Automated Termination Analysis of JAVA BYTECODE by Term Rewriting

Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

joint work with C. Otto, M. Brockschmidt, C. von Essen

Termination Analysis for TRSs

$$\begin{array}{rrcl} \mathcal{R}: & \mathsf{plus}(x, 0) & \to & x \\ & \mathsf{plus}(x, \mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{plus}(x, y)) \end{array}$$

$\mathcal{R}$ is *terminating* iff there is no infinite evaluation $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \ldots$

Computation of "2 + 1":

$$\begin{array}{rcl} \mathsf{plus}(\mathsf{s}(\mathsf{s}(0)), \mathsf{s}(0)) & \to_{\mathcal{R}} & \mathsf{s}(\mathsf{plus}(\mathsf{s}(\mathsf{s}(0)), 0)) \\ & \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{s}(0))) \end{array}$$

- easier / more general than for programs
- suitable for automation
- **But:** halting problem is undecidable!
  $\Rightarrow$ automated termination proofs do not always succeed

Termination Analysis for TRSs

$$\begin{array}{rcl} \mathcal{P}ol(\,\mathsf{plus}(0,y)\,) & > & \mathcal{P}ol(\,y\,) \\ \mathcal{P}ol(\,\mathsf{plus}(\mathsf{s}(x),y)\,) & > & \mathcal{P}ol(\,\mathsf{s}(\mathsf{plus}(x,y))\,) \end{array}$$

$\mathcal{R}$ is *terminating* iff there is no infinite evaluation $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \ldots$

- **Goal:** Find order $\succ$ such that $\ell \succ r$ for all rules $\ell \to r \in \mathcal{R}$
- **Polynomial Order:** $\ell \succ r$ iff $\mathcal{P}ol(\ell) > \mathcal{P}ol(r)$

$$\begin{array}{rcl} \mathcal{P}ol(0) & = & 1 \\ \mathcal{P}ol(\mathsf{s}(t)) & = & 1 + \mathcal{P}ol(t) \\ \mathcal{P}ol(\mathsf{plus}(t_1,t_2)) & = & 2\,\mathcal{P}ol(t_1) + \mathcal{P}ol(t_2) \end{array}$$

Termination Analysis for TRSs

$$\begin{array}{rcl} 2 + \mathcal{P}ol(y) & > & \mathcal{P}ol(y) \\ 2\,(1 + \mathcal{P}ol(x)) + \mathcal{P}ol(y) & > & 1 + 2\,\mathcal{P}ol(x) + \mathcal{P}ol(y) \end{array}$$

$\mathcal{R}$ is *terminating* iff there is no infinite evaluation $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \ldots$

- **Goal:** Find order $\succ$ such that $\ell \succ r$ for all rules $\ell \to r \in \mathcal{R}$
- **Polynomial Order:** $\ell \succ r$ iff $\mathcal{P}ol(\ell) > \mathcal{P}ol(r)$

$$\begin{array}{rcl} \mathcal{P}ol(0) & = & 1 \\ \mathcal{P}ol(\mathsf{s}(t)) & = & 1 + \mathcal{P}ol(t) \\ \mathcal{P}ol(\mathsf{plus}(t_1, t_2)) & = & 2\,\mathcal{P}ol(t_1) + \mathcal{P}ol(t_2) \end{array}$$

- can be generated automatically by SAT solving (SAT '07)

Automated Termination Analysis for TRSs

- Classical Techniques (simplification orders)
  - Knuth–Bendix Order *(Knuth & Bendix, 70)*
  - Polynomial Order *(Lankford, 79)*
  - Lexicographic Path Order *(Kamin & Lévy, 80)*
  - Recursive Path Order *(Dershowitz, 82)*
- Recent Techniques (beyond simplification orders)
  - Transformation Order *(Bellegarde & Lescanne, 87)*
  - Semantic Labelling *(Zantema, 95)*
  - Dependency Pairs *(Arts & Giesl, 96)*
  - Monotonic Semantic Path Order *(Borralleras, Ferreira, Rubio, 00)*
  - Match-Bounds *(Geser, Hofbauer, Waldmann, 03)*
  - Matrix Order *(Endrullis, Hofbauer, Waldmann, Zantema, 06)*
- Active area of research: *International Workshop on Termination*

Integer Term Rewriting

Termination of Term Rewriting

- powerful for algorithms on user-defined data structures
(automatic generation of orders to compare arbitrary terms)

- naive handling of pre-defined data structures
(represent data objects by terms)

Representing integers

$$
\begin{aligned}
0 &\equiv \mathsf{pos}(0) \equiv \mathsf{neg}(0)\\
1 &\equiv \mathsf{pos}(\mathsf{s}(0))\\
-1 &\equiv \mathsf{neg}(\mathsf{s}(0))\\
1000 &\equiv \mathsf{pos}(\mathsf{s}(\mathsf{s}(\ldots \mathsf{s}(0)\ldots)))
\end{aligned}
$$

Rules for pre-defined operations

$$
\begin{aligned}
\mathsf{pos}(x) + \mathsf{neg}(y) &\to \mathsf{minus}(x, y)\\
\mathsf{neg}(x) + \mathsf{pos}(y) &\to \mathsf{minus}(y, x)\\
\mathsf{pos}(x) + \mathsf{pos}(y) &\to \mathsf{pos}(\mathsf{plus}(x, y))\\
\mathsf{neg}(x) + \mathsf{neg}(y) &\to \mathsf{neg}(\mathsf{plus}(x, y))\\
\mathsf{minus}(x, 0) &\to \mathsf{pos}(x)\\
\mathsf{minus}(0, y) &\to \mathsf{neg}(y)\\
\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{minus}(x, y)
\end{aligned}
$$

Integer Term Rewriting

Termination of Term Rewriting

- powerful for algorithms on user-defined data structures
  (automatic generation of orders to compare arbitrary terms)

- naive handling of pre-defined data structures
  (represent data objects by terms)

Integer Term Rewriting (RTA '09)

- integrated pre-defined data structures like $\mathbb{Z}$ into term rewriting
- adapted techniques to prove termination of integer TRSs
- for algorithms on integers: as powerful as direct techniques
- for user-defined data structures: as powerful as before

Integer Term Rewriting

- $\mathcal{F}_{int}$: pre-defined symbols
  - $\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$
  - $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$
  - $+, -, *, /, \%$
  - $>, \geq, <, \leq, ==, !=$
  - $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$
- $\mathcal{PD}$: pre-defined rules

  $2*21 \to 42 \qquad 42 \geq 23 \to \mathsf{true}$

  $\mathsf{true} \wedge \mathsf{false} \to \mathsf{false} \qquad \ldots$

  $\Rightarrow$ pre-defined operations only evaluated if all arguments are from $\mathbb{Z}$ or $\mathbb{B}$

- ITRS $\mathcal{R}$: finite TRS
  - no pre-defined symbols except $\mathbb{Z}$ and $\mathbb{B}$ in lhs
  - lhs $\notin \mathbb{Z} \cup \mathbb{B}$
  - rewrite relation defined w.r.t. $\mathcal{R} \cup \mathcal{PD}$ (innermost rewriting)

Example ITRS computing $\sum_{i=y}^{x} i$

$$
\begin{aligned}
\mathsf{sum}(x, y) &\to \mathsf{sif}(x \geq y, x, y)\\
\mathsf{sif}(\mathsf{true}, x, y) &\to y + \mathsf{sum}(x, y + 1)\\
\mathsf{sif}(\mathsf{false}, x, y) &\to 0
\end{aligned}
$$

Integer Term Rewriting

$$
\begin{array}{llllll}
\underline{\mathsf{sum}(1,1)} & \to_{\mathcal{R}} & \mathsf{sif}(\underline{1 \geq 1}, 1, 1) & \to_{\mathcal{R}} & \underline{\mathsf{sif}(\mathsf{true}, 1, 1)} \\
 & \to_{\mathcal{R}} & 1 + \mathsf{sum}(1, \underline{1+1}) & \to_{\mathcal{R}} & 1 + \underline{\mathsf{sum}(1, 2)} \\
 & \to_{\mathcal{R}} & 1 + \mathsf{sif}(\underline{1 \geq 2}, 1, 2) & \to_{\mathcal{R}} & 1 + \underline{\mathsf{sif}(\mathsf{false}, 1, 2)} \\
 & \to_{\mathcal{R}} & \underline{1 + 0} & \to_{\mathcal{R}} & 1
\end{array}
$$

- ITRS $\mathcal{R}$: finite TRS
  - no pre-defined symbols except $\mathbb{Z}$ and $\mathbb{B}$ in lhs
  - lhs $\notin \mathbb{Z} \cup \mathbb{B}$
  - rewrite relation defined w.r.t. $\mathcal{R} \cup \mathcal{PD}$ (innermost rewriting)

Example ITRS computing $\sum_{i=y}^{x} i$

$$
\begin{array}{rcl}
\mathsf{sum}(x, y) & \to & \mathsf{sif}(x \geq y, x, y) \\
\mathsf{sif}(\mathsf{true}, x, y) & \to & y + \mathsf{sum}(x, y + 1) \\
\mathsf{sif}(\mathsf{false}, x, y) & \to & 0
\end{array}
$$

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?

Termination of Programming Languages

Functional Languages

- first-order languages with strict evaluation strategy
  (Walther, 94), (Giesl, 95), (Lee, Jones, Ben-Amram, 01)
- ensuring termination (e.g., by typing)
  (Telford & Turner, 00), (Xi, 02), (Abel, 04), (Barthe et al, 04) etc.
- outermost termination of untyped first-order rewriting
  (Fissore, Gnaedig, Kirchner, 02), (Endrullis & Hendriks, 09), (Raffelsieper & Zantema, 09), (Thiemann, 09)
- automated technique for small HASKELL-like language
  (Panitz & Schmidt-Schauss, 97)

- do not work on full existing languages
- no use of TRS-techniques (stand-alone methods)

Functional Languages

- using TRS-techniques for HASKELL is challenging

  - HASKELL has a lazy evaluation strategy.
    For TRSs, one proves termination of *all* reductions.
  - HASKELL's equations are handled from top to bottom.
    For TRSs, *any* rule may be used for rewriting.
  - HASKELL has polymorphic types.
    TRSs are *untyped*.
  - In HASKELL-programs, often only some functions terminate.
    TRS-methods try to prove termination of *all* terms.
  - HASKELL is a higher-order language.
    Most automatic TRS-methods only handle *first-order* rewriting.

Termination of Programming Languages

Functional Languages

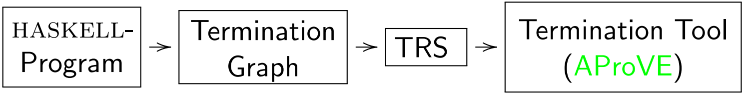
- using TRS-techniques for HASKELL is challenging
- **New approach** (RTA '06),(TOPLAS '10)
  - Frontend
    - evaluate HASKELL a few steps ⇒ termination graph

      termination graph captures evaluation strategy, types, etc.
    - transform termination graph ⇒ TRS
  - Backend
    - prove termination of the resulting TRS
      (using existing techniques & tools)
- implemented in **AProVE**
  - accepts full HASKELL 98 language
  - successfully evaluated with standard HASKELL-libraries
    (succeeds on approx. 80 % of the functions in standard libraries)

Termination of Programming Languages

Functional Languages

- using TRS-techniques for HASKELL is challenging



- implemented in **AProVE**
  - accepts full HASKELL 98 language
  - successfully evaluated with standard HASKELL-libraries
    (succeeds on approx. 80 % of the functions in standard libraries)

Termination of Programming Languages

Logic Languages

- well-developed field *(De Schreye & Decorte, 94) etc.*
- **direct approaches:** work directly on the logic program
  - cTI *(Mesnard et al)*
  - TerminWeb *(Codish et al)*
  - TermiLog *(Lindenstrauss et al)*
  - Polytool *(Nguyen, De Schreye, Giesl, Schneider-Kamp)*

  TRS-techniques can be adapted to work *directly* on the LP
- **transformational approaches:** transform LP to TRS
  - TALP *(Ohlebusch et al)*
  - AProVE *(Giesl et al)*
- only for *definite* LP (without cut)
- not for real PROLOG

Termination of Programming Languages

Logic Languages

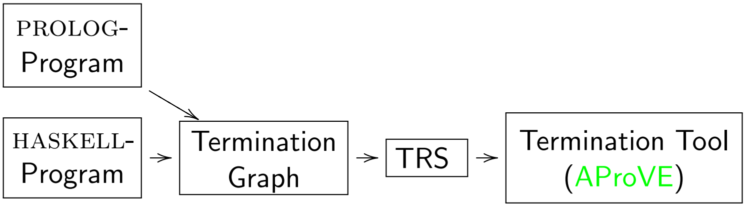
- analyzing PROLOG is challenging due to cuts etc.
- **New approach** (ICLP '10)
  - Frontend
    - evaluate PROLOG a few steps ⇒ termination graph

      termination graph captures evaluation strategy due to cuts etc.
    - transform termination graph ⇒ TRS
  - Backend
    - prove termination of the resulting TRS
      (using existing techniques & tools)
- implemented in **AProVE**
  - successfully evaluated on PROLOG-collections with cuts
  - most powerful termination tool for PROLOG
    (winner of the international *termination competition* for PROLOG)

Termination of Programming Languages

Logic Languages

- analyzing PROLOG is challenging due to cuts etc.



- implemented in **AProVE**
  - successfully evaluated on PROLOG-collections with cuts
  - most powerful termination tool for PROLOG
    (winner of the international *termination competition* for PROLOG)

Termination of Programming Languages

Imperative Languages

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

- used at Microsoft for verifying Windows device drivers
- no use of TRS-techniques (stand-alone methods)

Termination of Programming Languages

Imperative Languages

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

Termination of Programming Languages

Imperative Languages

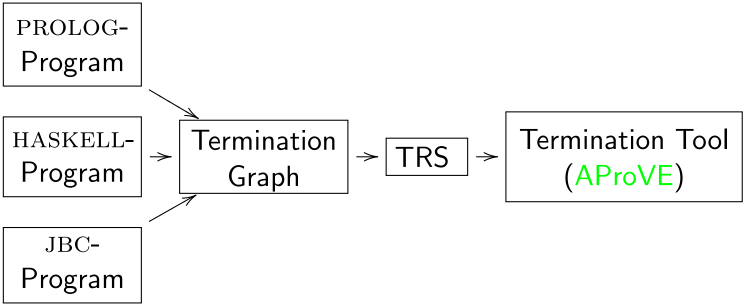
- using TRS-techniques for JBC is challenging
- **New approach** (RTA '10)
  - Frontend
    - evaluate JBC a few steps ⇒ termination graph

      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**
  - successfully evaluated on JBC-collection
  - most powerful termination tool for JBC
    (winner of the international *termination competition* for JBC)

Termination of Programming Languages

Imperative Languages

- using TRS-techniques for JBC is challenging



- implemented in **AProVE**
  - successfully evaluated on JBC-collection
  - most powerful termination tool for JBC
    (winner of the international *termination competition* for JBC)

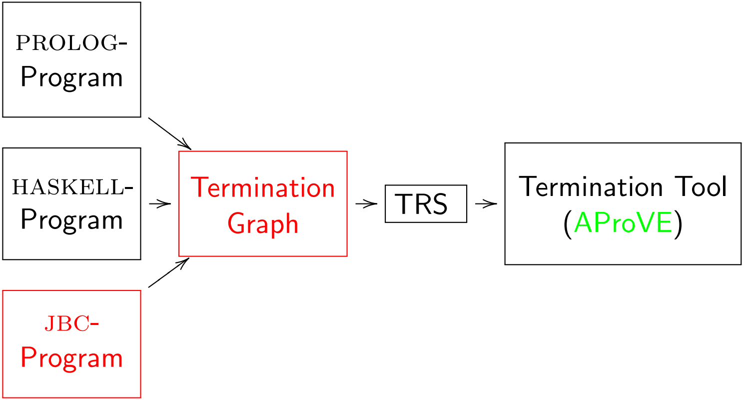
Termination of Programming Languages

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

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

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

From JBC to Termination Graphs



Example

```
00: aload_0       // load num to opstack
01: ifnull 8      // jump to line 8 if top
                  // of opstack is null
04: aload_1       // load limit
05: ifnonnull 9  // jump if not null
08: return
09: aload_0       // load num
10: astore_2      // store into copy
11: aload_0       // load num
12: getfield val // load field val
15: aload_1       // load limit
16: getfield val // load field val
19: if_icmpge 35 // jump if
                  // num.val >= limit.val
22: aload_2       // load copy
23: aload_2       // load copy
24: getfield val // load field val
27: iconst_1      // load constant 1
28: iadd          // add copy.val and 1
29: putfield val // store into copy.val
32: goto 11
35: return
```

```
public class Int {
  // only wrap a primitive int
  private int val;

  // count up to the value
  // in "limit"
  public static void count(
      Int num, Int limit) {

    if (num == null
        || limit == null) {
        return;
    }

    // introduce sharing
    Int copy = num;

    while (num.val < limit.val) {
        copy.val++;
    }
  }
}
```

Abstract States of the JVM

```
00: aload_0       // load num to opstack
01: ifnull 8      // jump to line 8 if top
                  // of opstack is null
04: aload_1       // load limit
05: ifnonnull 9   // jump if not null
08: return
09: aload_0       // load num
10: astore_2      // store into copy
11: aload_0       // load num
12: getfield val  // load field val
15: aload_1       // load limit
16: getfield val  // load field val
19: if_icmpge 35  // jump if
                  // num.val >= limit.val
22: aload_2       // load copy
23: aload_2       // load copy
24: getfield val  // load field val
27: iconst_1      // load constant 1
28: iadd          // add copy.val and 1
29: putfield val  // store into copy.val
32: goto 11
35: return
```

$\texttt{ifnull 8} \mid \texttt{n}:o_1, \texttt{l}:o_2 \mid o_1$

$o_1 = \texttt{Int}(\texttt{val} = i_1) \quad i_1 = (-\infty, \infty)$

$o_2 = \texttt{Int}(?)$

4 components

1. next program instruction
2. values of local variables (value of `num` is *reference* o_1)
3. values on the operand stack
4. information about the heap
   - object at address o_2 is `null` or of type `Int`
   - object at o_1 has type `Int`, `val`-field has value i_1
   - i_1 is an arbitrary integer
   - no sharing

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
     ⋮
19: if_icmpge 35
     ⋮
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

A

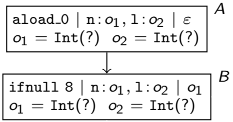
aload_0 | n : o_1, l : o_2 | ε
o_1 = Int(?) o_2 = Int(?)

State A:

- do all calls of `count` terminate?
- `num` and `limit` are arbitrary, but distinct `Int`-objects

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
     ⋮
19: if_icmpge 35
     ⋮
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

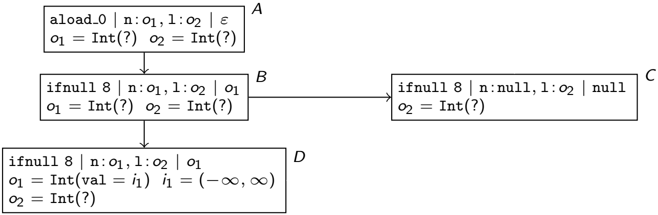


State *B*:

- "aload_0" loads value of num on operand stack
- *A* connected to *B* by *evaluation edge*

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
     ⋮
19: if_icmpge 35
     ⋮
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

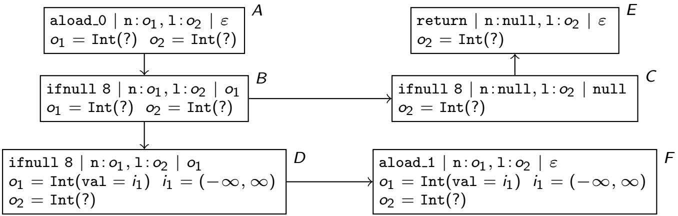


States *C* and *D*:

- "`ifnull 8`" needs to know whether o_1 is `null`
- *refine* information about heap (*refinement edges*)

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
     ⋮
19: if_icmpge 35
     ⋮
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

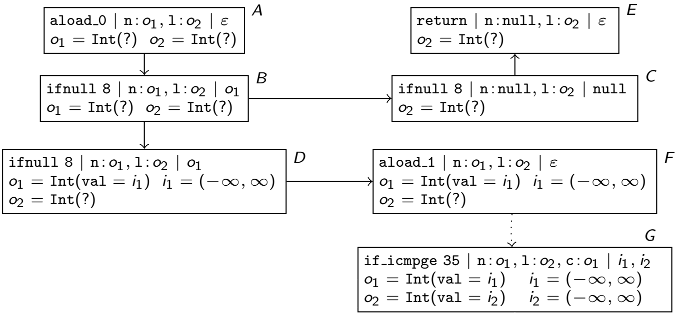


States *E* and *F*:

- evaluate "`ifnull 8`" in *C* and *D*
- *evaluation edges*

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
     :
19: if_icmpge 35
     :
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

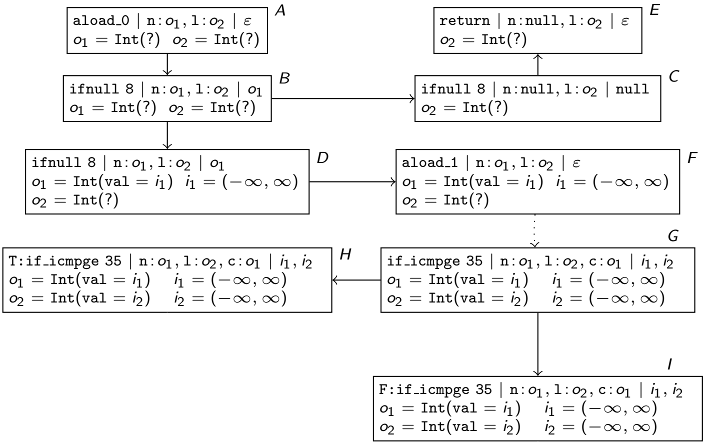


State G:

- in state F, check if `limit` is `null` analogously
- aliasing in G: `num` and `copy` point to the same address o_1
- `val`-fields of `num` and `limit` pushed on operand stack

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
     :
19: if_icmpge 35
     :
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

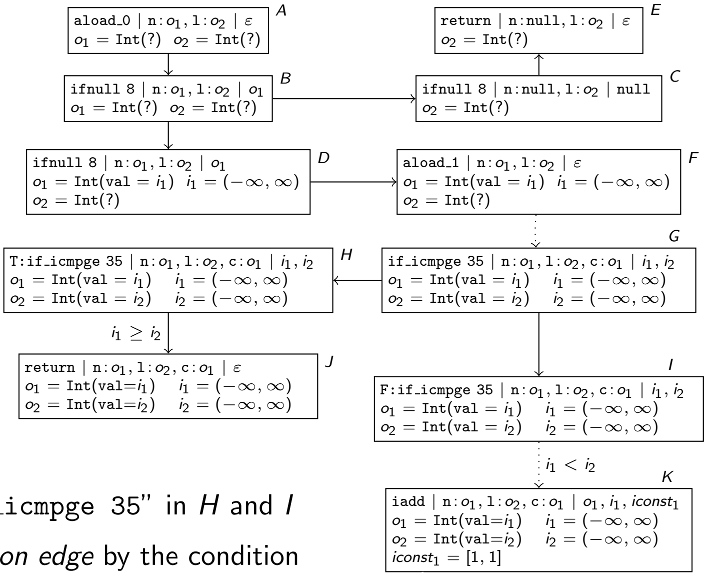


States H and I:

- "`if_icmpge 35`" needs to know whether $i_1 \geq i_2$
- *refine* information about heap (*refinement edges*)

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
    ⋮
19: if_icmpge 35
    ⋮
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```



States J and K:

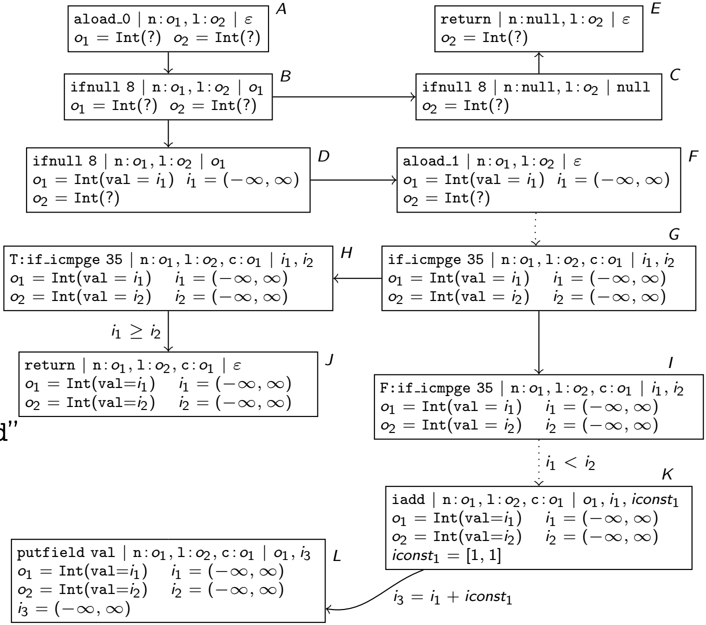
- evaluate "if_icmpge 35" in H and I
- label *evaluation edge* by the condition
- val-field of copy and integer variable with value 1 on operand stack

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
      ⋮
19: if_icmpge 35
      ⋮
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

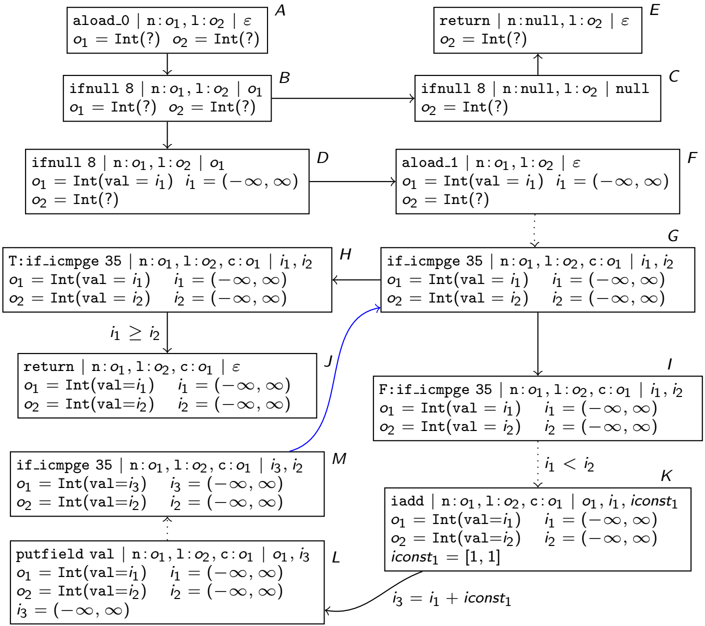
State L:

- evaluate "iadd"
- new variable i_3
- label edge by connection



From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
    :
19: if_icmpge 35
    :
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```



State *M*:

- again reaches "`if_icmpge`"
- *M* *instance* of *G*
- *instantiation edge*

From JBC to Termination Graphs

Termination Graphs

- expand nodes until all leaves correspond to program ends
- by appropriate generalization steps,
  one always reaches a *finite* termination graph
- state s_1 is *instance* of s_2 iff
  every concrete state described by s_1 is also described by s_2

Using Termination Graphs for Termination Proofs

- every JBC-computation of concrete states
  corresponds to a *computation path* in the termination graph
- termination graph is called *terminating* iff
  it has no infinite computation path

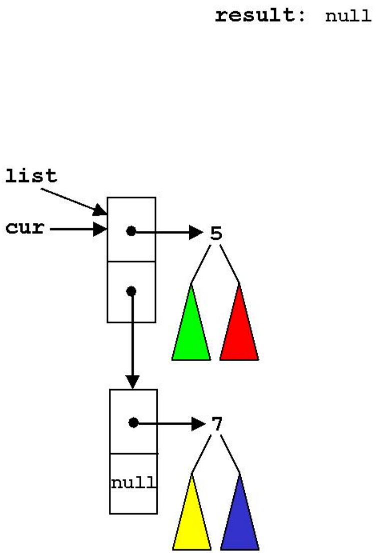
Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

```
public class Tree {
  int value;
  Tree left;
  Tree right;
}

public class TreeList {
  Tree value;
  TreeList next;
}

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

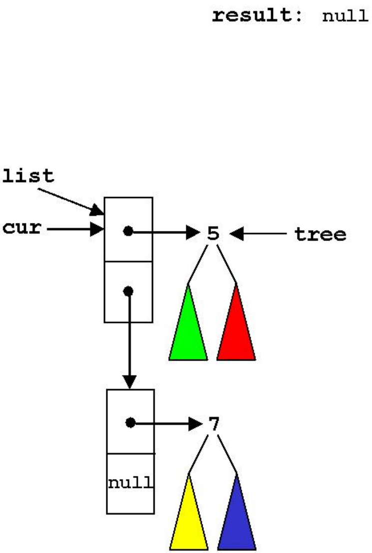
Example with User-Defined Data Type

```java
public class Flatten {
  public static IntList
        flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```



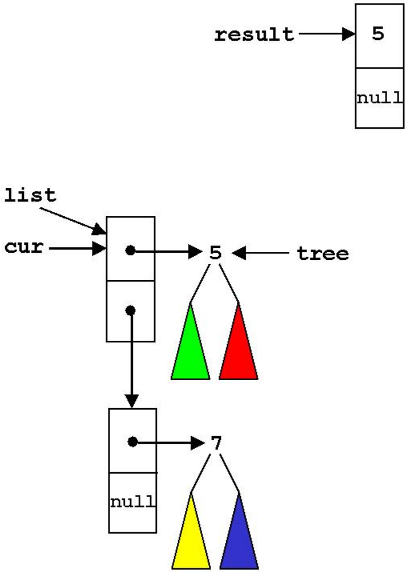
Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```



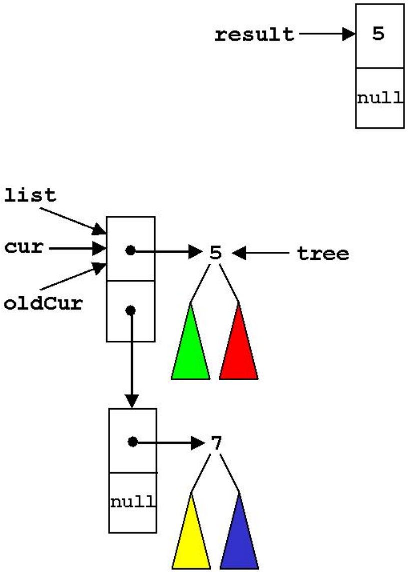
Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```



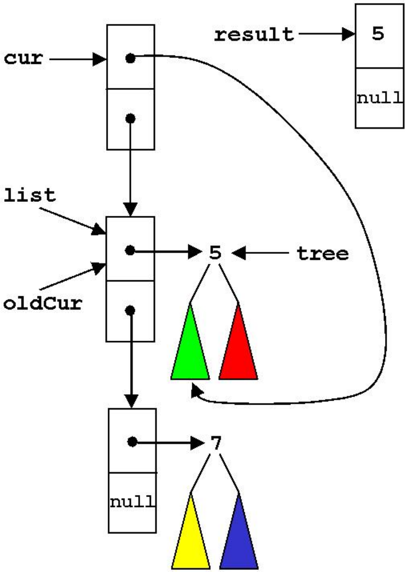
Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```



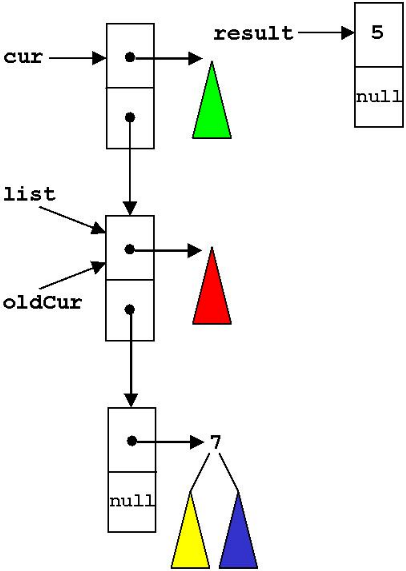
Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
        IntList oldIntList = result;
        result = new IntList();
        result.value = tree.value;
        result.next = oldIntList;
        TreeList oldCur = cur;
        cur = new TreeList();
        cur.value = tree.left;
        cur.next = oldCur;
        oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```



Example with User-Defined Data Type

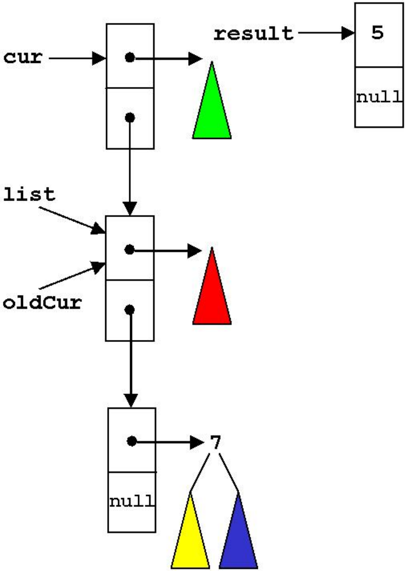
```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```



no termination by *path length*

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
        flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```



cur and list can be *sharing*

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

General state at beginning of loop body

$\texttt{aload\_1} \mid \texttt{l}:o_1, \texttt{c}:o_2, \texttt{r}:o_3 \mid \varepsilon$
$o_1 = \texttt{TreeList}(?)$ $o_2 = \texttt{TreeList}(?)$
$o_3 = \texttt{IntList}(?)$
$o_1 =^? o_2$ $o_1 \searrow\!\swarrow o_2$

Annotations

- $o_1 =^? o_2$: o_1 and o_2 *may be equal*
- $o_1 \searrow\!\swarrow o_2$: o_1 and o_2 *may join*
  - $o \to o'$ iff object at address o has a field with value o'
  - $o_1 \searrow\!\swarrow o_2$: $o_1 \to^* \circ \leftarrow^+ o_2$ or $o_1 \to^+ \circ \leftarrow^* o_2$
- $o\,!$: *o does not have to be a tree*

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

A

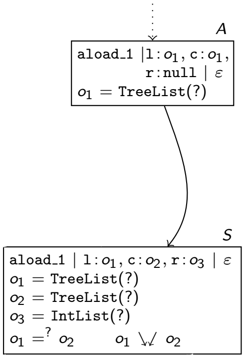
aload_1 | l : o_1, c : o_1, r : null | ε
o_1 = TreeList(?)

State *A*:

- reaches loop condition "`cur != null`" for the first time
- `list` and `cur` (o_1) are equal

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
        flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```



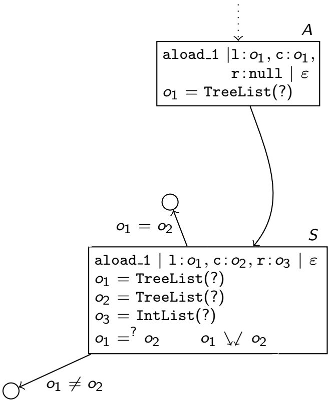
State S:

- *generalize* A to obtain finite termination graph
- `list` (o_1) and `cur` (o_2) may be equal and may join

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

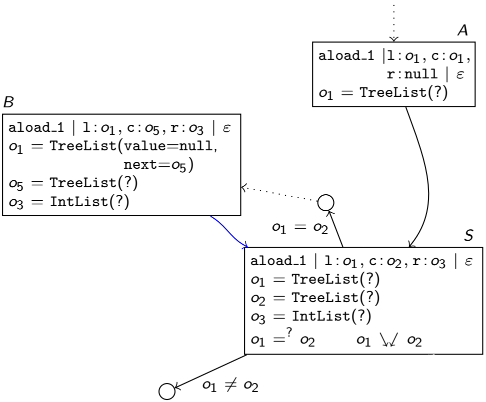
State S:



- *refinement* of annotation $o_1 =^? o_2$

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

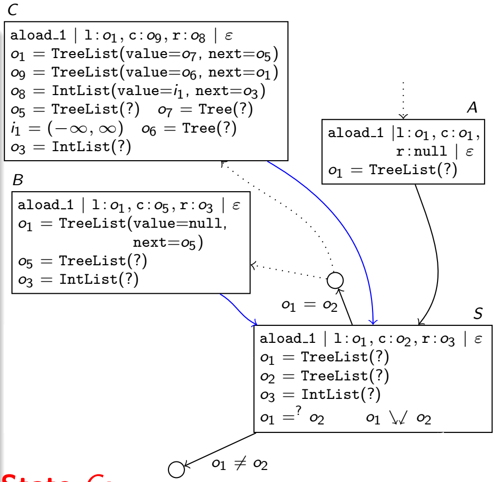


State B:

- reach loop condition if `tree == null`
- `list` $\rightarrow^{+} \circ \leftarrow^{*}$ `cur`
- B is *instance* of S

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
        flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

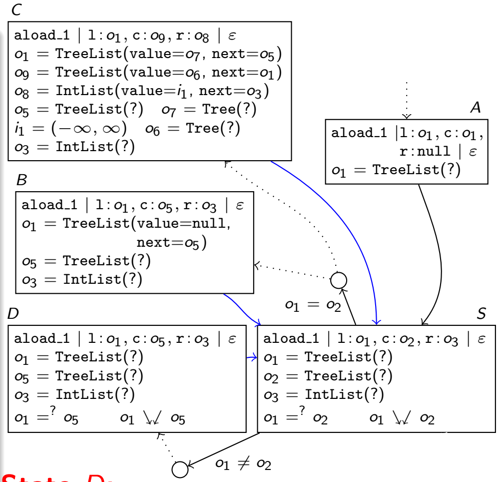


State C:

- $\texttt{Tree(value=}i_1\texttt{, left=}o_6\texttt{, right=}o_7\texttt{)}$
- $\texttt{list} \rightarrow^* \circ \leftarrow^+ \texttt{cur}$
- C is *instance* of S

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
        flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

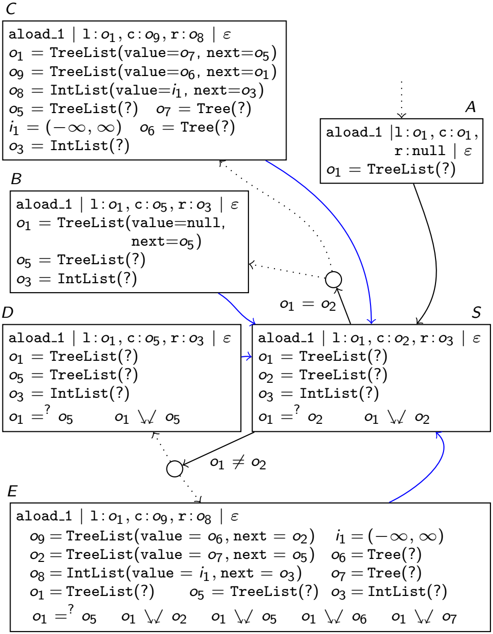


State *D*:

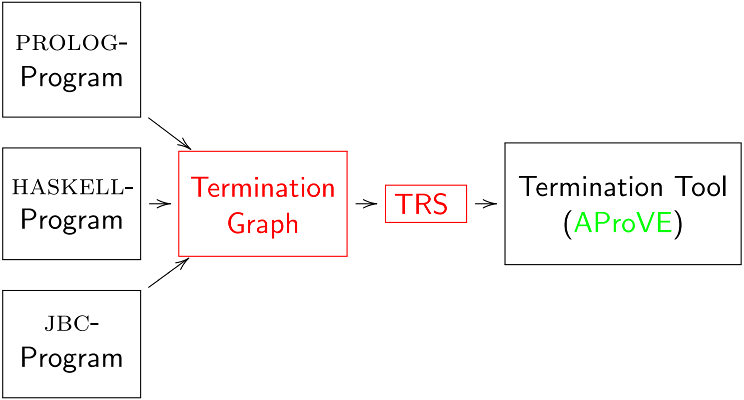
- o_2 = TreeList(value=o_4, next=o_5)
- tree (o_4) is null
- *D* is *instance* of *S*

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
        flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
        IntList oldIntList = result;
        result = new IntList();
        result.value = tree.value;
        result.next = oldIntList;
        TreeList oldCur = cur;
        cur = new TreeList();
        cur.value = tree.left;
        cur.next = oldCur;
        oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```



From Termination Graphs to TRSs



Transforming Objects to Terms

$\texttt{aload\_1} \mid \color{red}{\texttt{l}: o_1}, \texttt{c}: o_9, \texttt{r}: o_8 \mid \varepsilon$

$o_9 = \texttt{TreeList}(\texttt{value} = o_6, \texttt{next} = o_2) \qquad i_1 = (-\infty, \infty)$

$o_2 = \texttt{TreeList}(\texttt{value} = o_7, \texttt{next} = o_5) \qquad o_6 = \texttt{Tree}(?)$

$o_8 = \texttt{IntList}(\texttt{value} = i_1, \texttt{next} = o_3) \qquad o_7 = \texttt{Tree}(?)$

$\color{red}{o_1 = \texttt{TreeList}(?)} \qquad o_5 = \texttt{TreeList}(?) \qquad o_3 = \texttt{IntList}(?)$

$o_1 =^? o_5 \qquad o_1 \searrow\!\swarrow o_2 \qquad o_1 \searrow\!\swarrow o_5 \qquad o_1 \searrow\!\swarrow o_6 \qquad o_1 \searrow\!\swarrow o_7$

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

- term for o_1: o_1

Transforming Objects to Terms

$\texttt{aload\_1} \mid \texttt{l}: o_1, \texttt{c}: o_9, \texttt{r}: o_8 \mid \varepsilon$

$o_9 = \texttt{TreeList}(\texttt{value} = o_6, \texttt{next} = o_2)$ $\quad i_1 = (-\infty, \infty)$

$o_2 = \texttt{TreeList}(\texttt{value} = o_7, \texttt{next} = o_5)$ $\quad o_6 = \texttt{Tree}(?)$

$o_8 = \texttt{IntList}(\texttt{value} = i_1, \texttt{next} = o_3)$ $\quad o_7 = \texttt{Tree}(?)$

$o_1 = \texttt{TreeList}(?)$ $\quad o_5 = \texttt{TreeList}(?)$ $\quad o_3 = \texttt{IntList}(?)$

$o_1 =^? o_5 \quad o_1 \searrow\!\!\swarrow o_2 \quad o_1 \searrow\!\!\swarrow o_5 \quad o_1 \searrow\!\!\swarrow o_6 \quad o_1 \searrow\!\!\swarrow o_7$

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

- term for o_1: o_1
- term for o_2: $\mathsf{TL}(o_7, o_5)$

Transforming Objects to Terms

$$
\begin{array}{l}
\texttt{aload\_1} \mid \texttt{l}: o_1, \texttt{c}: o_9, \texttt{r}: o_8 \mid \varepsilon \\
o_9 = \texttt{TreeList}(\texttt{value} = o_6, \texttt{next} = o_2) \quad i_1 = (-\infty, \infty) \\
o_2 = \texttt{TreeList}(\texttt{value} = o_7, \texttt{next} = o_5) \quad o_6 = \texttt{Tree}(?) \\
o_8 = \texttt{IntList}(\texttt{value} = i_1, \texttt{next} = o_3) \quad o_7 = \texttt{Tree}(?) \\
o_1 = \texttt{TreeList}(?) \quad o_5 = \texttt{TreeList}(?) \quad o_3 = \texttt{IntList}(?) \\
o_1 =^? o_5 \quad o_1 \searrow\!\!\swarrow o_2 \quad o_1 \searrow\!\!\swarrow o_5 \quad o_1 \searrow\!\!\swarrow o_6 \quad o_1 \searrow\!\!\swarrow o_7
\end{array}
$$

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

- term for o_1: o_1
- term for o_2: $\mathsf{TL}(o_7, o_5)$
- term for o_9: $\mathsf{TL}(o_6, \mathsf{TL}(o_7, o_5))$

Transforming Objects to Terms

$\texttt{aload\_1} \mid \texttt{l}: o_1, \texttt{c}: o_9, \texttt{r}: o_8 \mid \varepsilon$

$o_9 = \texttt{TreeList}(\texttt{value} = o_6, \texttt{next} = o_2)$ $\quad i_1 = (-\infty, \infty)$

$o_2 = \texttt{TreeList}(\texttt{value} = o_7, \texttt{next} = o_5)$ $\quad o_6 = \texttt{Tree}(?)$

$o_8 = \texttt{IntList}(\texttt{value} = i_1, \texttt{next} = o_3)$ $\quad o_7 = \texttt{Tree}(?)$

$o_1 = \texttt{TreeList}(?)$ $\quad o_5 = \texttt{TreeList}(?)$ $\quad o_3 = \texttt{IntList}(?)$

$o_1 =^? o_5 \quad o_1 \searrow\!\!\swarrow o_2 \quad o_1 \searrow\!\!\swarrow o_5 \quad o_1 \searrow\!\!\swarrow o_6 \quad o_1 \searrow\!\!\swarrow o_7$

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

- term for o_1: o_1
- term for o_2: $\mathsf{TL}(o_7, o_5)$
- term for o_9: $\mathsf{TL}(o_6, \mathsf{TL}(o_7, o_5))$
- term for o_8: $\mathsf{IL}(i_1, o_3)$

Transforming Objects to Terms

Class Hierarchy

- for every class `C` with n fields, introduce function symbol C with $n+1$ arguments
- first argument: part of the object corresponding to subclasses of `C`

```
public class A {
  int a;
}
```

```
public class B extends A {
  int b;
}
```

```
A x = new A();
x.a = 1;
```

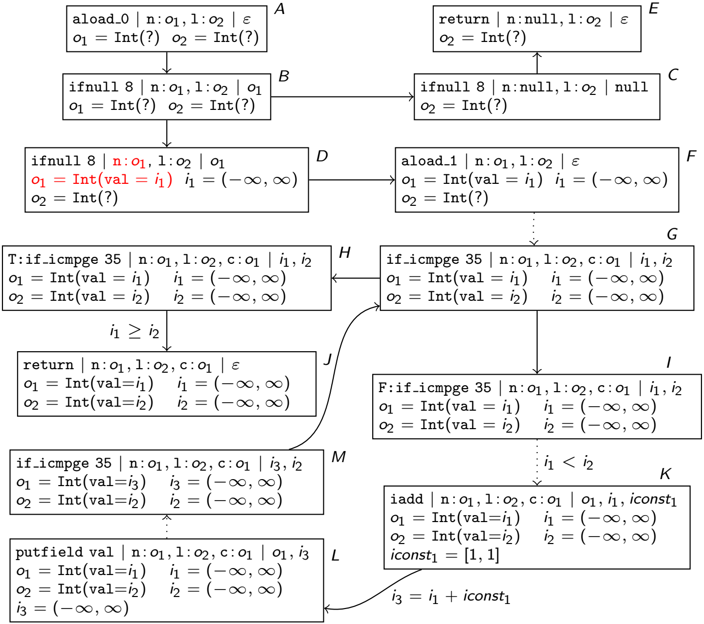
```
B y = new B();
y.a = 2;
y.b = 3;
```

- term for `x`: $\mathsf{jlO}(\mathsf{A}(\mathsf{eoc}, 1))$ (eoc for "end of class")
- term for `y`: $\mathsf{jlO}(\mathsf{A}(\mathsf{B}(\mathsf{eoc}, 3), 2))$ (jlO for "`java.lang.Object`")

Transforming States to Tuples of Terms

Transforming D

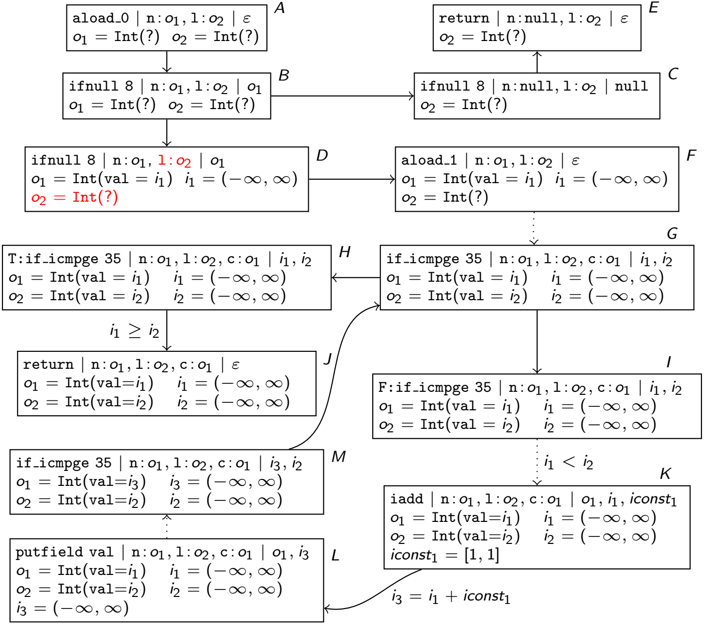
jlO(Int(eoc, i_1))



Transforming States to Tuples of Terms

Transforming *D*

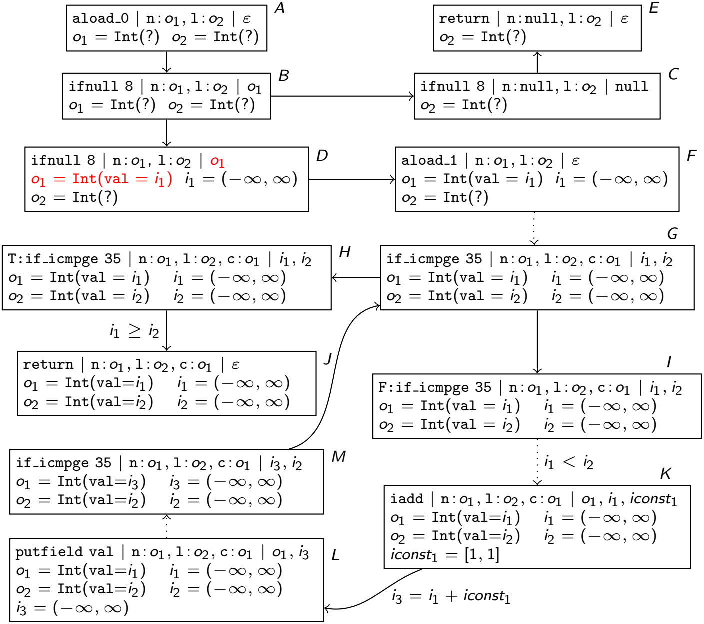
jlO(Int(eoc, i_1)),
o_2



Transforming States to Tuples of Terms

Transforming D

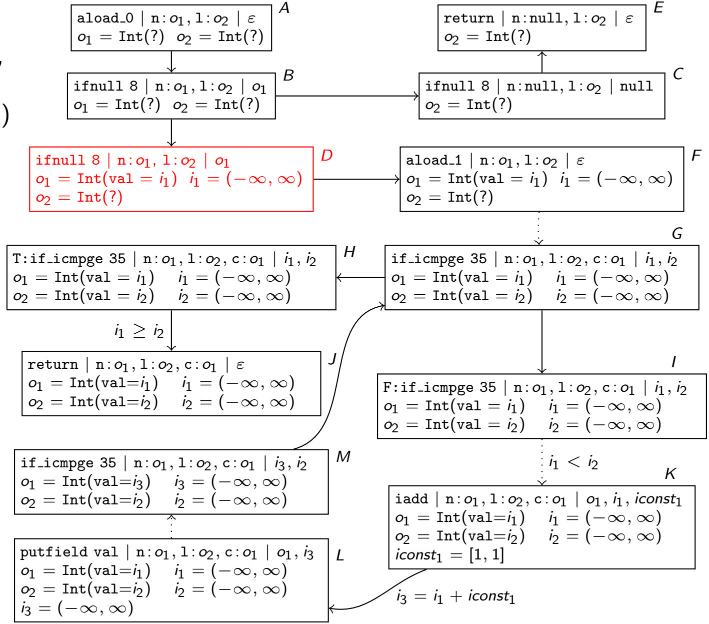
$\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1))$,
o_2,
$\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1))$



Transforming States to Tuples of Terms

Transforming D

$f_D($ jlO(Int(eoc, i_1)),
    o_2,
    jlO(Int(eoc, i_1)) $)$



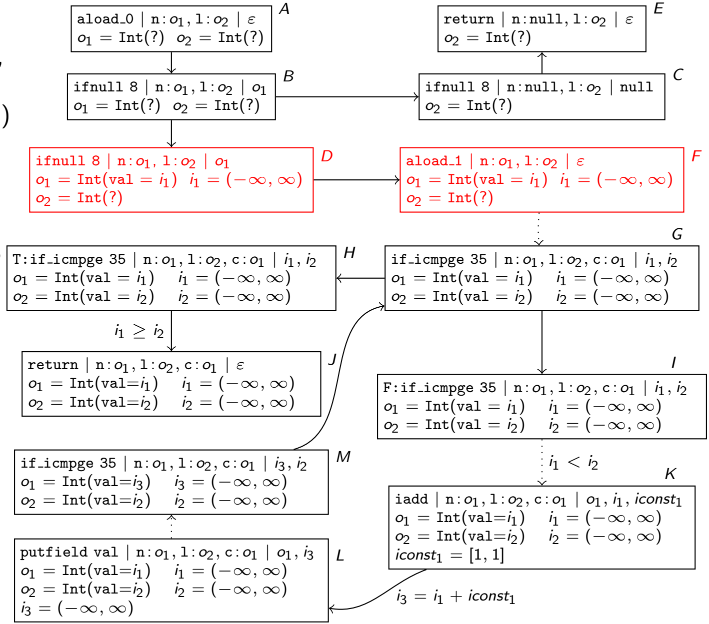
Transforming States to Tuples of Terms

Transforming D

$$f_D(\,\text{jlO}(\text{Int}(\text{eoc}, i_1)),\; o_2,\; \text{jlO}(\text{Int}(\text{eoc}, i_1))\,)$$

Transforming F

$$f_F(\,\text{jlO}(\text{Int}(\text{eoc}, i_1)),\; o_2\,)$$

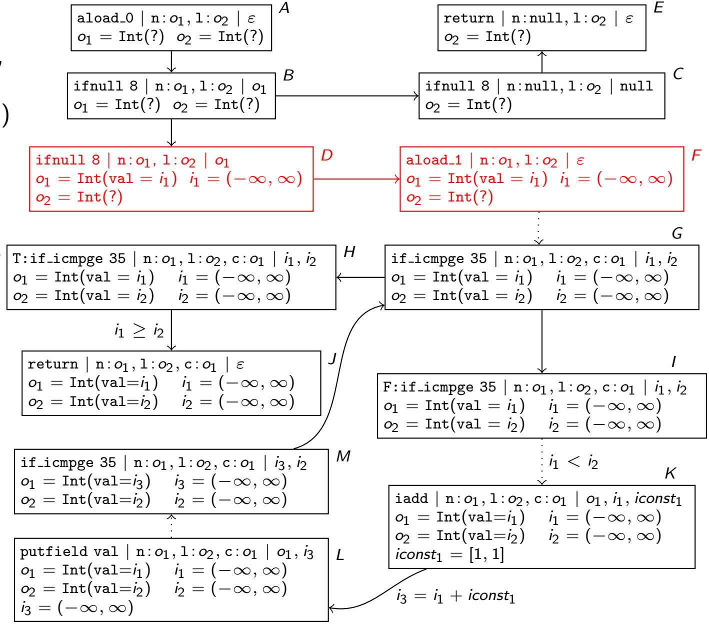


Transforming Edges to Rewrite Rules

$$f_D(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\ o_2,\ \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1))\,)$$

$$\rightarrow$$

$$f_F(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\ o_2\,)$$



Transforming Edges to Rewrite Rules

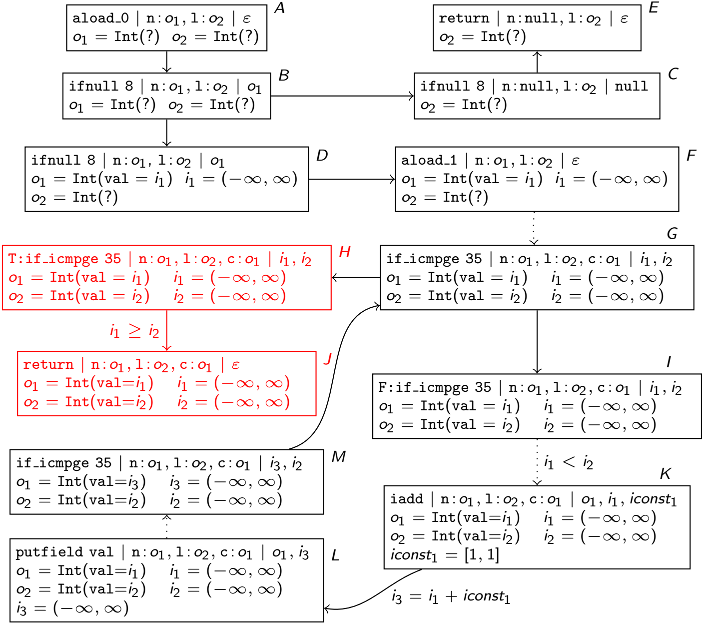
Transforming Evaluation Edges with Conditions

$$f_H(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\ \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_2)),\ \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\ i_1,\ i_2\,)$$

$$\rightarrow$$

$$f_J(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\ \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_2)),\ \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1))\,)$$

$$\mid\ i_1 \geq i_2$$



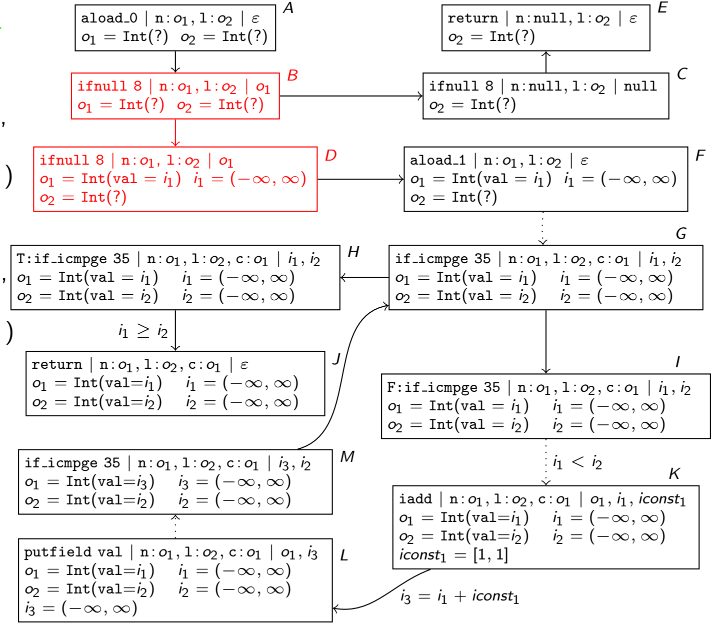
Transforming Edges to Rewrite Rules

Transforming Refinement Edges

$$f_B(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\; o_2,\; \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1))\,)$$

$$\longrightarrow$$

$$f_D(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\; o_2,\; \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1))\,)$$



Transforming Edges to Rewrite Rules

Merging Rewrite Rules

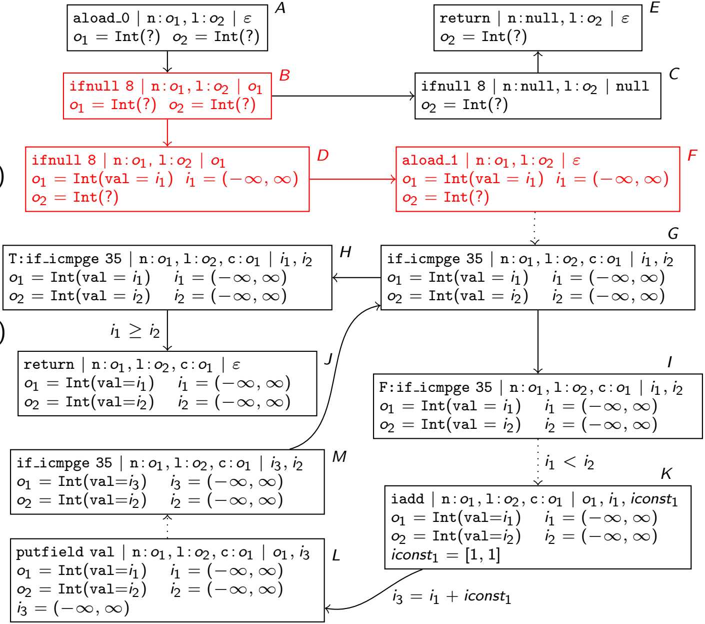
$$f_B(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\; o_2,\; \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1))\,)$$

$$\rightarrow$$

$$f_D(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\; o_2,\; \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1))\,)$$

$$\rightarrow$$

$$f_F(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\; o_2\,)$$



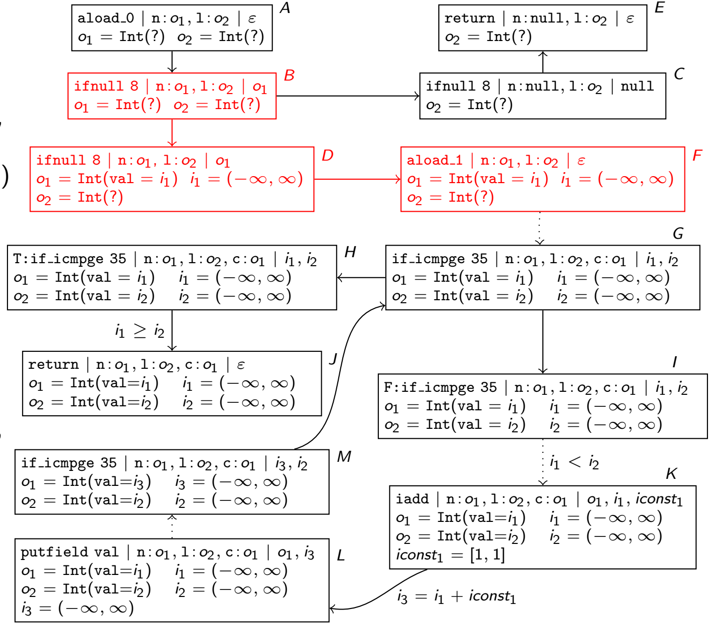
Transforming Edges to Rewrite Rules

Merging Rewrite Rules

$f_B(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),$
$\quad o_2,$
$\quad \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1))\,)$

$\rightarrow$

$f_F(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),$
$\quad o_2\,)$



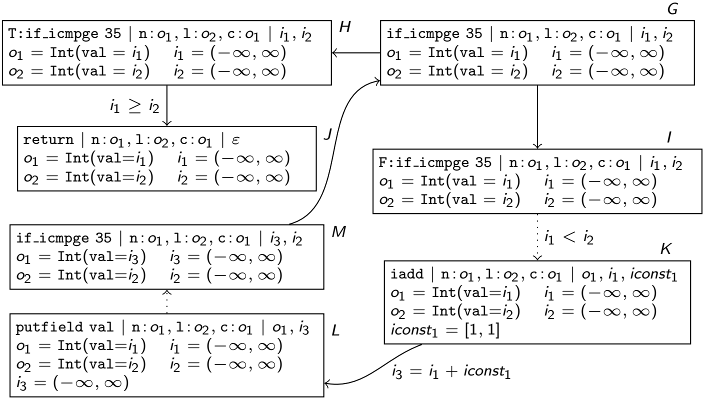
Transforming Edges to Rewrite Rules

TRS for count

$$
\begin{array}{l}
\mathsf{f}_G(\,\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)),\ \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_2)),\ \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)),\ i_1,\ i_2) \ \to \\
\mathsf{f}_G(\,\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1+1)),\ \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_2)),\ \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1+1)),\ i_1+1,\ i_2) \ \mid \ i_1 < i_2
\end{array}
$$

TRS is "*natural*"

termination easy to prove automatically



From Termination Graphs to TRSs

TRS for count

$$\begin{array}{lllllll} \mathsf{f}_G(\,\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)), & \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_2)), & \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)), & i_1, & i_2) & \rightarrow & \\ \mathsf{f}_G(\,\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1+1)), & \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_2)), & \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1+1)), & i_1+1, & i_2) & \mid & i_1 < i_2 \end{array}$$

- every JBC-computation of concrete states corresponds to a *computation path* in the termination graph
- termination graph is called *terminating* iff it has no infinite computation path
- every computation path corresponds to rewrite sequence in TRS

Theorem

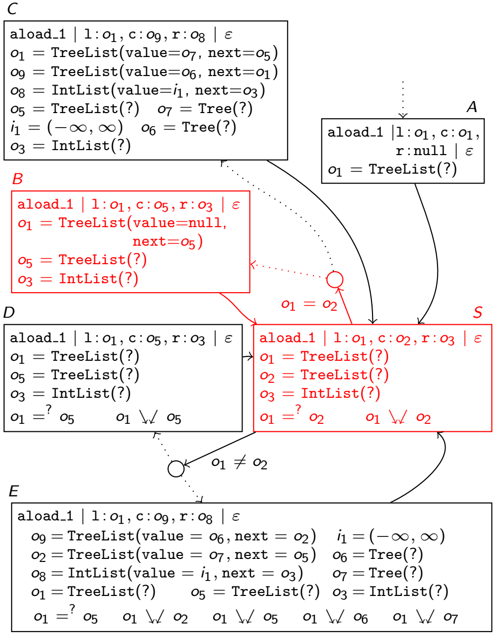
TRS corresponding to termination graph is terminating $\Rightarrow$

termination graph is terminating $\Rightarrow$

JBC-program terminating for all states represented in termination graph

From Termination Graphs to TRSs

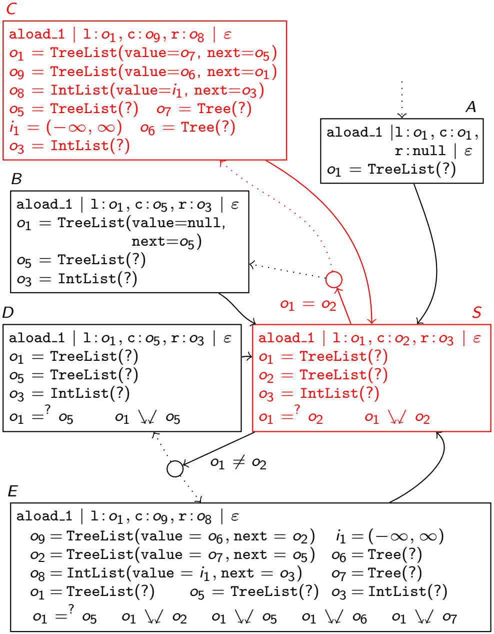
$$f_S(\mathsf{TL}(\mathsf{null}, o_5), \mathsf{TL}(\mathsf{null}, o_5), o_3) \to$$
$$f_S(\mathsf{TL}(\mathsf{null}, o_5), \quad o_5 \ , o_3)$$



From Termination Graphs to TRSs

$$f_S(\mathsf{TL}(\mathsf{null}, o_5), \mathsf{TL}(\mathsf{null}, o_5), o_3) \to$$
$$f_S(\mathsf{TL}(\mathsf{null}, o_5), \quad o_5 \ , o_3)$$

$$f_S(\ldots, \mathsf{TL}(\mathsf{T}(i_1, o_6, o_7), o_5), \quad o_3 \ ) \to$$
$$f_S(\ldots, \mathsf{TL}(\ o_6, \mathsf{TL}(o_7, o_5)), \mathsf{IL}(i_1, o_3))$$

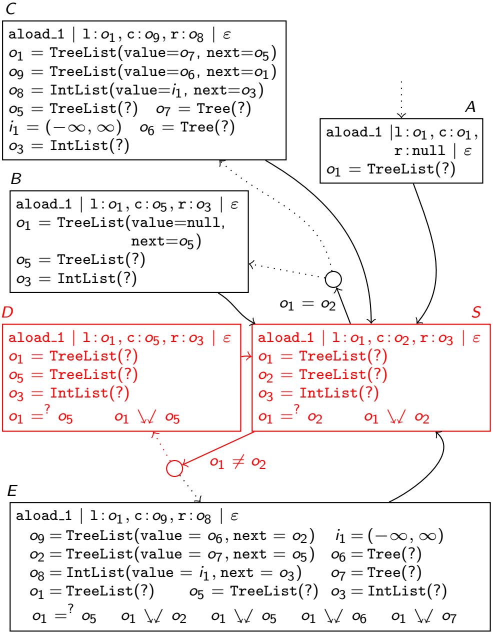


From Termination Graphs to TRSs

$$f_S(\mathsf{TL}(\mathsf{null}, o_5), \mathsf{TL}(\mathsf{null}, o_5), o_3) \to$$
$$f_S(\mathsf{TL}(\mathsf{null}, o_5), \quad o_5 \;, o_3)$$

$$f_S(\ldots, \mathsf{TL}(\mathsf{T}(i_1, o_6, o_7), o_5), \quad o_3 \;) \to$$
$$f_S(\ldots, \mathsf{TL}(\; o_6, \mathsf{TL}(o_7, o_5)), \mathsf{IL}(i_1, o_3))$$

$$f_S(o_1, \mathsf{TL}(\mathsf{null}, o_5), o_3) \to$$
$$f_S(o_1, \quad o_5 \;, o_3)$$



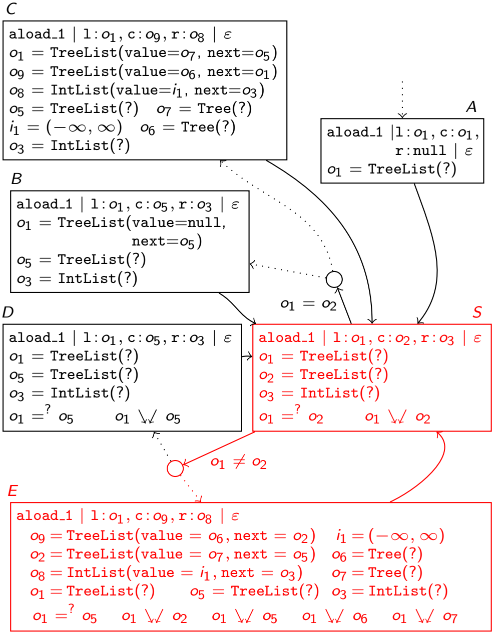
From Termination Graphs to TRSs

$$f_S(\mathsf{TL}(\mathsf{null}, o_5), \mathsf{TL}(\mathsf{null}, o_5), o_3) \to f_S(\mathsf{TL}(\mathsf{null}, o_5), o_5, o_3)$$

$$f_S(\ldots, \mathsf{TL}(\mathsf{T}(i_1, o_6, o_7), o_5), o_3) \to f_S(\ldots, \mathsf{TL}(o_6, \mathsf{TL}(o_7, o_5)), \mathsf{IL}(i_1, o_3))$$

$$f_S(o_1, \mathsf{TL}(\mathsf{null}, o_5), o_3) \to f_S(o_1, o_5, o_3)$$

$$f_S(o_1, \mathsf{TL}(\mathsf{T}(i_1, o_6, o_7), o_5), o_3) \to f_S(o_1', \mathsf{TL}(o_6, \mathsf{TL}(o_7, o_5)), \mathsf{IL}(i_1, o_3))$$



From Termination Graphs to TRSs

$$f_S(\mathsf{TL}(\mathsf{null}, o_5), \mathsf{TL}(\mathsf{null}, o_5), o_3) \to$$
$$f_S(\mathsf{TL}(\mathsf{null}, o_5), \quad o_5 \quad, o_3)$$

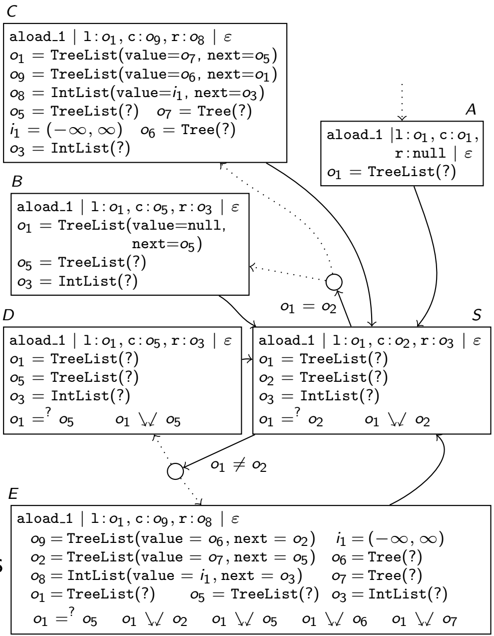
$$f_S(\ldots, \mathsf{TL}(\mathsf{T}(i_1, o_6, o_7), o_5), \quad o_3 \quad) \to$$
$$f_S(\ldots, \mathsf{TL}(\quad o_6, \mathsf{TL}(o_7, o_5)), \mathsf{IL}(i_1, o_3))$$

$$f_S(o_1, \mathsf{TL}(\mathsf{null}, o_5), o_3) \to$$
$$f_S(o_1, \quad o_5 \quad, o_3)$$

$$f_S(o_1, \mathsf{TL}(\mathsf{T}(i_1, o_6, o_7), o_5), \quad o_3 \quad) \to$$
$$f_S(o_1', \mathsf{TL}(\quad o_6, \mathsf{TL}(o_7, o_5)), \mathsf{IL}(i_1, o_3))$$

Rewrite Rules & Annotations

- when writing to a field of o_2 with $o_1 \searrow\!\!\swarrow o_2$: o_1 on lhs, fresh variable o_1' on rhs
- cyclic objects: fresh variable on rhs



From Termination Graphs to TRSs

$f_S(\mathsf{TL}(\mathsf{null}, o_5), \mathsf{TL}(\mathsf{null}, o_5), o_3) \to$

$f_S(\mathsf{TL}(\mathsf{null}, o_5), \quad o_5 \;, o_3)$

$f_S(\ldots, \mathsf{TL}(\mathsf{T}(i_1, o_6, o_7), o_5), \quad o_3 \;) \to$

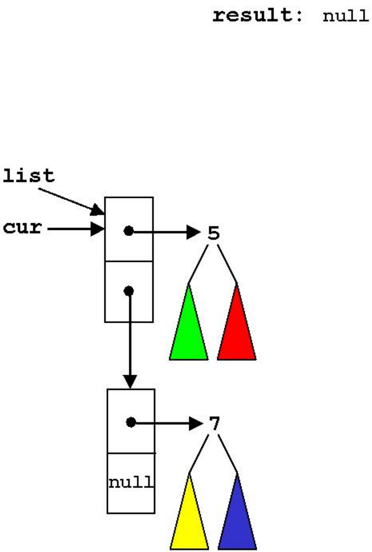
$f_S(\ldots, \mathsf{TL}(\; o_6, \mathsf{TL}(o_7, o_5)), \mathsf{IL}(i_1, o_3))$

$f_S(o_1, \mathsf{TL}(\mathsf{null}, o_5), o_3) \to$

$f_S(o_1, \quad o_5 \;, o_3)$

$f_S(o_1, \mathsf{TL}(\,\mathsf{T}(5, o_6, o_7), o_5), \quad \mathsf{null} \;) \to$
$f_S(o_1', \mathsf{TL}(\; o_6, \mathsf{TL}(o_7, o_5)), \mathsf{IL}(5, \mathsf{null}))$

TRS is “*natural*”



From Termination Graphs to TRSs

$f_S(\mathsf{TL}(\mathsf{null}, o_5), \mathsf{TL}(\mathsf{null}, o_5), o_3) \rightarrow$

$f_S(\mathsf{TL}(\mathsf{null}, o_5), \quad o_5 \ , o_3)$

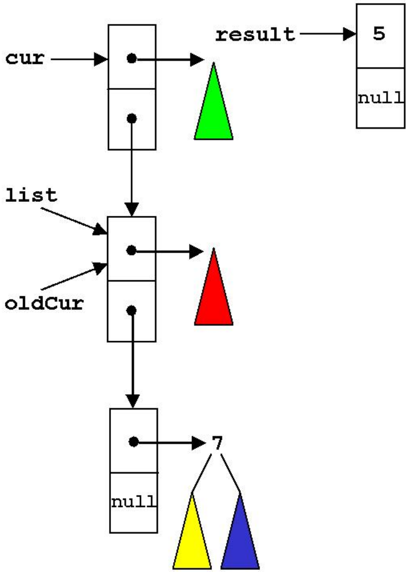
$f_S(\ldots, \mathsf{TL}(\mathsf{T}(i_1, o_6, o_7), o_5), \quad o_3 \ ) \rightarrow$

$f_S(\ldots, \mathsf{TL}(\ o_6, \mathsf{TL}(o_7, o_5)), \mathsf{IL}(i_1, o_3))$

$f_S(o_1, \mathsf{TL}(\mathsf{null}, o_5), o_3) \rightarrow$

$f_S(o_1, \quad o_5 \ , o_3)$

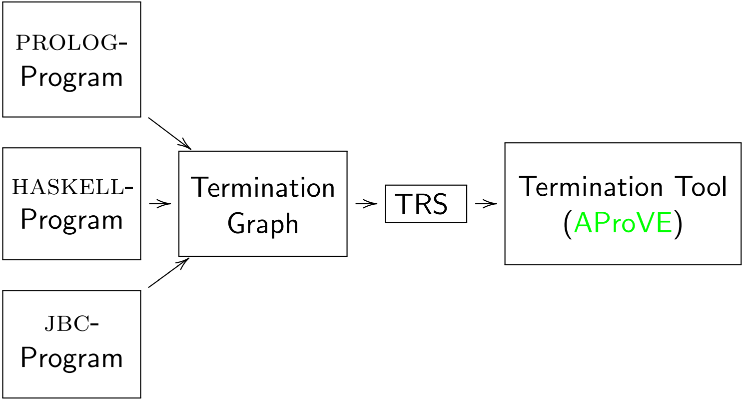
$f_S(o_1, \mathsf{TL}(\ \mathsf{T}(5, o_6, o_7), o_5), \quad \mathsf{null} \ ) \rightarrow$
$f_S(o_1', \mathsf{TL}(\ o_6, \mathsf{TL}(o_7, o_5)), \mathsf{IL}(5, \mathsf{null}))$



TRS is "*natural*"

termination easy
to prove automatically

Automated Termination Analysis of JAVA BYTECODE by Term Rewriting



Automated Termination Analysis of JAVA BYTECODE by Term Rewriting

- implemented in AProVE and evaluated on collection of 125 JBC-programs (*Termination Problem Data Base*)

	Success	Failure	Timeout	Runtime
AProVE	102	6	17	15.0
Julia	91	34	0	2.7
COSTA	72	52	1	4.5

- AProVE winner of the *International Termination Competition* 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