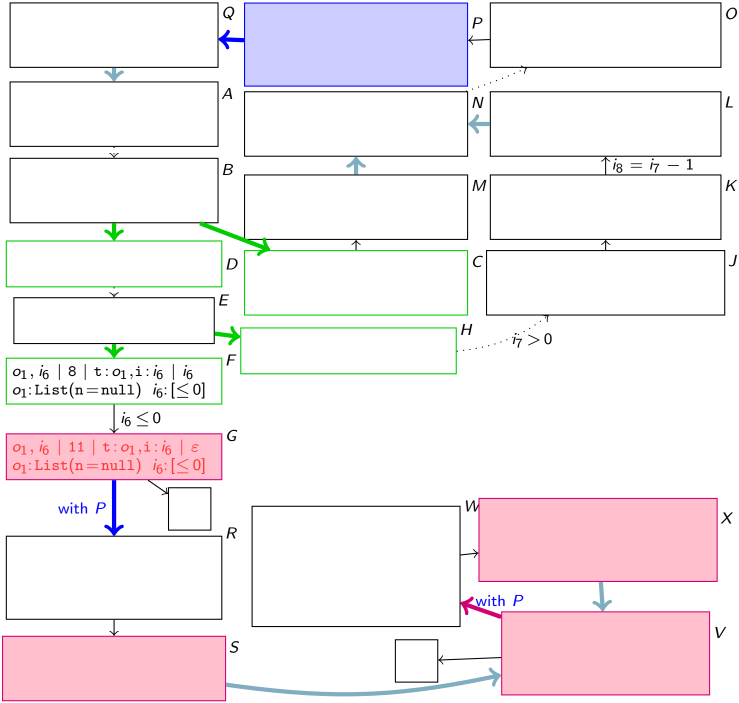
$f_{F,8}(\mathrm{eos},$
$\quad \mathrm{List}(\mathrm{null}),$
$\quad i_6,$
$\quad \mathrm{List}(\mathrm{null}),$
$\quad i_6)$



Converting Evaluation edges

$f_{F,8}(\mathsf{eos}, \mathsf{List}(\mathsf{null}), i_6, \mathsf{List}(\mathsf{null}), i_6)$

$f_{G,11}(\mathsf{eos}, \mathsf{List}(\mathsf{null}), i_6, \mathsf{List}(\mathsf{null}), i_6)$

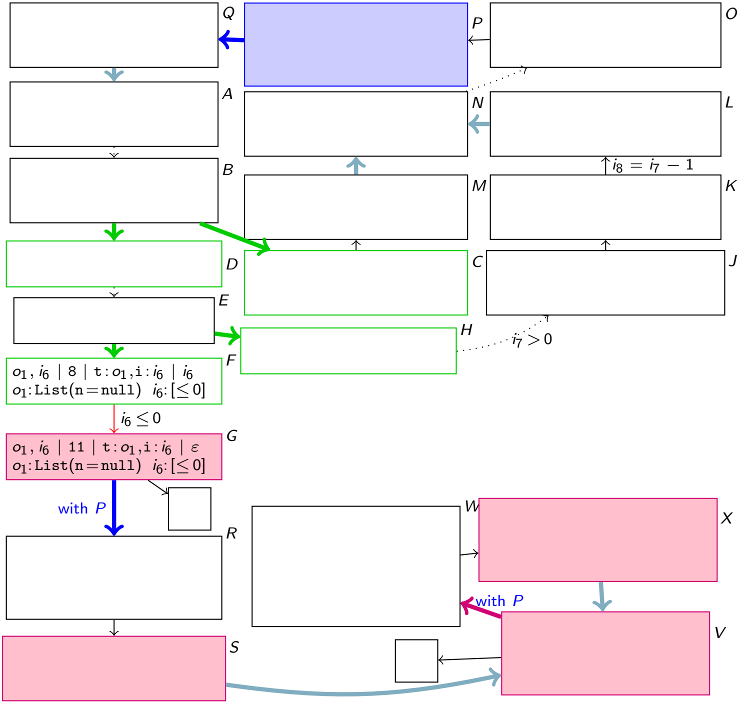


Converting Evaluation edges

$$f_{F,8}(\text{eos}, \text{List}(\text{null}), i_6, \text{List}(\text{null}), i_6)$$

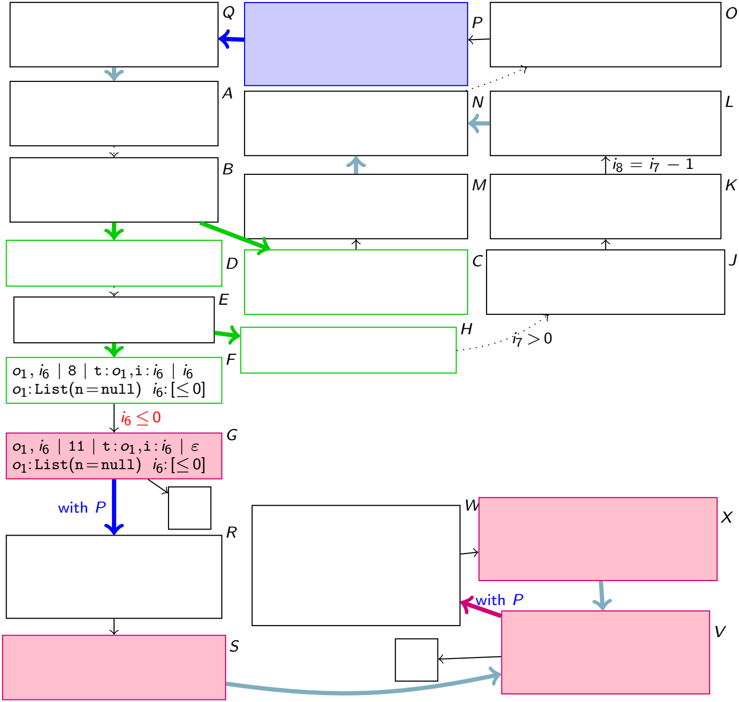
$$\longrightarrow$$

$$f_{G,11}(\text{eos}, \text{List}(\text{null}), i_6, \text{List}(\text{null}), i_6)$$



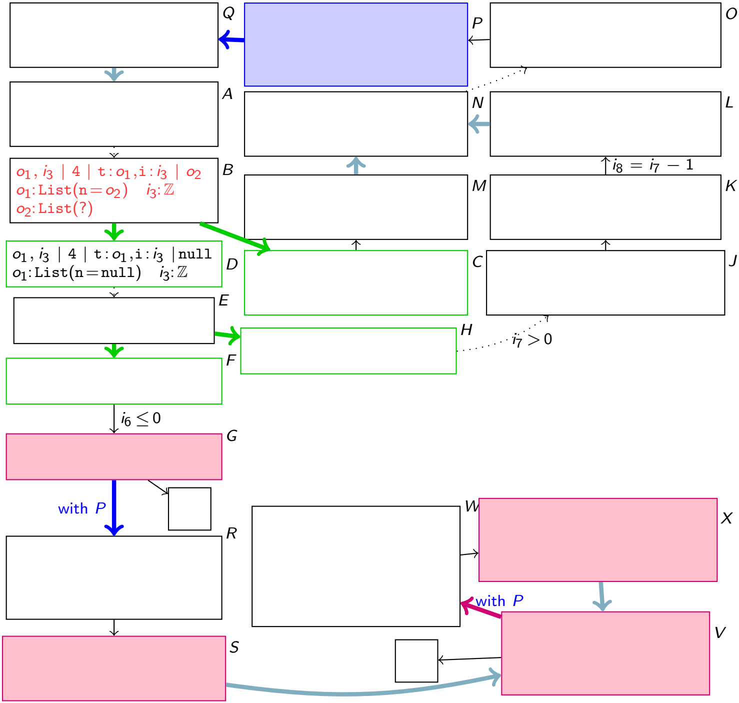
Converting Evaluation edges

$$f_{F,8}(\text{eos}, \text{List}(\text{null}), i_6, \text{List}(\text{null}), i_6) \longrightarrow f_{G,11}(\text{eos}, \text{List}(\text{null}), i_6, \text{List}(\text{null}), i_6) \mid i_6 \leq 0$$



Converting Refinement edges

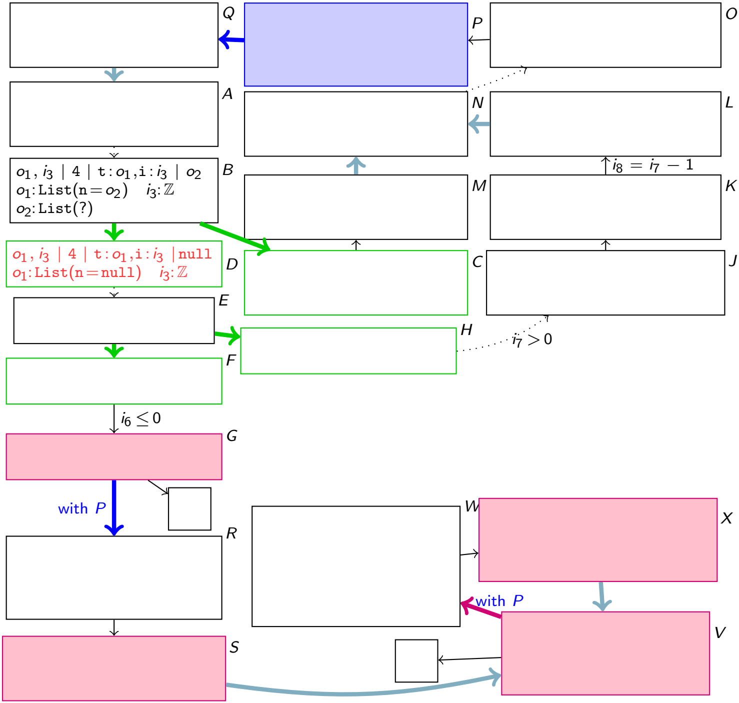
$\mathsf{f}_{B,4}(\mathsf{eos},$
$\mathsf{List}(o_2),$
$i_3,$
$\mathsf{List}(o_2),$
$i_3,$
$o_2)$



Converting Refinement edges

$f_{B,4}($eos,
List(o_2),
i_3,
List(o_2),
i_3,
$o_2)$

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

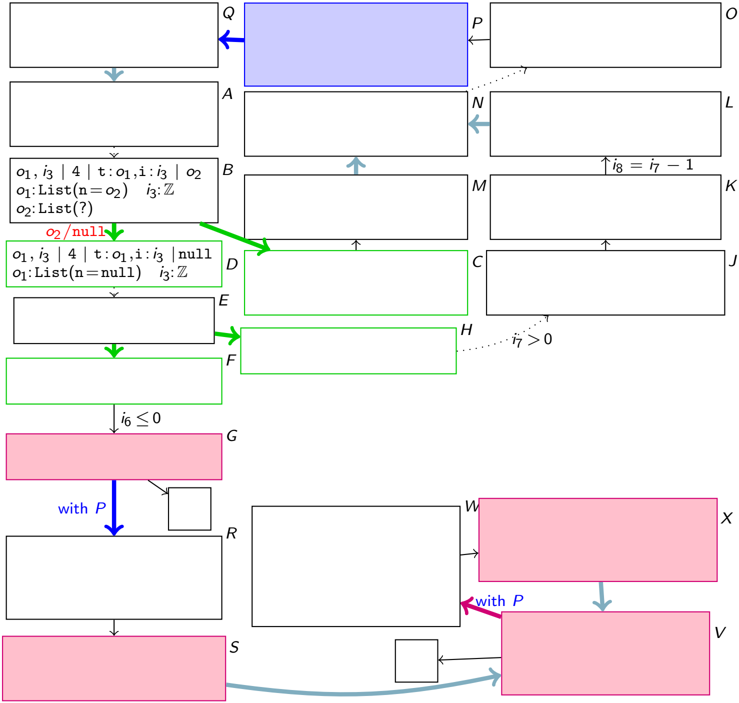


Converting Refinement edges

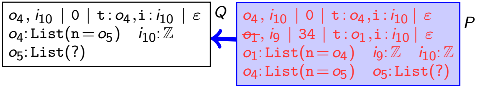
$\mathsf{f}_{B,4}(\mathsf{eos},$
$\quad \mathsf{List}(\mathsf{null}),$
$\quad i_3,$
$\quad \mathsf{List}(\mathsf{null}),$
$\quad i_3,$
$\quad \mathsf{null})$

$\longrightarrow$

$\mathsf{f}_{D,4}(\mathsf{eos},$
$\quad \mathsf{List}(\mathsf{null}),$
$\quad i_3,$
$\quad \mathsf{List}(\mathsf{null}),$
$\quad i_3,$
$\quad \mathsf{null})$

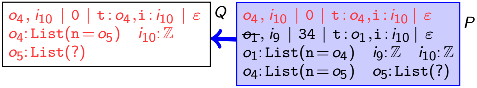


Converting stack split edges



$$\mathsf{f}_{P,34}(\mathsf{f}_{P,0}(\mathsf{eos}, \mathsf{List}(o_5), i_{10}, \mathsf{List}(o_5), i_{10}), \mathsf{List}(\mathsf{List}(o_5)), i_9, \mathsf{List}(\mathsf{List}(o_5)), i_{10})$$

Converting stack split edges



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

$$\longrightarrow$$

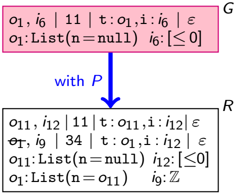
$$\mathsf{f}_{Q,0}(\mathsf{eos}, \mathsf{List}(o_5), i_{10}, \mathsf{List}(o_5), i_{10})$$

Converting call return edges

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

$$\mathsf{f}_{P,34}(\mathsf{f}_{P,0}(\mathsf{eos}, \mathsf{List}(o_5), i_{10}, \mathsf{List}(o_5), i_{10}), \mathsf{List}(\mathsf{List}(o_5)), i_9, \mathsf{List}(\mathsf{List}(o_5)), i_{10}))$$



Converting call return edges

P

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

$$\mathsf{f}_{P,34}(\mathsf{f}_{P,0}(\mathsf{eos}, \mathsf{List}(o_5), i_{10}, \mathsf{List}(o_5), i_{10}), \mathsf{List}(\mathsf{List}(o_5)), i_9, \mathsf{List}(\mathsf{List}(o_5)), i_{10}))$$

$$\mathsf{f}_{G,11}(\mathsf{eos}, \mathsf{List}(\mathsf{null}), i_6, \mathsf{List}(\mathsf{null}), i_6)$$

G

$o_1, i_6 \mid 11 \mid \texttt{t}: o_1, \texttt{i}: i_6 \mid \varepsilon$
$o_1: \texttt{List}(\texttt{n} = \texttt{null}) \quad i_6: [\leq 0]$

↓ with P

R

$o_{11}, i_{12} \mid 11 \mid \texttt{t}: o_{11}, \texttt{i}: i_{12} \mid \varepsilon$
$\not{o_1}, i_9 \mid 34 \mid \texttt{t}: o_1, \texttt{i}: i_{12} \mid \varepsilon$
$o_{11}: \texttt{List}(\texttt{n} = \texttt{null}) \quad i_{12}: [\leq 0]$
$o_1: \texttt{List}(\texttt{n} = o_{11}) \quad i_9: \mathbb{Z}$

From intersection:

- o_5 is `null`
- i_6, i_{10} renamed to i_{12}
- i_9 renamed to i_{13}

Converting call return edges

P

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

$$\mathsf{f}_{P,34}(\mathsf{f}_{G,11}(\mathsf{eos}, \mathsf{List}(\mathsf{null}), i_{12}, \mathsf{List}(\mathsf{null}), i_{12}), \mathsf{List}(\mathsf{List}(\mathsf{null})), i_{13}, \mathsf{List}(\mathsf{List}(\mathsf{null})), i_{12})$$

G

$o_1, i_6 \mid 11 \mid \mathtt{t}: o_1, \mathtt{i}: i_6 \mid \varepsilon$
$o_1: \mathtt{List}(\mathtt{n}{=}\mathtt{null}) \quad i_6: [\leq 0]$

↓ with P

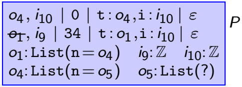
R

$o_{11}, i_{12} \mid 11 \mid \mathtt{t}: o_{11}, \mathtt{i}: i_{12} \mid \varepsilon$
$\cancel{o_1}, i_9 \mid 34 \mid \mathtt{t}: o_1, \mathtt{i}: i_{12} \mid \varepsilon$
$o_{11}: \mathtt{List}(\mathtt{n}{=}\mathtt{null}) \quad i_{12}: [\leq 0]$
$o_1: \mathtt{List}(\mathtt{n}{=}o_{11}) \quad i_9: \mathbb{Z}$

From intersection:

- o_5 is `null`
- i_6, i_{10} renamed to i_{12}
- i_9 renamed to i_{13}

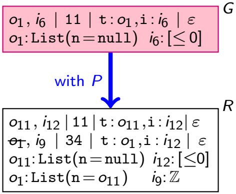
Converting call return edges



$$\mathsf{f}_{P,34}(\mathsf{f}_{G,11}(\mathsf{eos}, \mathsf{List}(\mathsf{null}), i_{12}, \mathsf{List}(\mathsf{null}), i_{12}), \mathsf{List}(\mathsf{List}(\mathsf{null})), i_{13}, \mathsf{List}(\mathsf{List}(\mathsf{null})), i_{12})$$

$$\longrightarrow$$

$$\mathsf{f}_{R,34}(\mathsf{f}_{R,11}(\mathsf{eos}, \mathsf{List}(\mathsf{null}), i_{12}, \mathsf{List}(\mathsf{null}), i_{12}), \mathsf{List}(\mathsf{List}(\mathsf{null})), i_9, \mathsf{List}(\mathsf{List}(\mathsf{null})), i_{12})$$



From intersection:

- o_5 is `null`
- i_6, i_{10} renamed to i_{12}
- i_9 renamed to i_{13}

The Example: TRS

TRS for appE

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

TRS corresponds to source program:

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

The Example: TRS

TRS for appE

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

TRS corresponds to source program:

(1) End of list and $\mathtt{i} \le 0$

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

The Example: TRS

TRS for appE

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

TRS corresponds to source program:

(1) End of list and $\texttt{i} \leq 0$

(2) End of list and $\texttt{i} > 0$

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

The Example: TRS

TRS for appE

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

TRS corresponds to source program:

(1) End of list and $\mathtt{i} \leq 0$

(2) End of list and $\mathtt{i} > 0$

(3) Not end of list

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

The Example: TRS

TRS for appE

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

TRS corresponds to source program:

(1) End of list and $\mathtt{i} \le 0$

(2) End of list and $\mathtt{i} > 0$

(3) Not end of list

(4), (5) Return from (1), (2)

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

The Example: TRS

TRS for appE

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

TRS corresponds to source program:

(1) End of list and $\texttt{i} \leq 0$

(2) End of list and $\texttt{i} > 0$

(3) Not end of list

(4), (5) Return from (1), (2)

Termination proof trivial.

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

The Example: TRS

TRS for appE

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

TRS corresponds to source program:

(1) End of list and `i` ≤ 0

(2) End of list and `i` > 0

(3) Not end of list

(4), (5) Return from (1), (2)

Termination proof trivial.

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

TRS for cappE

$$
\begin{array}{rcll}
f_{A'}(\ldots) & \to & f_{B'}(f_A(\ldots), \ldots) & \quad (1')\\
f_{B'}(f_V(\ldots), \ldots) & \to & f_{D'}(f_{D'}(\ldots), \ldots) & \quad (2')
\end{array}
$$

Theoretical basis

Definition 1: Java and the Termination Graph

A *computation path* in the Termination Graph is the embedding of a standard JVM computation into the graph.

Theoretical basis

Definition 1: Java and the Termination Graph

A *computation path* in the Termination Graph is the embedding of a standard JVM computation into the graph.

Theorem 1: Soundness of our approach

TRS corresponding to Termination Graph is terminating

Theoretical basis

Definition 1: Java and the Termination Graph

A *computation path* in the Termination Graph is the embedding of a standard JVM computation into the graph.

Theorem 1: Soundness of our approach

TRS corresponding to Termination Graph is terminating

$\Rightarrow$ Termination Graph has no infinite computation path

Theoretical basis

Definition 1: Java and the Termination Graph

A *computation path* in the Termination Graph is the embedding of a standard JVM computation into the graph.

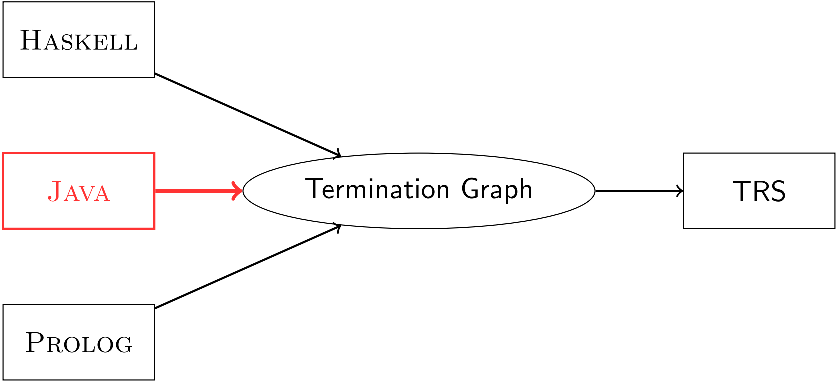
Theorem 1: Soundness of our approach

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 of JBC via term rewriting



Modular termination analysis of JBC via term rewriting

- Implemented in AProVE for full single-threaded JAVA

Modular termination analysis of JBC via term rewriting

- Implemented in AProVE for full single-threaded JAVA
- Evaluated on collection of 216 programs (including the *Termination Problem Data Base*):

Modular termination analysis of JBC via term rewriting

- Implemented in AProVE for full single-threaded JAVA
- Evaluated on collection of 216 programs (including the *Termination Problem Data Base*):

	Success	Failure	Timeout	Runtime
AProVE 2011	177	17	23	17.1
AProVE 2010	118	16	82	51.1
Julia	153	63	0	2.4
COSTA	120	95	1	5.4

Modular termination analysis of JBC via term rewriting

- Implemented in AProVE for full single-threaded JAVA
- Evaluated on collection of 216 programs (including the *Termination Problem Data Base*):

	Success	Failure	Timeout	Runtime
AProVE 2011	177	17	23	17.1
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 of JBC via term rewriting

- Implemented in AProVE for full single-threaded Java
- Evaluated on collection of 216 programs (including the *Termination Problem Data Base*):

	Success	Failure	Timeout	Runtime
AProVE 2011	177	17	23	17.1
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
- `http://aprove.informatik.rwth-aachen.de`

Modular termination analysis of JBC via term rewriting

- Implemented in AProVE for full single-threaded JAVA
- Evaluated on collection of 216 programs (including the *Termination Problem Data Base*):

	Success	Failure	Timeout	Runtime
AProVE 2011	177	17	23	17.1
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
- `http://aprove.informatik.rwth-aachen.de`
- termination of "real" languages can be analyzed automatically via existing term rewriting techniques