RWTH Aachen

Department of Computer Science
Technical Report

Automated Termination Analysis of Java Bytecode by Term Rewriting

Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2010-08

RWTH Aachen · Department of Computer Science · April 2010

AUTOMATED TERMINATION ANALYSIS OF JAVA BYTECODE BY TERM REWRITING

C. OTTO AND M. BROCKSCHMIDT AND C. VON ESSEN AND J. GIESL

LuFG Informatik 2, RWTH Aachen University, Germany

Abstract. We present an automated approach to prove termination of Java Bytecode (JBC) programs by automatically transforming them to term rewrite systems (TRSs). In this way, the numerous techniques and tools developed for TRS termination can now be used for imperative object-oriented languages like Java, which can be compiled into JBC.

1. Introduction

Termination of TRSs and logic programs has been studied for decades. But as imperative programs dominate in practice, recently many results on termination of imperative programs were developed as well (e.g., [2, 3, 4, 5, 12]). Our goal is to re-use the wealth of techniques and tools from TRS termination when tackling imperative object-oriented programs. Similar TRS-based approaches have already proved successful for termination analysis of Prolog and Haskell [10, 16]. A first approach to prove termination of imperative programs by transforming them to TRSs was presented in [7]. However, [7] only analyzes a toy programming language without heap, whereas our goal is to analyze JBC programs.

JBC [14] is an assembly-like object-oriented language designed as intermediate format for the execution of Java [11] programs by a Java Virtual Machine (JVM). Moreover, JBC is a common compilation target for many other languages besides Java. While there exist several static analysis techniques for JBC, we are only aware of two other automated methods to analyze termination of JBC, implemented in the tools COSTA [1] and Julia [18]. They transform JBC into a constraint logic program by abstracting every object of a dynamic data type to an integer denoting its path-length (i.e., the maximal length of the path of pointers that can be obtained by following the fields of objects). For example, consider a data structure `IntList` with the field `value` for the first list element and the field `next` which points to the next list element. Now an object of type `IntList` representing the list $[0, 1, 2]$ would be abstracted to its length 3, but one would disregard the values of the list elements. While this fixed mapping from data objects to integers leads to a very efficient analysis, it also restricts the power of these methods. In contrast, in our approach we represent data objects not by integers, but by *terms*. To this end, we introduce a function symbol for every class. So the `IntList` object above is represented by a term like

Key words and phrases: Java Bytecode, termination, term rewriting.

Supported by the DFG grant GI 274/5-2 and by the G.I.F. grant 966-116.6.

IntList(0, IntList(1, IntList(2, null))), which keeps the whole information of the data object.

So compared to [1, 18] and to direct termination analysis of imperative programs, rewrite techniques[1] have the advantage that they are very powerful for algorithms on user-defined data structures, since they can automatically generate suitable well-founded orders comparing arbitrary forms of terms. Moreover, by using TRSs with built-in integers [8], rewrite techniques are also powerful for algorithms on pre-defined data types like integers.

Inspired by our approach for termination of Haskell [10], in this paper we present a method to translate JBC programs to TRSs. More precisely, in Sect. 2 we show how to automatically construct a *termination graph* representing all execution paths of the JBC program. Similar graphs are also used in program optimization techniques, e.g. [17]. While we perform considerably less abstraction than [1, 18], we also apply a suitable abstract interpretation [6] in order to obtain finite representations for all possible forms of the heap at a certain state. In contrast to *control flow graphs*, the nodes of the termination graph contain not just the current program position, but also detailed information on the values of the variables and on the content of the heap. Thus, the termination graph usually has several nodes which represent the same program position, but where the values of the variables and the heap are different. This is caused by different runs through the program code. The termination graph takes care of all aliasing, sharing, and cyclicity effects in the JBC program. This is needed in order to express these effects in a TRS afterwards. Then, a TRS is generated from the termination graph such that termination of the TRS implies termination of the original JBC program (Sect. 3). The resulting TRSs can be handled by existing TRS termination techniques and tools.

As described in Sect. 4, we implemented the transformation in our tool AProVE [9]. In the first *International Termination Competition* on automated termination analysis of JBC, AProVE achieved competitive results compared to Julia and COSTA. So this paper shows for the first time that rewriting techniques can indeed be successfully used for termination of imperative object-oriented languages like Java.

2. From JBC to Termination Graphs

To obtain a finite representation of all execution paths, we evaluate the JBC program symbolically, resulting in a *termination graph*. Afterwards, this graph is used to generate a TRS suitable for termination analysis. Sect. 2.1 introduces the abstract states used in termination graphs. Then Sect. 2.2 illustrates the construction of termination graphs for simple programs and Sect. 2.3 extends it to programs with complex forms of sharing.

2.1. Representing States of the JVM

We define *abstract states* which represent *sets* of concrete JVM states, using a formalization which is especially suitable for a translation into TRSs (see e.g. [13] for related formalizations). Our approach is restricted to verified sequential JBC programs without recursion. To simplify the presentation in the paper, we only consider program runs involving a single

[1] Of course, one could also use a transformation similar to ours where JBC is transformed to (constraint) logic programs, but where data objects are also represented by terms instead of integers. In principle, such an approach would be as powerful as ours, provided that one uses sufficiently powerful underlying techniques for automated termination analysis of logic programs. However, since some of the most powerful current termination analyzers for logic programs are based on term rewriting [15, 16], it seems more natural to transform JBC to term rewriting directly.

method, and exclude floating point arithmetic, arrays, exceptions, and static class fields. However, our approach can easily be extended to such constructs and to arbitrary many non-recursive methods. For the latter, we represent the frames of the call stack individually and simply "inline" the code of invoked methods. Indeed, our implementation also handles programs with several methods including floats, arrays, exceptions, and static fields.

Definition 2.1. The set of abstract states is $\textsc{States} = \textsc{ProgPos} \times \textsc{LocVar} \times \textsc{OpStack} \times \textsc{Heap}$.

The first component of a state corresponds to the program counter. We represent it by the next program instruction to be executed (e.g., by a **JBC** instruction like "`ifnull 8`").

The second component is an array of the local variables which have a defined value at the current program position, represented by a partial function $\textsc{LocVar} = \mathbb{N} \to \textsc{References}$. Here, $\textsc{References}$ are addresses in the heap. So in our representation, we do not store primitive values directly, but indirectly using references to the heap. This enables us to retain equality information for two otherwise unknown primitive values. Moreover, we require `null` $\in \textsc{References}$ to represent the `null` reference. To ease readability, in examples we usually denote local variables by names instead of numbers. Thus, "$\mathtt{o} : o_1, \mathtt{l} : o_2$" denotes an array where the 0-th local variable `o` references the address o_1 in the heap and the 1-st local variable `l` references the address o_2 in the heap. Of course, different local variables can point to the same address (e.g., in "$\mathtt{o}:o_1, \mathtt{l}:o_2, \mathtt{c}:o_1$", `o` and `c` refer to the same object).

The third component is the operand stack that **JBC** instructions operate on. It will be filled with intermediate values such as operands of arithmetic operations when evaluating the bytecode. We represent it by a partial function $\textsc{OpStack} = \mathbb{N} \to \textsc{References}$. The empty operand stack is denoted by "ε" and "i_1, i_2" denotes a stack with top element i_2.

`ifnull 8` $\mid \mathtt{o}:o_1, \mathtt{l}:o_2 \mid o_1$
$o_1 = \mathtt{Int}(\mathtt{val} = i_1) \quad i_1 = (-\infty, \infty)$
$o_2 = \mathtt{Int}(?)$

Figure 1: An abstract **JVM** state

To depict abstract states in examples, we write the first three components in the first line and separate them by "|". The fourth $\textsc{Heap}$ component is written in the lines below, cf. Fig. 1. It describes the values of $\textsc{References}$. We represent the $\textsc{Heap}$ by a partial function $\textsc{Heap} : \textsc{References} \to \textsc{Integers} \cup \textsc{Instances} \cup \textsc{Unknown}$.

The values in $\textsc{Unknown} = \textsc{Classnames} \times \{?\}$ represent tree-shaped (and thus acyclic) objects for which we have no information except their type. $\textsc{Classnames}$ contains the names of all classes and interfaces of the program. So for a class `Int`, "$o_2 = \mathtt{Int}(?)$" means that the object at address o_2 is `null` or an instance of type `Int` (or a subtype of `Int`).

We represent integers as possibly unbounded intervals, i.e. $\textsc{Integers} = \{\{x \in \mathbb{Z} \mid a \leq x \leq b\} \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \leq b\}$. So $i_1 = (-\infty, \infty)$ means that any integer can be at the address i_1. Since current TRS termination tools cannot handle 32-bit `int`-numbers as in **JBC**, we treat `int` as the infinite set of all integers, i.e., we cannot handle problems related to overflows. Note that in **JBC**, `int` is also used for Boolean values.

To represent $\textsc{Instances}$ (i.e., objects) of some class, we describe the values of their fields, i.e., $\textsc{Instances} = \textsc{Classnames} \times (\textsc{FieldIdentifiers} \to \textsc{References})$. To prevent ambiguities, in general the $\textsc{FieldIdentifiers}$ also contain the respective class names. So if the class `Int` has the field `val` of type `int`, then "$o_1 = \mathtt{Int}(\mathtt{val} = i_1)$" means that at the address o_1, there is an instance of class `Int` and its field `val` references the address i_1 in the heap. Note that all sharing and aliasing must be explicitly represented in the abstract state. So since the state in Fig. 1 contains no sharing information for o_1 and o_2, o_1 and the references reachable from o_1 are disjoint from o_2 and from the references reachable from o_2.

```
00: aload_0       // load orig 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 orig
10: astore_2      // store into copy
11: aload_0       // load orig
12: getfield val  // load field val
15: aload_1       // load limit
16: getfield val  // load field val
19: if_icmpge 35  // jump if
                  // orig.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
```

(a) Java Bytecode

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

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

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

    // introduce sharing
    Int copy = orig;

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

(b) Java Source Code

Figure 2: Example using aliasing and an integer counting upwards

2.2. Termination Graphs for Simple Programs

We now introduce the *termination graph* using a simple example. In Fig. 2(a) we present the analyzed JBC program and Fig. 2(b) shows the corresponding Java source code.

We create the termination graph using the states of a run of our abstract virtual machine as nodes, starting in a suitable general state. In our example, we want to know if *all* calls of the method `count` with two distinct arbitrary `Int` objects (or `null`) as arguments terminate. Here it is important to handle the aliasing of the variables `copy` and `orig`.

In Fig. 3, node A contains the start state. For the local variables `orig` and `limit` (abbreviated `o` and `l`), we only know their type and we know that they do not share any part of the heap. The first JBC instruction `aload_0` loads the value of the 0-th local variable (the argument `orig`) on the operand stack. The variable `orig` references some address o_1 in the heap, but we do not need concrete information about o_1 for this instruction. The resulting new state B is connected to A by an *evaluation edge*.

To evaluate the `ifnull` instruction, we need to know if the reference on top of the operand stack is `null`. This is not yet known for o_1. We *refine* the information and create successor nodes C and D for all possible cases (i.e., for o_1 == `null`, and for `Int` and all its non-abstract subclasses). In C, o_1 is `null`, and in D it is an instance of `Int` (`Int` has no proper subtypes). In D, the field values are new references in the heap. So instead of "$o_1 = \texttt{Int}(?)$", we now have "$o_1 = \texttt{Int}(\texttt{val} = i_1)$". Note that while "$o_1 = \texttt{Int}(?)$" in node B means that if o_1 is not `null`, then it has type `Int` or a subtype of it, "$o_1 = \texttt{Int}(\texttt{val} = i_1)$" in node D means that o_1's type is exactly `Int` and not a proper subtype. We have no information about the value at i_1. Therefore, i_1 gets the most general value for Integers, i.e., $i_1 = (-\infty, \infty)$. C and D are connected to B by *refinement edges*.

Now we can evaluate the instruction both for C and D, leading to E and F. Evaluation stops in E, while for F, the same procedure is repeated for the argument `limit`, leading to node G (among others) after several steps, indicated by a dotted arrow. Note the aliasing between `copy` and `orig`, since both reference the same object at the address o_1.

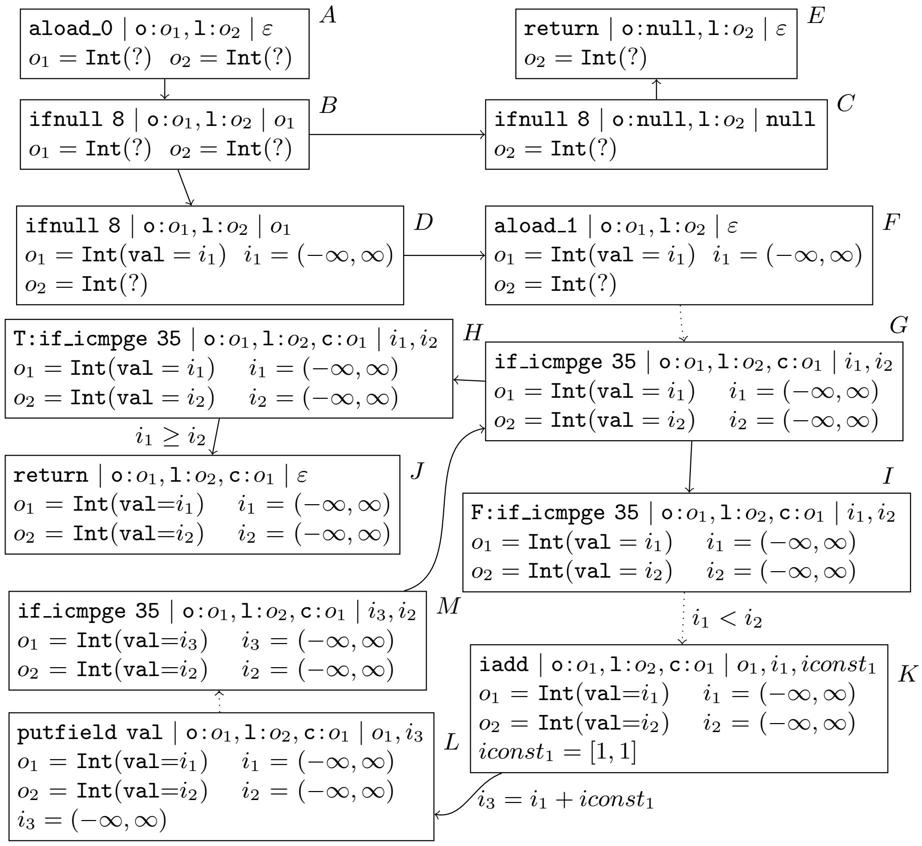

Figure 3: Termination graph for `count`

In G, we have already evaluated the two "`getfield val`" instructions and have pushed the two integer values on the operand stack. Now `if_icmpge` requires us to compare the unknown integers at i_1 and i_2. If we had to compare i_1 with a fixed number like 0, we could refine the information about i_1 and i_2 and create two successor nodes with $i_1 = (-\infty, -1]$ and $i_1 = [0, \infty)$. But "$i_1 \geq i_2$" is not expressible in our abstract states. Here, we split according to both possible values of the condition (depicted using the labels "`T`" and "`F`", respectively). This leads to the nodes H and I which are connected to G by *split edges*.

We can evaluate the condition in H to `true` and label the resulting evaluation edge to J by this condition. We will use these labels when constructing a TRS from the termination graph. J marks the program end and thus, it remains a leaf of the graph.

In I, we can evaluate the condition to `false` and label the next edge by the converse of the condition. After evaluating the next four instructions we reach node K. On the top positions of the operand stack, there are two integer variables (where the topmost variable has the value 1). The instruction `iadd` adds these two variables resulting in a new integer variable i_3. The relation between i_3, i_1, and $iconst_1$ is added as a label on the evaluation edge to the new node L. This label will again be used in the TRS construction.

From L on, we evaluate instructions until we again arrive at the instruction `if_icmpge` in node M. It turns out that M is an *instance* of the previous node G. Hence, we can connect M with G by an *instantiation edge*. The reason is that every concrete state which would be described by the abstract state M could also be described by the state G.

One has to expand termination graphs until all leaves correspond to program ends. Hence, our graph is now completed. By using appropriate generalization steps (which transform nodes into more general ones), one can always obtain a finite termination graph. To this end, one essentially executes the program symbolically until one reaches some position in the program for the second time. Then, a new state is created that is a generalization of both original states and one introduces instantiation edges from the two original states to the new generalized state. Of course, in our implementation we apply suitable heuristics to ensure that one only performs finitely many such generalization steps and to guarantee that the construction always terminates with a finite termination graph.

To define "*instance*" formally, we first define all positions π of references in a state s, where $s|_\pi$ denotes the reference at position π. A position π is a sequence starting with $\textsc{lv}_n$ or $\textsc{os}_n$ for some $n \in \mathbb{N}$ (indicating the n-th reference in the local variable array or in the operand stack), followed by zero or more $\textsc{FieldIdentifiers}$.

Definition 2.2 (position, $\textsc{SPos}$)**.** Let $s = (pp, l, op, h) \in \textsc{States}$. Then $\textsc{SPos}(s)$ is the smallest set such that one of the following holds for all $\pi \in \textsc{SPos}(s)$:

- $\pi = \textsc{lv}_n$ for some $n \in \mathbb{N}$ where $l(n)$ is defined. Then $s|_\pi$ is $l(n)$.
- $\pi = \textsc{os}_n$ for some $n \in \mathbb{N}$ where $op(n)$ is defined. Then $s|_\pi$ is $op(n)$.
- $\pi = \pi'\, v$ for some $v \in \textsc{FieldIdentifiers}$ and some $\pi' \in \textsc{SPos}(s)$ where $h(s|_{\pi'}) = (c, f) \in \textsc{Instances}$ and where $f(v)$ is defined. Then $s|_\pi$ is $f(v)$.

As an example, consider the state s depicted in node G of Fig. 3. Here we have three local variables and two elements on the operand stack. Thus, $\textsc{SPos}(s)$ contains $\textsc{lv}_0, \textsc{lv}_1, \textsc{lv}_2, \textsc{os}_0, \textsc{os}_1$, where $s|_{\textsc{lv}_0} = s|_{\textsc{lv}_2} = o_1$, $s|_{\textsc{lv}_1} = o_2$, $s|_{\textsc{os}_0} = i_1$, and $s|_{\textsc{os}_1} = i_2$. If h is the heap of that state, then $h(o_1) = (\texttt{Int}, f_1) \in \textsc{Instances}$, where $f_1(\texttt{val}) = i_1$. Hence, "$\textsc{lv}_0\,\texttt{val}$" is also a position in $\textsc{SPos}(s)$ and $s|_{\textsc{lv}_0\,\texttt{val}} = i_1$. The remaining elements of $\textsc{SPos}(s)$ are "$\textsc{lv}_2\,\texttt{val}$" and "$\textsc{lv}_1\,\texttt{val}$", where $s|_{\textsc{lv}_2\,\texttt{val}} = i_1$ and $s|_{\textsc{lv}_1\,\texttt{val}} = i_2$.

Intuitively, a state s' is an instance of a state s if they correspond to the same program position and whenever there is a reference $s'|_\pi$, then either the values represented by $s'|_\pi$ in the heap of s' are a subset of the values represented by $s|_\pi$ in the heap of s or else, π is no position in s. Moreover, shared parts of the heap in s' must also be shared in s. Note that since s and s' correspond to the same position in a *verified* JBC program, s and s' have the same number of local variables and their operand stacks have the same size.

Definition 2.3 (Instance)**.** We say that $s' = (pp', l', op', h')$ is an *instance* of state $s = (pp, l, op, h)$ (denoted $s' \sqsubseteq s$) iff $pp = pp'$, and for all $\pi, \pi' \in \textsc{SPos}(s')$:

(a) if $s'|_\pi = s'|_{\pi'}$ and $h'(s'|_\pi) \in \textsc{Instances} \cup \textsc{Unknown}$, then $\pi, \pi' \in \textsc{SPos}(s)$ and $s|_\pi = s|_{\pi'}$

(b) if $s'|_\pi \neq s'|_{\pi'}$ and $\pi, \pi' \in \textsc{SPos}(s)$, then $s|_\pi \neq s|_{\pi'}$

(c) if $h'(s'|_\pi) \in \textsc{Integers}$ and $\pi \in \textsc{SPos}(s)$, then $h(s|_\pi) \in \textsc{Integers}$ and $h'(s'|_\pi) \subseteq h(s|_\pi)$

(d) if $s'|_\pi = \texttt{null}$ and $\pi \in \textsc{SPos}(s)$, then $s|_\pi = \texttt{null}$ or $h(s|_\pi) = (c, ?) \in \textsc{Unknown}$

(e) if $h'(s'|_\pi) = (c', ?)$ and $\pi \in \textsc{SPos}(s)$, then $h(s|_\pi) = (c, ?)$ where c' is c or a subtype of c

(f) if $h'(s'|_\pi) = (c', f') \in \textsc{Instances}$ and $\pi \in \textsc{SPos}(s)$, then $h(s|_\pi) = (c', f) \in \textsc{Instances}$ or $h(s|_\pi) = (c, ?)$, where c' must be c or a subtype of c.

The state s' in node M of Fig. 3 is an instance of the state s in node G. Clearly, they both refer to the same program position. It remains to examine the references reachable in s'. We have $\textsc{SPos}(s') = \textsc{SPos}(s) = \{\textsc{lv}_0, \textsc{lv}_1, \textsc{lv}_2, \textsc{os}_0, \textsc{os}_1, \textsc{lv}_0\,\texttt{val}, \textsc{lv}_1\,\texttt{val}, \textsc{lv}_2\,\texttt{val}\}$. It is easy to check that the conditions of Def. 2.3 are satisfied for all these positions π. We illustrate this for $\pi = \textsc{lv}_0\,\texttt{val}$. Here, $s'|_\pi = i_3$ and if h' is the heap of s', then $h'(i_3) =$

$(-\infty,\infty)$. Similarly, $s|_\pi = i_1$ and if h is the heap of s, then $h(i_1) = (-\infty,\infty)$. Here, s' and s are in fact equivalent, since M is an instance of G and G is an instance of M.

`if_icmpge35` | `o`:o_1, `l`:o_2, `c`:o_1 | i_1, i_2	
o_1 = `Int(val=`i_1`)`	$i_1 = [1,1]$
o_2 = `Int(val=`i_2`)`	$i_2 = [10000, 10000]$

Figure 4: A concrete state

As remarked before, abstract states describe sets of *concrete states* like the one in Fig. 4, which is an instance of G and M. Here, the values for i_1 and i_2 are proper integers instead of intervals.

Definition 2.4 (Concrete state)**.** A state $s = (pp, l, op, h)$ is *concrete* if for all $\pi \in \text{SPos}(s)$:

- $h(s|_\pi) \notin$ Unknown and
- if $h(s|_\pi) \in$ Integers, then $h(s|_\pi)$ is just a singleton interval $[i,i]$ for some $i \in \mathbb{Z}$

A concrete state has no proper instances (i.e., if s is concrete and $s' \sqsubseteq s$, then $s \sqsubseteq s'$). Concrete states that are not a program end can always be evaluated and have exactly one (concrete) successor state. For Fig. 4, since i_1's value is not greater or equal than i_2's, the successor state corresponds to the instruction "`aload_2`", with the same local variables and empty operand stack. Such a sequence of concrete states, obtained by JBC evaluation, is called a *computation sequence*. Our construction of termination graphs ensures that

> if s is an abstract state in the termination graph and there is a concrete state $t \sqsubseteq s$ where t evaluates to the concrete state t', then the termination graph contains a path from s to a state s' with $t' \sqsubseteq s'$. (2.1)

To see why (2.1) holds, note that in the termination graph, s is first refined to a state $\overline{s}$ with $t \sqsubseteq \overline{s}$. So there is a path from s to $\overline{s}$, and in the state $\overline{s}$, all concrete information needed for an actual evaluation according to the JBC specification [14] is available. Note that "evaluation edges" in the termination graph are defined by exactly following the specification of JBC in [14]. Thus, there is an evaluation edge from $\overline{s}$ to s', where $t' \sqsubseteq s'$.

The computation sequence from Fig. 4 to its concrete successor corresponds to the path from node M or G to I's successor. Paths in the graph that correspond to computation sequences are called *computation paths*. Our goal is to show that all these paths are finite.

Definition 2.5 (Graph termination)**.** A finite or infinite path $s_1^1, \dots, s_1^{n_1}, s_2^1, \dots, s_2^{n_2}, \dots$ through the termination graph is called a *computation path* iff there is a computation sequence $t_1, t_2, \dots$ of concrete states where $t_i \sqsubseteq s_i^1$ for all i. A termination graph is called *terminating* iff it has no infinite computation path. Note that due to (2.1), if the termination graph is terminating, then the original JBC program is also terminating for all concrete states t where $t \sqsubseteq s$ for some abstract state s in the termination graph.

2.3. Termination Graphs for Complex Programs

Now we discuss sharing problems in complex programs with recursive data types. In Fig. 5, `flatten` takes a list of binary trees whose nodes are labeled by integers. It performs a depth-first run through all trees and returns the list of all numbers in these trees. It terminates because each loop iteration decreases the total number of all nodes in the trees of `list`, even though `list`'s length may increase. Note that `list` and `cur` share part of the heap.

Consider the three states A, B, and C in Fig. 6. A is the state of our abstract JVM when it first reaches the loop condition "`cur != null`" (where `list`, `cur`, and `result` are abbreviated by `l`, `c`, and `r`). After one execution of the loop body, one obtains the state B if `tree` is `null` and C otherwise. Note that local variables declared in the loop body are no longer defined at the loop condition, and hence, they do not occur in A, B, or C.

```
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;
}
```

Figure 5: Example converting a list of binary trees to a list of integers

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

B: `aload_1` | `l`:o_1, `c`:o_3, `r`:`null` | ε
o_1 = `TreeList`(`value`=`null`, `next`=o_3)
o_3 = `TreeList`(?)

C: `aload_1` | `l`:o_1, `c`:o_7, `r`:o_6 | ε
o_1 = `TreeList`(`value`=o_5, `next`=o_3)
o_3 = `TreeList`(?) o_5 = `Tree`(?)
o_6 = `IntList`(`value`=i_1, `next`=`null`)
o_7 = `TreeList`(`value`=o_4, `next`=o_1)
$i_1 = (-\infty, \infty)$ o_4 = `Tree`(?)

Figure 6: Three states of the termination graph of `flatten`

If one continued the evaluation like this, one would obtain an infinite tree, since one never reaches any state which is an instance of a previous state. (In particular, B and C are no instances of A.) Hence, to obtain *finite* graphs, one sometimes has to *generalize* states. Thus, we want to create a new general state S such that A, B, and C are instances of S. Note that in S, `l` and `c` cannot point to different references with Unknown values, since then S would only represent states where `l` and `c` are tree-shaped and not sharing. However, `l` and `c` point to the *same* object in A, one can reach `c`:o_3 from `l`:o_1 in B (i.e., `l` *joins* `c`, since a field value of o_1 is o_3), and one can reach `l`:o_1 from `c`:o_7 in C. To express such sharing information in general states, we extend states by *annotations*.

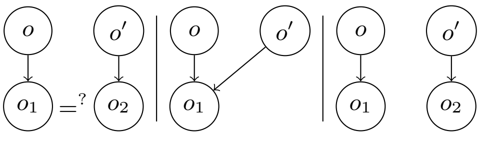

Figure 7: "$=^?$" annotation

In Fig. 7, the leftmost picture depicts a heap where an instance referenced by o has a field value o_1 and o' has a field value o_2. The annotation "$o_1 =^? o_2$" means that o_1 and o_2 could be equal. Here the value of at least one of o_1 and o_2 must be Unknown. So both the second and the third shape in Fig. 7 are instances of the first. In the second shape, o_1 and o_2 are equal and all occurrences of o_2 can be replaced by o_1 (or vice versa). In the third shape, o_1 and o_2 are not the same and thus, the annotation has been removed.

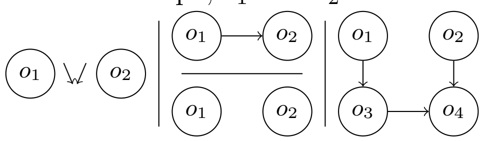

Figure 8: "$\searrow\!\swarrow$" annotation

So the $=^?$ annotation covers both the equality of `l` and `c` in state A and their non-equality in states B and C. To represent states where `l` and `c` *may join*, we use the annotation "$\searrow\!\swarrow$". We say that a reference o' is a *direct successor* of a reference o (denoted $o \to o'$) iff the object at address o has a field whose value is o'. As an example, consider state B in Fig. 6, where $o_1 \to o_3$ holds. Then the annotation "$o_1 \searrow\!\!\swarrow o_2$" means that if o_1 is UNKNOWN, then there could be an object o with $o_1 \to^+ o$ and $o_2 \to^* o$, i.e., o is a proper successor of o_1 and a (possibly non-proper) successor of o_2. Note that $\searrow\!\!\swarrow$ is symmetric,[2] so $o_1 \searrow\!\!\swarrow o_2$ also means that if o_2 is UNKNOWN, then there could be an object o' with $o_1 \to^* o'$ and $o_2 \to^+ o'$. The shapes 2-4 in Fig. 8 visualize three possible instances of the state with annotation "$o_1 \searrow\!\!\swarrow o_2$". Note that a state in which o_1 and o_2 do not share is also an instance.

We can now create a state S (see Fig. 9) such that $A, B, C \sqsubseteq S$. The annotations state that `l` and `c` may be equal (as in A), that `l` may join `c` (as in B), or `c` may join `l` (as in C).

So to obtain a finite termination graph, after reaching A, we generalize it to a new node S connected by an instantiation edge. As seen in D, we introduce new forms of refinement edges to refine a state with the annotation "$o_1 =^? o_7$" into the two instances where $o_1 = o_7$ and where $o_1 \neq o_7$. For $o_1 = o_7$, we reach B' and C' which are like B and C but now `r` points to a list ending with o_6 instead of `null`. The nodes B' and C' are connected back to S with instantiation edges. For $o_1 \neq o_7$, due to `c != null`, we first refine the information about o_7, and obtain $o_7 = \texttt{TreeList}(\texttt{value} = o_8, \texttt{next} = o_9)$. Note that "$\searrow\!\!\swarrow$" annotations have to be updated during refinements. If we have the annotation "$o_1 \searrow\!\!\swarrow o_7$" and if one refines o_7 by introducing references like o_8, o_9 for its non-primitive fields, then we have to add corresponding annotations such as "$o_1 =^? o_9$" for all field references like o_9 whose types correspond to the type of o_1. Moreover, we add "$\searrow\!\!\swarrow$" annotations for all non-primitive field references (i.e., "$o_1 \searrow\!\!\swarrow o_8$" and "$o_1 \searrow\!\!\swarrow o_9$"). If after this refinement neither o_1 nor o_7 were UNKNOWN, we would delete the annotation $o_1 \searrow\!\!\swarrow o_7$ since it has no effect anymore.

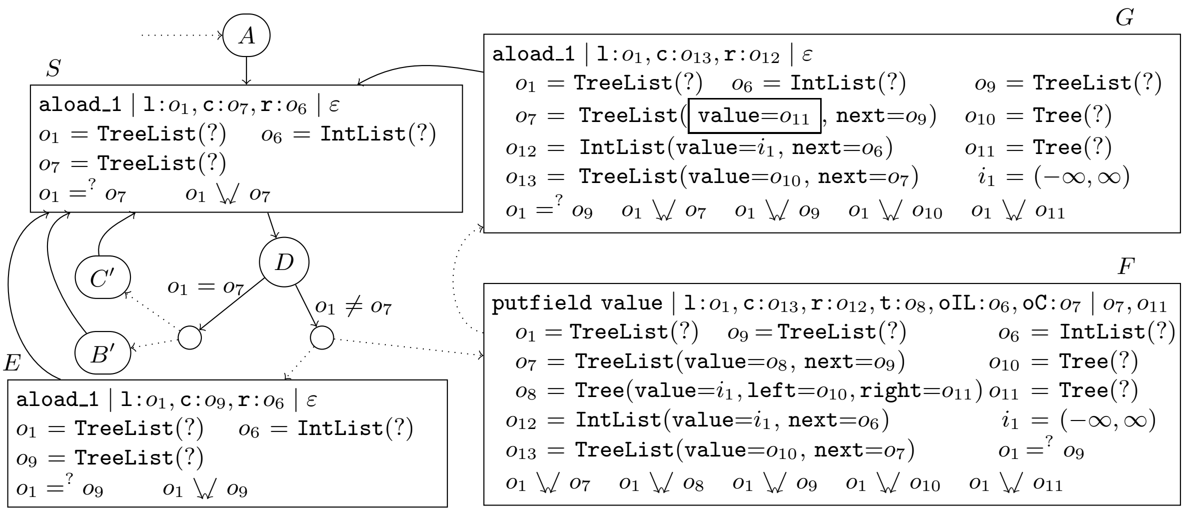

Figure 9: Termination graph for `flatten`

Now we use a refinement that corresponds to the case analysis whether `tree` is `null`. For `tree == null`, after one loop iteration we reach node E which is again an instance of S. Here, the local variable `tree` is no longer visible.

---

[2] Since both "$=^?$" and "$\searrow\!\!\swarrow$" are symmetric, we do not distinguish between "$o_1 =^? o_2$" and "$o_2 =^? o_1$" and we also do not distinguish between "$o_1 \searrow\!\!\swarrow o_2$" and "$o_2 \searrow\!\!\swarrow o_1$".

For `tree != null`, the graph shows nodes F and G. In F we need to evaluate a `putfield` instruction (corresponding to "`oldCur.value = tree.right`"), i.e., we have to put the object at address o_{11} to the field `value` of the object at address o_7. The effect of this operation can be seen in the box in state G, where the value of the object at o_7 was changed from o_8 to o_{11}. In G (which again corresponds to the loop condition), we removed the reference o_8 since it is no longer accessible from the local variables or the operand stack.

In contrast to other evaluation steps, such `putfield` instructions can give rise to additional annotations, since objects that already shared parts of the heap with o_7 now may also share parts of the heap with o_{11}. We say that a reference o *reaches* a reference o' iff there is a successor r of o (i.e., $o \to^* r$) such that $r = o'$ or $r =^? o'$ or $r \searrow\!\!\swarrow o'$. So in our example, o_{11} reaches just o_{11} and o_1. Now if we write o_{11} to a field of o_7, then for all references o with $o \searrow\!\!\swarrow o_7$, we have to add the annotation $o \searrow\!\!\swarrow o'$ for all o' where o_{11} reaches o'. Hence, in our example, we would have to add $o_1 \searrow\!\!\swarrow o_{11}$ (which is already present in the state) and $o_1 \searrow\!\!\swarrow o_1$. However, the annotation $o_1 \searrow\!\!\swarrow o_1$ has no effect, since by "$o_1 = \texttt{TreeList}(?)$", we know that o_1 only represents tree-shaped objects. Therefore, we can immediately drop $o_1 \searrow\!\!\swarrow o_1$ from the state. Concrete non-tree-shaped objects can of course be represented easily (e.g., "$o = \texttt{TreeList}(\texttt{value} = o', \texttt{next} = o)$"). But to represent an *arbitrary* possibly non-tree-shaped object o, we use a special annotation (depicted "$o!$").[3]

Definition 2.6 (Instance & annotations)**.** We extend Def. 2.3 to $s, s' = (pp', l', op', h') \in$ States possibly containing annotations. Now $s' \sqsubseteq s$ holds iff for all $\pi, \pi' \in \mathrm{SPos}(s')$, the following conditions are satisfied in addition to Def. 2.3 (b)-(f). Here, let τ resp. τ' be the maximal prefix of π resp. π' such that both $\tau, \tau' \in \mathrm{SPos}(s)$.

(a) if $s'|_\pi = s'|_{\pi'}$ where $h'(s'|_\pi) \in \mathrm{Instances} \cup \mathrm{Unknown}$, and if $\pi, \pi' \in \mathrm{SPos}(s)$,[4] then $s|_\pi = s|_{\pi'}$ or $s|_\pi =^? s|_{\pi'}$

(b) if $s'|_\pi =^? s'|_{\pi'}$ and $\pi, \pi' \in \mathrm{SPos}(s)$, then $s|_\pi =^? s|_{\pi'}$

(c) if $\big(s'|_\pi = s'|_{\pi'}$ where $h'(s'|_\pi) \in \mathrm{Instances} \cup \mathrm{Unknown}$, or $s'|_\pi =^? s'|_{\pi'}\big)$ and π or $\pi' \notin \mathrm{SPos}(s)$ with $\pi \neq \pi'$, then $s|_\tau \searrow\!\!\swarrow s|_{\tau'}$

(d) if $s'|_\pi \searrow\!\!\swarrow s'|_{\pi'}$, then $s|_\tau \searrow\!\!\swarrow s|_{\tau'}$

(e) if $s'|_\pi!$ holds, then $s|_\tau!$

(f) if there exist (possibly empty) sequences $\rho \neq \rho'$ of FieldIdentifiers without common prefix, where $s'|_{\pi\rho} = s'|_{\pi\rho'}$, $h'(s'|_{\pi\rho}) \in \mathrm{Instances} \cup \mathrm{Unknown}$, and ($\pi\rho$ or $\pi\rho' \notin \mathrm{SPos}(s)$ or $s|_{\pi\rho} =^? s|_{\pi\rho'}$), then $s|_\tau!$

3. From Termination Graphs to TRSs

Now we transform termination graphs into *integer term rewrite systems (ITRSs)*. These are TRSs where the Booleans $\mathbb{B}$, the integers $\mathbb{Z}$, and their built-in operations $\mathcal{A}rith\mathcal{O}p = \{+, -, *, /, \%, \ll, \gg, \ggg, \hat{}, \&, |\}$ and $\mathcal{R}el\mathcal{O}p = \{>, \geq, <, \leq, ==, \neq\}$ are pre-defined by an infinite set of variable-free rules $\mathcal{PD}$. For example, $\mathcal{PD}$ contains $\mathsf{1}+\mathsf{2} \to \mathsf{3}$ and $\mathsf{5} < \mathsf{4} \to \mathsf{false}$. As shown in [8], TRS termination techniques can easily be adapted to ITRSs as well.

[3] Such annotations can also result from `putfield` operations which write a reference o_2 to a field `f` of o_1. If o_1 already reached o_2 before through some field $\texttt{g} \neq \texttt{f}$, then we add "$o_1!$", since o_1 is no longer a tree. Even worse, if o_2 reached o_1 before, then `putfield` creates a cyclic object and we add "$o_1!$" and "$o_2!$".

[4] In contrast to Def. 2.3(a), here one may allow that π or $\pi' \notin \mathrm{SPos}(s)$. This case is handled in (c).

Definition 3.1 (ITRS [8])**.** An *ITRS* is a finite conditional TRS with rules "$\ell \to r \mid b$". Here ℓ, r, b are terms, where $\ell \notin \mathbb{B} \cup \mathbb{Z}$ and ℓ contains no symbol from $\mathcal{A}rith\mathcal{O}p \cup \mathcal{R}el\mathcal{O}p$. However, b and r may contain extra variables not occurring in ℓ. We often omit the condition b if b is true. The *rewrite relation* $\hookrightarrow_{\mathcal{R}}$ of an ITRS $\mathcal{R}$ is the smallest relation where $t_1 \hookrightarrow_{\mathcal{R}} t_2$ iff there is a rule $\ell \to r \mid b$ from $\mathcal{R} \cup \mathcal{PD}$ such that $t_1|_p = \ell\sigma$, $b\sigma \hookrightarrow_{\mathcal{R}}^* \mathsf{true}$, and $t_2 = t_1[r\sigma]_p$. Here, $\ell\sigma$ must not have instances of left-hand sides of rules as proper subterms, and σ must be *normal* (i.e., $\sigma(y)$ is in normal form also for variables y occurring only in b or r). Thus, the rewrite relation $\hookrightarrow_{\mathcal{R}}$ corresponds to an *innermost* evaluation strategy.

So if $\mathcal{R}$ contains the rule "$\mathsf{f}(x) \to \mathsf{g}(x, y) \mid x > 2$", then $\mathsf{f}(1 + 2) \hookrightarrow_{\mathcal{R}} \mathsf{f}(3) \hookrightarrow_{\mathcal{R}} \mathsf{g}(3, 27)$. Hence, extra variables in conditions or right-hand sides of rules stand for arbitrary values.

We first show how to transform a reference o in a state s into a term $\mathrm{tr}(s, o)$. References pointing to concrete integers like $iconst_1 = [1, 1]$ in state K of Fig. 3 are transformed into the corresponding integer constant $\mathsf{1}$. The reference `null` is transformed into the constant null. References pointing to instances will be transformed by a refined transformation $\mathrm{ti}(s, o)$ in a more subtle way in order to take their types and the values of their fields into account. Finally, any other reference o is transformed into a variable (which we also call o). So $i_1 = (-\infty, \infty)$ in state K of Fig. 3 is transformed to the variable i_1.

Definition 3.2 (Transforming references)**.** Let $s = (pp, l, op, h) \in \textsc{States}, o \in \textsc{References}$.

$$\mathrm{tr}(s, o) = \begin{cases} i & \text{if } h(o) = [i, i], \text{ where } i \in \mathbb{Z} \\ \mathsf{null} & \text{if } o = \texttt{null} \\ \mathrm{ti}(s, o) & \text{if } h(o) \in \textsc{Instances} \\ o & \text{otherwise} \end{cases}$$

The main advantage of our approach becomes obvious when transforming instances (i.e., data objects) into terms. The reason is that data objects essentially *are* terms and we simply keep their structure when transforming them. So for any object, we use the name of its class as a function symbol. The arguments of the function symbol correspond to the fields of the class. As an example, consider o_{13} in state F of Fig. 9. This data object is transformed to the term $\mathsf{TreeList}(o_{10}, \mathsf{TreeList}(\mathsf{Tree}(i_1, o_{10}, o_{11}), o_9))$.

However, we also have to take the class hierarchy into account. Therefore, for any class c with n fields, let the corresponding function symbol now have arity $n+1$. The arguments $2, \ldots, n + 1$ correspond to the values of the fields declared in class c. The first argument represents the part of the object that corresponds to subclasses of c. As an example, consider a class `A` with a field `a` of type `int` and a class `B` which extends `A` and has a field `b` of type `int`. If `x` is a data object of type `A` where `x.a` is `1`, then we now represent it by the term $\mathsf{A}(\mathsf{eoc}, 1)$. Here, the first argument of A is a constant eoc (for "end of class") which indicates that the type of `x` is really `A` and not a subtype of `A`. If `y` is a data object of type `B` where `y.a` is `2` and `y.b` is `3`, then we represent it by the term $\mathsf{A}(\mathsf{B}(\mathsf{eoc}, 3), 2)$. So the class hierarchy is represented by nesting the function symbols corresponding to the respective classes.

More precisely, since every class extends `java.lang.Object` (which has no fields), each such term now has the form $\mathsf{java.lang.Object}(\ldots)$. Hence, if we abbreviate the function symbol $\mathsf{java.lang.Object}$ by jlO, then for the `TreeList` object above, now the corresponding term is $\mathsf{jlO}(\mathsf{TreeList}(\mathsf{eoc}, o_{10}, \mathsf{jlO}(\mathsf{TreeList}(\mathsf{eoc}, \mathsf{jlO}(\mathsf{Tree}(\mathsf{eoc}, i_1, o_{10}, o_{11}), o_9)))))$.

Of course, we can only transform tree-shaped objects to terms. If $\pi \in \textsc{SPos}(s)$ and if there is a non-empty sequence ρ of $\textsc{FieldIdentifiers}$ such that $s|_\pi = s|_{\pi\rho}$, then $s|_\pi$ is called *cyclic* in s. If $s|_\pi$ is cyclic or marked by "!", then $s|_\pi$ is called *special*. Every special reference o is transformed into a variable o in order to represent an "arbitrary unknown" object. To define the transformation $\mathrm{ti}(s,o)$ formally, we use an auxiliary transformation $\overline{\mathrm{ti}}(s,o,c)$ which only considers the part of the class hierarchy starting with c.

Definition 3.3 (Transforming instances)**.** We start the construction at the root of the class hierarchy (i.e., with `java.lang.Object`) and define $\mathrm{ti}(s,o) = \overline{\mathrm{ti}}(s,o,\texttt{java.lang.Object})$.

Let $s = (pp, l, op, h) \in \textsc{States}$ and let $h(o) = (c_o, f) \in \textsc{Instances}$. Let $(c_1 = \texttt{java.lang.Object}, c_2, \ldots, c_n = c_o)$ be ordered according to the class hierarchy, i.e., c_i is the direct superclass of c_{i+1}. We define the term $\overline{\mathrm{ti}}(s,o,c_i)$ as follows:

$$\overline{\mathrm{ti}}(s,o,c_i) = \begin{cases} o & \text{if } o \text{ is special} \\ c_o(\mathsf{eoc}, \mathrm{tr}(s,v_1), \ldots, \mathrm{tr}(s,v_m)) & \text{if } c_i = c_o, \mathrm{fv}(c_o,f) = v_1,\ldots,v_m \\ c_i(\overline{\mathrm{ti}}(s,o,c_{i+1}), \mathrm{tr}(s,v_1), \ldots, \mathrm{tr}(s,v_m)) & \text{if } c_i \neq c_o, \mathrm{fv}(c_i,f) = v_1,\ldots,v_m \end{cases}$$

For $c \in \textsc{Classnames}$, let `f1`, ..., `fm` be the fields declared in c in some fixed order. Then for $f : \textsc{FieldIdentifiers} \to \textsc{References}$, $\mathrm{fv}(c,f)$ gives the values $f(\texttt{f1}), \ldots, f(\texttt{fm})$.

So for class `A` and `B` above, if $h(o) = (\texttt{B}, f)$ where $f(\mathsf{a}) = [2,2]$ and $f(\mathsf{b}) = [3,3]$, then $\mathrm{fv}(\texttt{A}, f) = [2,2]$, $\mathrm{fv}(\texttt{B}, f) = [3,3]$ and thus, $\overline{\mathrm{ti}}(s,o,\texttt{java.lang.Object}) = \mathsf{jlO}(\mathsf{A}(\mathsf{B}(\mathsf{eoc},3),2))$.

To transform a whole state, we create the tuple of the terms that correspond to the references in the local variables and the operand stack. For example, state J in Fig. 3 is transformed to the tuple of the terms $\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1))$, $\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_2))$, and $\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1))$.

Definition 3.4 (Transforming states)**.** Let $s = (pp, l, op, h) \in \textsc{States}$, let $lv_0, \ldots, lv_n$ and $os_0, \ldots, os_m$ be the references in l and op, respectively (i.e., $h(\textsc{lv}_i) = lv_i$ and $h(\textsc{os}_i) = os_i$). We define the following mapping: $\mathrm{ts}(s) = (\mathrm{tr}(s, lv_0), \ldots, \mathrm{tr}(s, lv_n), \mathrm{tr}(s, os_0), \ldots, \mathrm{tr}(s, os_m))$.

There is a connection between the instance relation on states and the matching relation on the corresponding terms. If s' is an instance of state s, then the terms in the transformation of s match the terms in the transformation of s'. Hence, if one generates rules matching the term representation of s, then these rules also match the term representation of s'.

Lemma 3.5. *Let $s' \sqsubseteq s$. Then there exists a substitution σ such that* $\mathrm{ts}(s)\sigma = \mathrm{ts}(s')$.[5]

Now we show how to build an ITRS from a termination graph such that termination of the ITRS implies termination of the graph, i.e., that there is no infinite computation path $s_1^1, \ldots, s_1^{n_1}, s_2^1, \ldots, s_2^{n_2}, \ldots$. In other words, there should be no infinite computation sequence $t_1, t_2, \ldots$ of concrete states where $t_i \sqsubseteq s_i^1$ for all i.

For any abstract state s of the graph, we introduce a new function symbol f_s. The arity of f_s is the number of components in the tuple $\mathrm{ts}(s)$. Our goal is to generate an ITRS $\mathcal{R}$ such that $\mathsf{f}_{s_i^1}(\mathrm{ts}(t_i)) \hookrightarrow_{\mathcal{R}}^{+} \mathsf{f}_{s_{i+1}^1}(\mathrm{ts}(t_{i+1}))$ for all i. In other words, every computation path in the graph must be transformable into a rewrite sequence. Then each infinite computation path corresponds to an infinite rewrite sequence with $\mathcal{R}$.

To this end, we transform each edge in the termination graph into a rewrite rule. Let s, s' be two states connected by an edge e. If e is a split edge or an evaluation edge, then the corresponding rule should rewrite any instance of s to the corresponding instance of s'. Hence, we generate the rule $\mathsf{f}_s(\mathrm{ts}(s)) \to \mathsf{f}_{s'}(\mathrm{ts}(s'))$. For example, the edge from D to F in Fig. 3 results in the rule $\mathsf{f}_D(\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)), o_2, \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1))) \to \mathsf{f}_F(\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)), o_2)$.

[5] For all proofs, we refer to Appendix A.

If the evaluation involves checking some integer condition, we create a corresponding conditional rule. For example, the edge from H to J in Fig. 3 yields the rule

$$\begin{aligned}&\mathsf{f}_H(\,\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc},i_1)),\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc},i_2)),\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc},i_1)),i_1,i_2) \rightarrow\\ &\mathsf{f}_J(\,\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc},i_1)),\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc},i_2)),\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc},i_1))) \qquad\qquad \mid i_1 \geq i_2\end{aligned}$$

The only evaluation edges which do not result in the rule $\mathsf{f}_s(\mathrm{ts}(s)) \rightarrow \mathsf{f}_{s'}(\mathrm{ts}(s'))$ are evaluations of `putfield` instructions. If `putfield` writes to the field of an object at reference o, then this could modify all objects at references o' with $o \searrow\!\!\swarrow o'$.[6] Therefore, in the right-hand side of the rule corresponding to `putfield`, we do not transform the reference o' to the variable o', but to a fresh variable $\overline{o'}$. As an example consider state F in Fig. 9, where we write to a field of o_7, and we have the annotation $o_1 \searrow\!\!\swarrow o_7$. In the resulting rule, we therefore have the variable o_1 on the left-hand side, but a fresh variable $\overline{o_1}$ on the right-hand side. The terms corresponding to o_7 on the left- and right-hand side of the resulting rule describe the update of its field precisely (i.e., $\mathsf{jlO}(\mathsf{Tree}(\mathsf{eoc},i_1,o_{10},o_{11}))$ is replaced by o_{11}).

Now let e be an instance edge from s to s'. Here we keep the information that we already have for the specialized state s (i.e., we keep $\mathrm{ts}(s)$) and continue rewriting with the rules we already created for s'. So instead of $\mathsf{f}_s(\mathrm{ts}(s)) \rightarrow \mathsf{f}_{s'}(\mathrm{ts}(s'))$ we generate $\mathsf{f}_s(\mathrm{ts}(s)) \rightarrow \mathsf{f}_{s'}(\mathrm{ts}(s))$.

Finally, let e be a refinement edge from s to s'. So some abstract information in s is refined to more concrete information in s' (e.g., by refining $(c,?)$ to `null`). These edges represent a case analysis and hence, some instances of s are also instances of s', but others are no instances of s'. Note that by Lemma 3.5, if a state t is an instance of s', then the term representation of s' matches t's term representation. Hence, we can use pattern matching to perform the necessary case analysis. So instead of the rule $f_s(\mathrm{ts}(s)) \rightarrow f_{s'}(\mathrm{ts}(s'))$, we create a rule whose left-hand side only matches instances of s', i.e., $f_s(\mathrm{ts}(s')) \rightarrow f_{s'}(\mathrm{ts}(s'))$. Consider for example the edge from B to C in Fig. 3. Any concrete state whose evaluation corresponds to this edge must have `null` at positions LV_0 and OS_0. Thus, we create the rule $\mathsf{f}_B(\mathsf{null},o_2,\mathsf{null}) \rightarrow \mathsf{f}_C(\mathsf{null},o_2,\mathsf{null})$ which is only applicable to such states.

Recall that possibly cyclic data objects are translated to variables in Def. 3.3. Although variables are only instantiated by finite (non-cyclic) terms in term rewriting, our approach remains sound because states with possibly cyclic objects result in rules with extra variables on right-hand sides. For example, consider a simple list traversal algorithm. Here, we would have a state s where the local variable points to a reference o_1 with $o_1 = \texttt{Intlist}(\texttt{value} = i_1, \texttt{next} = o_2)$ and in the successor state s', the local variable would point to o_2. Then, after refinement to $o_2 = \texttt{IntList}(\texttt{value} = i_2, \texttt{next} = o_3)$, there would be an instantiation edge back to s. For acyclic lists, this results in the rules $\mathsf{f}_s(\mathsf{jlO}(\mathsf{IntList}(\mathsf{eoc},i_1,o_2))) \rightarrow \mathsf{f}_{s'}(o_2)$, $\mathsf{f}_{s'}(\mathsf{jlO}(\mathsf{IntList}(\mathsf{eoc},i_2,o_3))) \rightarrow \mathsf{f}_{s''}(\mathsf{jlO}(\mathsf{IntList}(\mathsf{eoc},i_2,o_3)))$ and $\mathsf{f}_{s''}(\mathsf{jlO}(\mathsf{IntList}(\mathsf{eoc},i_2,o_3))) \rightarrow \mathsf{f}_s(\mathsf{jlO}(\mathsf{IntList}(\mathsf{eoc},i_2,o_3)))$ whose termination is easy to show. But if we had the annotations $o_1!$ and $o_2!$ in s, $o_2!$ in s', and $o_2!$ and $o_3!$ in s'', then we would obtain the rules $\mathsf{f}_s(o_1) \rightarrow \mathsf{f}_{s'}(o_2)$, $\mathsf{f}_{s'}(o_2) \rightarrow \mathsf{f}_{s''}(o_2)$ and $\mathsf{f}_{s''}(o_2) \rightarrow \mathsf{f}_s(o_2)$. So in the first rule, o_2 would be an extra variable representing an arbitrary list, and the resulting rules would not be terminating.

Definition 3.6 (Rewrite rules from termination graphs)**.** Let there be an edge e from the state $s = (pp, l, op, h)$ to the state s' in a termination graph. Then we generate $rule(e)$:

- if e is an instance edge, then $rule(e) = \mathsf{f}_s(\mathrm{ts}(s)) \rightarrow \mathsf{f}_{s'}(\mathrm{ts}(s))$
- if e is a refinement edge, then $rule(e) = \mathsf{f}_s(\mathrm{ts}(s')) \rightarrow \mathsf{f}_{s'}(\mathrm{ts}(s'))$
- if e is an evaluation or split edge, we perform the following case analysis:

[6]Note that this is only possible if o' is UNKNOWN.

– if e is labeled by a statement of the form $o_1 = o_2 \oplus o_3$ where $\oplus \in \mathcal{A}rith\mathcal{O}p$, then $rule(e) = \mathsf{f}_s(\text{ts}(s)) \to \mathsf{f}_{s'}(\text{ts}(s'))\sigma$, where σ substitutes o_1 by $\text{tr}(s, o_2) \oplus \text{tr}(s, o_3)$
– if e is labeled by a condition $o_1 \oplus o_2$ where $\oplus \in \mathcal{R}el\mathcal{O}p$, then $rule(e) = \mathsf{f}_s(\text{ts}(s)) \to \mathsf{f}_{s'}(\text{ts}(s')) \mid \text{tr}(s, o_1) \oplus \text{tr}(s, o_2)$
– if pp is the instruction `putfield` writing to a field of reference o, then $rule(e) = \mathsf{f}_s(\text{ts}(s)) \to \mathsf{f}_{s'}(\text{ts}_o(s'))$, where $\text{ts}_o(s')$ is defined like $\text{ts}(s')$, but each reference o' with $o \searrow\!\swarrow o'$ is transformed into a new fresh variable.
– for all other instructions, $rule(e) = \mathsf{f}_s(\text{ts}(s)) \to \mathsf{f}_{s'}(\text{ts}(s'))$

Our main theorem states that every computation path of the termination graph can be simulated by a rewrite sequence using the corresponding ITRS. Of course, the converse does not hold, i.e., our approach cannot be used to prove non-termination of JBC programs.

Theorem 3.7 (Proving termination of JBC by ITRSs)**.** *If the ITRS corresponding to a termination graph is terminating, then the termination graph is terminating as well. Hence, then the original JBC program is also terminating for all concrete states t where $t \sqsubseteq s$ for some abstract state s in the termination graph.*

The resulting ITRSs are usually large, since they contain one rule for each edge of the termination graph. But since our ITRSs have a special form where the roots of all left- and right-hand sides are defined, where defined symbols do not occur below the roots, and where we only consider rewriting with normal substitutions, one can simplify the ITRSs substantially by *merging* their rules: Let $\mathcal{R}_1$ (resp. $\mathcal{R}_2$) be those rules in $\mathcal{R}$ where the root of the right- (resp. left-)hand side is f. Then one can replace the rules $\mathcal{R}_1 \cup \mathcal{R}_2$ by the rules "$\ell\sigma \to r'\sigma \mid b\sigma \;\&\&\; b'\sigma$" for all $\ell \to r \mid b \in \mathcal{R}_1$ and all $\ell' \to r' \mid b' \in \mathcal{R}_2$, where $\sigma = \text{mgu}(r, \ell')$. Of course, we also have to add rules for the Boolean conjunction "&&". Clearly, this process does not modify the termination behavior of $\mathcal{R}$. Moreover, it suffices to create rules only for those edges that occur in cycles of the termination graph.

With this simplification, we automatically obtain the following 1-rule ITRS for the `count` program. It increases the value i_1 of f_G's first and third argument (corresponding to the `value`-field of `orig` and `copy`) as long as $i_1 < i_2$ (where i_2 is the `value`-field of `limit`).

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

For the `flatten` program we automatically obtain the following ITRS. To ease readability, we replaced every subterm "$\mathsf{jlO}(t)$" by just t, and we replaced "$\mathsf{TreeList}(\mathsf{eoc}, v, n)$" by "$\mathsf{TL}(v, n)$", "$\mathsf{Tree}(\mathsf{eoc}, v, l, r)$" by "$\mathsf{T}(v, l, r)$", and "$\mathsf{IntList}(\mathsf{eoc}, v, n)$" by "$\mathsf{IL}(v, n)$".

$$\mathsf{f}_S(\mathsf{TL}(\mathsf{null}, o_9), \mathsf{TL}(\mathsf{null}, o_9), o_6) \to \mathsf{f}_S(\mathsf{TL}(\mathsf{null}, o_9), o_9, o_6) \tag{3.1}$$

$$\mathsf{f}_S(\mathsf{TL}(\mathsf{T}(i_1, o_{10}, o_{11}), o_9), \mathsf{TL}(\mathsf{T}(i_1, o_{10}, o_{11}), o_9), o_6) \to \mathsf{f}_S(\mathsf{TL}(o_{11}, o_9), \mathsf{TL}(o_{10}, \mathsf{TL}(o_{11}, o_9)), \mathsf{IL}(i_1, o_6)) \tag{3.2}$$

$$\mathsf{f}_S(o_1, \mathsf{TL}(\mathsf{null}, o_9), o_6) \to \mathsf{f}_S(o_1, o_9, o_6) \tag{3.3}$$

$$\mathsf{f}_S(o_1, \mathsf{TL}(\mathsf{T}(i_1, o_{10}, o_{11}), o_9), o_6) \to \mathsf{f}_S(\overline{o_1}, \mathsf{TL}(o_{10}, \mathsf{TL}(o_{11}, o_9)), \mathsf{IL}(i_1, o_6)) \tag{3.4}$$

Rules (3.1) and (3.3) correspond to the cycles from S over B' and over E. Their difference is whether `l` and `c` point to the same object in S (i.e., whether the first two arguments of f_S in the left-hand side are identical). But both handle the case where the first tree in the list `c` (i.e., in f_S's second argument) is null. Then this null-tree is simply removed from the list and the result `r` (i.e., the third argument of f_s) does not change. The rules (3.2) and (3.4) correspond to the cycles from S over C' and over G. Here, the list `c` has the form $\mathsf{TL}(\mathsf{T}(i_1, o_{10}, o_{11}), o_9)$. Hence, the value i_1 of the first tree in the list is stored in the result list (which is modified from o_6 to $\mathsf{IL}(i_1, o_6)$) and the list `c` is modified to $\mathsf{TL}(o_{10}, \mathsf{TL}(o_{11}, o_9))$. So the length of the list increases, but the number of nodes in the list decreases.

These examples illustrate that the ITRSs resulting from our automatic transformation of JBC are often very readable and constitute a natural representation of the original algorithm as a rewrite system. Not surprisingly, existing TRS techniques can easily prove termination of the resulting rules. For example, termination of the above ITRS for `flatten` is easily proved using a straightforward polynomial interpretation and dependency pairs. In contrast, abstraction-based tools like Julia and COSTA fail on examples like `flatten`. In fact, Julia and COSTA also fail on the `count` example from Sect. 2.2.

4. Experiments and Conclusion

We introduced an approach to prove termination of JBC programs automatically by first transforming them to termination graphs. Then an integer TRS is generated from the termination graph and existing TRS tools can be used to show its termination.

We implemented our approach in the termination prover AProVE [9] and evaluated it on the 106 non-recursive JBC examples from the *termination problem data base (TPDB)* used in the *International Termination Competition*.[7] In our experiments, we removed one controversial example ("`overflow`") from the TPDB whose termination depends on the treatment of integer overflows and we added the two examples `count` and `flatten` from this paper. Of these 106 examples, 10 are known to be non-terminating. See `http://aprove.informatik.rwth-aachen.de/eval/JBC` for the origins of the individual examples. As in the competition, we ran AProVE and the tools Julia [18] and COSTA [1] with a time limit of 60 seconds on each example. "Success" gives the number of examples where termination was proved, "Failure" means that the proof failed in less than 60 seconds, "Timeout" gives the number of examples where the tool took longer than 60 seconds, and "Runtime" is the average time (in s) needed per example. Note that for those examples from this collection where AProVE resulted in a timeout, the tool would also fail when using a longer timeout.

	all 106 non-recursive examples			
	Success	Failure	Timeout	Runtime
AProVE	89	5	12	14.3
Julia	74	32	0	2.6
COSTA	60	46	0	3.4

Our experiments show that for the problems in the current example collection, our rewriting-based approach in AProVE currently yields the most precise results. The main reason is that we do not use a fixed abstraction from data objects to integers, but represent objects as terms. On the other hand, this also explains the larger runtimes of AProVE compared to Julia and COSTA. Still, our approach is efficient enough to solve most examples in reasonable time. Our method benefits substantially from the representation of objects as terms, since afterwards arbitrary TRS termination techniques can be used to prove termination of the algorithms. Of course, while the examples in the TPDB are challenging, they are still quite small. Future work will be concerned with the application and adaption of our approach in order to use it also for large examples and Java libraries.

In the current paper, we restricted ourselves to JBC programs without recursion, whereas the approaches of Julia and COSTA also work on recursive programs. Of course,

[7]See `http://www.termination-portal.org/wiki/Termination_Competition`.

an extension of our method to recursive programs is another main point for future work. Our experiments also confirm the results at the *International Termination Competition* in December 2009, where the first competition on termination of JBC programs took place. Here, the three tools above were run on a random selection of the examples from the TPDB with similar results. To experiment with our implementation via a web interface and for details about the above experiments, we refer to `http://aprove.informatik.rwth-aachen.de/eval/JBC`.

Acknowledgement

We are indebted to the Julia- and the COSTA-team for their help with the experiments. We thank the anonymous reviewers for their valuable comments.

References

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination analysis of Java Bytecode. In *Proc. FMOODS '08*, LNCS 5051, pages 2–18, 2008.
[2] J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proc. CAV '06*, LNCS 4144, pages 386–400, 2006.
[3] A. R. Bradley, Z. Manna, and H. B. Sipma. Termination of polynomial programs. In *Proc. VMCAI '05*, LNCS 3385, pages 113–129, 2005.
[4] M. Colón and H. Sipma. Practical methods for proving program termination. In *Proc. CAV '02*, LNCS 2404, pages 442–454, 2002.
[5] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. PLDI '06*, pages 415–426. ACM Press, 2006.
[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL '77*, pages 238–252. ACM Press, 1977.
[7] S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *Proc. CADE '09*, LNAI 5663, pages 277–293, 2009.
[8] C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *Proc. RTA '09*, LNCS 5595, pages 32–47, 2009.
[9] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR '06*, LNAI 4130, pages 281–286, 2006.
[10] J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. In *RTA '06*, LNCS 4098, pp. 297–312, 2006.
[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2005.
[12] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *Proc. POPL '09*, pages 127–139. ACM Press, 2009.
[13] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.
[14] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Prentice Hall, 1999.
[15] M. T. Nguyen, D. De Schreye, J. Giesl, and P. Schneider-Kamp. Polytool: Polynomial interpretations as a basis for termination analysis of logic programs. *Theory and Practice of Logic Programming*, 2010. To appear. Available from `http://arxiv.org/pdf/0912.4360`.
[16] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Transactions on Computational Logic*, 11(1), Article 2, 2009.
[17] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In *Proc. ILPS '95*, pages 465–479. MIT Press, 1995.
[18] F. Spoto, F. Mesnard, and É. Payet. A termination analyser for Java Bytecode based on path-length. *ACM Transactions on Programming Languages and Systems*, 32(3), Article 8, 2010.

Appendix A. Proofs

To prove Lemma 3.5, we first need several auxiliary lemmas. The first auxiliary lemma shows that if a state s' is an instance of a state s, then all positions of s also occur in s'.

Lemma 4.1 (Positions in instances)**.** *Let* $s, s' \in \text{States}$ *where* $s' \sqsubseteq s$. *Then* $\text{SPos}(s) \subseteq \text{SPos}(s')$.

Proof. Let $s = (pp, l, op, h)$ and $s' = (pp', l', op', h')$. Moreover, let $\pi \in \text{SPos}(s)$. We prove $\pi \in \text{SPos}(s')$ by induction on π.

If $\pi = \text{LV}_n$ or $\pi = \text{OS}_n$, then the claim follows from the fact that $pp = pp'$ and that in verified JBC, states that correspond to the same program position have the same local variables and the same number of entries on the operand stack.

If $\pi = \pi' v$ for some $v \in \text{FieldIdentifiers}$, then $h(s|_{\pi'}) = (c, f) \in \text{Instances}$ where $f(v)$ is defined (i.e., v is a field declared in class c). As $\pi' \in \text{SPos}(s)$, by the induction hypothesis, we know that $\pi' \in \text{SPos}(s')$ as well. Since $s' \sqsubseteq s$, $h(s|_{\pi'}) = (c, f) \in \text{Instances}$ implies $h(s'|_{\pi'}) = (c, f') \in \text{Instances}$. As v is declared in class c, $f'(v)$ must be defined and thus, $\pi' v = \pi \in \text{SPos}(s')$. ∎

Not every position π of a state s is "reached" by our transformation which transforms states s into tuples $\text{ts}(s)$ of terms. For example, if $s|_\pi$ is a reference that is part of a cyclic structure, then this position is never reached during the transformation, since cyclic objects are directly converted to variables in Def. 3.3. The set of reached positions is defined as follows.

Definition 4.2 (Reached positions)**.** Let $s \in \text{States}$. Then we define the set of *reached positions* $\overline{\text{SPos}}(s)$ as the smallest set satisfying the following conditions.

- If $\text{LV}_i \in \text{SPos}(s)$ holds for some number i, then $\text{LV}_i \in \overline{\text{SPos}}(s)$ holds, too.
- If $\text{OS}_i \in \text{SPos}(s)$ holds for some number i, then $\text{OS}_i \in \overline{\text{SPos}}(s)$ holds, too.
- If we have $\pi = \pi' v \in \text{SPos}(s)$ for some $v \in \text{FieldIdentifiers}$ and if $s|_{\pi'}$ is not special, then $\pi \in \overline{\text{SPos}}(s)$ holds, too.

With this definition, for each position referencing a cyclic structure, only the topmost special reference is included in $\overline{\text{SPos}}(s)$.

The well-known concept of positions of terms can easily be extended to tuples of terms. We say that "$i\,p$" is a position in the tuple $(t_1, \dots, t_n)$ if $1 \leq i \leq n$ and p is a position of t_i. Then $(t_1, \dots, t_n)|_{i\,p} = t_i|_p$. Let $Pos((t_1, \dots, t_n))$ be the set of all positions of the tuple $(t_1, \dots, t_n)$.

Now for any state s and any position $\pi \in \overline{\text{SPos}}(s)$, we can define the *corresponding position* $\text{cp}_s(\pi) \in Pos(\text{ts}(s))$. The term $\text{ts}(s)|_{\text{cp}_s(\pi)}$ is the one which results from the reference at the position $\pi \in \text{SPos}(s)$.

Definition 4.3 (Corresponding position)**.** Let $s = (pp, l, op, h) \in \text{States}$, and let $n_1, n_2 \in \mathbb{N}$ be the maximum numbers such that $l(i)$ is defined for all $0 \leq i < n_1$ and $op(i)$ is defined for all $0 \leq i < n_2$. For any position π of $\overline{\text{SPos}}(s)$, we define the position $\text{cp}_s(\pi)$ of $\text{ts}(s)$ as follows:

- if $\pi = \text{LV}_i$ for some $0 \leq i < n_1$, then we define $\text{cp}_s(\pi) = i + 1$.
- if $\pi = \text{OS}_i$ for some $0 \leq i < n_2$, then we define $\text{cp}_s(\pi) = n_1 + i + 1$.
- if $\pi = \pi' v$ for some $v \in \text{FieldIdentifiers}$, then let c be the class described by v. Furthermore let ($c_1 =$ `java.lang.Object`$, c_2, \dots, c_n = c$) be the sequence of classes

such that c_i is the direct superclass of c_{i+1}. Finally let m be the number of the field described by v in c according to the order used in Def. 3.3. Then we define $\mathrm{cp}_s(\pi) = \mathrm{cp}_s(\pi')\, 1^{n-1}\, (m+1)$.

As an example, consider state s in node J of Fig. 3. Here we have

$$\mathrm{ts}(s) = (\, \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)),\ \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_2)),\ \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1))\,)$$

We have $s|_{\mathrm{LV}_0} = o_1$, and the corresponding position of LV_0 is $\mathrm{cp}_s(\mathrm{LV}_0) = 1$. The term at that position is $\mathrm{ts}(s)|_{\mathrm{cp}_s(\mathrm{LV}_0)} = \mathrm{ts}(s)|_1 = \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1))$, which is the transformation of o_1 (i.e., $\mathrm{tr}(s, s|_{\mathrm{LV}_0}) = \mathrm{tr}(s, o_1) = \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)))$.

As another example consider position $\pi = \mathrm{LV}_0\, \mathtt{val}$ from $\mathrm{SPos}(s)$. Here we have $s|_\pi = i_1$. The sequence of classes as used in the previous definition is $(\mathtt{java.lang.Object}, \mathtt{Int})$ and `val` is the first field of class `Int`. Using the same variables as in the previous definition we hence have $m = 1$ and $n = 2$. So we obtain $\mathrm{cp}_s(\mathrm{LV}_0\, \mathtt{val}) = \mathrm{cp}_s(\mathrm{LV}_0) 1^{n-1}\, (m+1) = 1\,1\,2$ and thus $\mathrm{ts}(s)|_{\mathrm{cp}_s(\mathrm{LV}_0\, \mathtt{val})} = \mathrm{ts}(s)|_{1\,1\,2} = i_1$ holds. The variable i_1, in turn, is the transformation of the reference $s|_{\mathrm{LV}_0\, \mathtt{val}} = i_1$ (i.e., $\mathrm{tr}(s, s|_{\mathrm{LV}_0\, \mathtt{val}}) = \mathrm{tr}(s, i_1) = i_1$). The following lemma shows that this equality of $\mathrm{ts}(s)|_{\mathrm{cp}_s(\pi)}$ and $\mathrm{tr}(s, s|_\pi)$ holds for any reached position π in s.

Lemma 4.4 ("Soundness" of corresponding position)**.** *Let* $s \in \mathrm{States}$ *and let* $\pi \in \overline{\mathrm{SPos}}(s)$. *Then we have* $\mathrm{ts}(s)|_{\mathrm{cp}_s(\pi)} = \mathrm{tr}(s, s|_\pi)$.

Proof. We prove this lemma by induction over π. In the base case π is either LV_i or OS_i for some number i. In this case the lemma obviously holds.

Let now $\pi = \pi'\, v$ for some $v \in \mathrm{FieldIdentifiers}$. Let c be the class described by v and let m be the number of the field v according to the order used in Def. 3.3. Let furthermore $(c_1 = \mathtt{java.lang.Object}, c_2, \ldots, c_n = c)$ be the sequence of classes such that c_i is the direct superclass of c_{i+1}. Then we have $\mathrm{cp}_s(\pi) = \mathrm{cp}_s(\pi')\, 1^{n-1}\, (m+1)$.

Furthermore we have that $\mathrm{tr}(s, s|_{\pi'}) = c_1(c_2(\ldots(c_n(t, t_1, \ldots, t_l), \ldots)))$ holds for some terms $t, t_1, \ldots, t_l$ where $l \geq m$ because of the definition of tr and because we know that $s|_{\pi'}$ is not special. Hence we have $\mathrm{tr}(s, s|_{\pi'})|_{1^{n-1}(m+1)} = t_m = \mathrm{tr}(s, s|_\pi)$ because of the definition of tr. The induction hypothesis implies $\mathrm{tr}(s, s|_{\pi'}) = \mathrm{ts}(s)|_{\mathrm{cp}_s(\pi')}$ and hence $\mathrm{tr}(s, s|_\pi) = \mathrm{tr}(s, s|_{\pi'})|_{1^{n-1}(m+1)} = (\mathrm{ts}(s)|_{\mathrm{cp}_s(\pi')})|_{1^{n-1}(m+1)} = \mathrm{ts}(s)|_{\mathrm{cp}_s(\pi)}$ holds by the definition of cp_s. ∎

We are not yet able to relate every position in the transformed tuple $\mathrm{ts}(s)$ to a position in the state s, i.e., cp_s is not surjective. For instance, for the state s from node J of Fig. 3, there is no $\pi \in \mathrm{SPos}(s)$ with $\mathrm{cp}_s(\pi) = 1\,1$ or $\mathrm{cp}_s(\pi) = 1\,1\,1$. So we cannot address the subterms $\mathsf{Int}(\mathsf{eoc}, i_1)$ and eoc at the positions $1\,1$ and $1\,1\,1$ of $\mathrm{ts}(s)$. Hence we define the set of corresponding suffixes, which we can append to corresponding positions to address all subterms.

Definition 4.5 (Corresponding suffixes)**.** Let $s = (pp, l, op, h) \in \mathrm{States}$ and $\pi \in \overline{\mathrm{SPos}}(s)$. Then we define $\overline{\mathrm{cp}}_s(\pi)$ as follows.

- If $s|_\pi = \mathtt{null}$, if $s|_\pi$ is special, or if $h(s|_\pi) \notin \mathrm{Instances}$, then we define $\overline{\mathrm{cp}}_s(\pi) = \{\varepsilon\}$.
- Otherwise let $h(s|_\pi) = (c, f)$, and let $(c_1 = \mathtt{java.lang.Object}, c_2, \ldots, c_n = c)$ be the sequence of classes such that c_i is the direct superclass of c_{i+1}. Then we define $\overline{\mathrm{cp}}_s(\pi) := \{\varepsilon, 1, \ldots, 1^n\}$.

For the state s in node J of Fig. 3, to address the subterms $\mathsf{Int}(\mathsf{eoc}, i_1)$ and eoc we need the suffixes 1 and 11, and indeed we have $\overline{\mathrm{cp}}_s(\mathrm{LV}_0) = \{\varepsilon, 1, 11\}$. Hence, using $\mathrm{cp}_s(\mathrm{LV}_0)$ as prefix and the elements of $\overline{\mathrm{cp}}_s(\mathrm{LV}_0)$ as suffixes, we can address any subterm for which the reference o_1 at the position LV_0 in s is directly responsible. As another example, we had $\text{cp}_s(\text{LV}_0\,\texttt{val}) = 1\,1\,2$, and $\text{ts}(s)|_{1\,1\,2} = i_1$. This is the term resulting from the reference i_1 at position $\text{LV}_0\,\texttt{val}$ in s. Note that $1\,1\,2$ cannot be constructed from $\text{cp}_s(\text{LV}_0)$ and $\overline{\text{cp}}_s(\text{LV}_0)$, and this is intended. The reference i_1 at position $\text{LV}_0\,\texttt{val}$ in s is directly responsible for this subterm, while the reference o_1 at position LV_0 in s is only indirectly responsible.

Together, these two functions can address every position in the tuple of terms resulting from the transformation of a state.

Lemma 4.6 (*cp_s and $\overline{cp}_s$ describe all positions of* $\text{ts}(s)$)**.** *Let $s \in$ States. Then we have $Pos(\text{ts}(s)) = \{\text{cp}_s(\pi)q \mid \pi \in \overline{\text{SPos}}(s), q \in \overline{\text{cp}}_s(\pi)\}$.*

Proof. This follows trivially from the definitions of ts, cp, and $\overline{\text{cp}}$. ∎

The following lemma shows the relationship of corresponding positions and suffixes for states that are instances of each other.

Lemma 4.7 (Instance and corresponding positions)**.** *Let $s, s' \in$ States where $s' \sqsubseteq s$, let $\pi \in \overline{\text{SPos}}(s)$. Then we have $\text{cp}_s(\pi) = \text{cp}_{s'}(\pi)$ and $\overline{\text{cp}}_s(\pi) \subseteq \overline{\text{cp}}_{s'}(\pi)$.*

Proof. First note that $\pi \in \text{SPos}(s)$ implies $\pi \in \text{SPos}(s')$ by Lemma 4.1. It is easy to prove that $\pi \in \overline{\text{SPos}}(s)$ also implies $\pi \in \overline{\text{SPos}}(s')$. Now $\text{cp}_s(\pi) = \text{cp}_{s'}(\pi)$ follows from the fact that the value of $\text{cp}_s(\pi)$ is independent from s and only depends on π. The claim $\overline{\text{cp}}_s(\pi) \subseteq \overline{\text{cp}}_{s'}(\pi)$ follows trivially from $s' \sqsubseteq s$. ∎

Now we show that if $s' \sqsubseteq s$ and if the same reference o occurs at two different positions p and p' in the tuple of terms $\text{ts}(s)$, then the terms at the positions p and p' in the tuple of terms $\text{ts}(s')$ are also identical.

Lemma 4.8 (Equality of terms)**.** *Let $s, s' \in States$ with $s' \sqsubseteq s$ and let $\text{ts}(s)|_p = \text{ts}(s)|_{p'} \in$ References (i.e., $\text{ts}(s)|_p$ and $\text{ts}(s)|_{p'}$ are the same variable). Then we also have $\text{ts}(s')|_p = \text{ts}(s')|_{p'}$.*

Proof. Because of $\text{ts}(s)|_p = o \in$ References, we have $\text{ts}(s)|_p = \text{tr}(s, o)$. Let $\pi \in \overline{\text{SPos}}(s)$ be such that $\text{cp}_s(\pi)q = p$ for a $q \in \overline{\text{cp}}_s(\pi)$. Such π and q must exist by Lemma 4.6. Then we have $q = \varepsilon$ because of the definitions of $\overline{\text{cp}}$ and tr. Analogously, let $\pi' \in \overline{\text{SPos}}(s)$ be such that $\text{cp}_s(\pi') = p'$.

Now we have $o = \text{ts}(s)|_p = \text{ts}(s)|_{\text{cp}_s(\pi)} = \text{tr}(s, s|_\pi)$ and hence $s|_\pi = o$. Analogously we have $s|_{\pi'} = o$. Hence π and π' address the same reference in s, i.e., we have $s|_\pi = s|_{\pi'}$. Because s' is an instance of s we also have $s'|_\pi = s'|_{\pi'}$. By Lemma 4.7, we have $p = \text{cp}_s(\pi) = \text{cp}_{s'}(\pi)$ and $p' = \text{cp}_s(\pi') = \text{cp}_{s'}(\pi')$. As π and π' also address the same reference in s', by Lemma 4.4 we have $\text{ts}(s')|_p = \text{ts}(s')|_{\text{cp}_{s'}(\pi)} = \text{tr}(s', s'|_\pi) = \text{tr}(s', s'|_{\pi'}) = \text{ts}(s')|_{\text{cp}_{s'}(\pi')} = \text{ts}(s')|_{p'}$. ∎

Now we can finally prove Lemma 3.5.

Lemma 3.5 (Instances and matching)**.** *Let $s' \sqsubseteq s$. Then there exists a substitution σ such that $\text{ts}(s)\sigma = \text{ts}(s')$.*

Proof. Let $s = (pp, l, op, h)$ and $s' = (pp', l', op', h')$. Moreover, let $\pi \in \overline{\text{SPos}}(s)$ with $o = s|_\pi$. Since $s' \sqsubseteq s$, by Lemma 4.1 (resp. by its extension to reached positions) we also have $\pi \in \overline{\text{SPos}}(s')$. Hence, there exists a reference o' with $o' = s'|_\pi$. We will now show that the term representation of o in s matches the term representation of o' in s'. In other words, there is a substitution σ_o such that $\text{tr}(s, o)\sigma_o = \text{tr}(s', o')$.

In Lemma 4.8 we showed that if the same variable o occurs at two different positions p and p' in the tuple of terms $\mathrm{ts}(s)$, then the terms at the positions p and p' in the tuple of terms $\mathrm{ts}(s')$ are also identical. Hence, the different substitutions σ_o for the different references o above can be combined into one single substitution σ satisfying $\mathrm{ts}(s)\sigma = \mathrm{ts}(s')$.

To prove the existence of a substitution σ_o with $\mathrm{tr}(s,o)\sigma_o = \mathrm{tr}(s',o')$, we perform induction on the term structure of $\mathrm{tr}(s,o)$.

If $\mathrm{tr}(s,o) = i \in \mathbb{Z}$, then $h(o) = h(s|_\pi) = [i,i]$ for an $i \in \mathbb{Z}$. As $s' \sqsubseteq s$, we also have $h'(s'|_\pi) = [i,i]$. Hence, $\mathrm{tr}(s',o') = i = \mathrm{tr}(s,o)$ and thus, σ_o is the identity.

If $\mathrm{tr}(s,o) = \mathsf{null}$, then $o = s|_\pi = \mathtt{null}$. Hence $s' \sqsubseteq s$ implies $o' = s'|_\pi = \mathtt{null}$ as well. So $\mathrm{tr}(s',o') = \mathsf{null} = \mathrm{tr}(s,o)$ and thus, σ_o is again the identity.

If $\mathrm{tr}(s,o)$ is the variable o, then we simply let σ_o be the substitution that instantiates the variable o by the term $\mathrm{tr}(s',o')$.

Finally, we regard the case where

$$\mathrm{tr}(s,o) = \mathrm{ti}(s,o) = c_1(c_2(\dots(c_n(\mathsf{eoc}, v_{n,1},\dots,v_{n,m_n}),\dots),v_{2,1},\dots,v_{2,m_2})).$$

(Note that the class $c_1 = \mathtt{java.lang.Object}$ has no fields.) Hence $h(o) = h(s|_\pi) \in$ Instances, and thus, $s' \sqsubseteq s$ implies that $h'(o') = h'(s'|_\pi) \in$ Instances as well and the types of $h(o)$ and $h'(o')$ are equal. Thus,

$$\mathrm{tr}(s',o') = \mathrm{ti}(s',o') = c_1(c_2(\dots(c_n(\mathsf{eoc}, v'_{n,1},\dots,v'_{n,m_n}),\dots),v'_{2,1},\dots,v'_{2,m_2})).$$

Let $p_{i,j}$ be the position of $v_{i,j}$ in $\mathrm{ts}(s)$ and of $v'_{i,j}$ in $\mathrm{ts}(s')$, and let $\pi_{i,j} \in \overline{\mathrm{SPos}}(s)$ be such that $p_{i,j} = \mathrm{cp}_s(\pi_{i,j})$. So for $\pi_{i,j}$ we have $s|_{\pi_{i,j}} = o_{i,j}$ such that $v_{i,j} = \mathrm{tr}(s, o_{i,j})$ by Lemma 4.4.

By Lemma 4.7, $\mathrm{cp}_s(\pi_{i,j}) = \mathrm{cp}_{s'}(\pi_{i,j}) = p_{i,j}$, and hence $v'_{i,j} = \mathrm{tr}(s', s'|_{\pi_{i,j}})$ by Lemma 4.4. Hence, by the induction hypothesis there is a substitution $\sigma_{o_{i,j}}$ such that $v_{i,j}\sigma_{o_{i,j}} = v'_{i,j}$.

As discussed above, by Lemma 4.8 the different substitutions $\sigma_{o_{i,j}}$ can be combined into one single substitution σ_o satisfying $v_{i,j}\sigma_o = v'_{i,j}$ for all i,j. This implies $\mathrm{tr}(s,o)\sigma_o = \mathrm{tr}(s',o')$. ∎

Now our goal is to prove Thm. 3.7. Again, this requires some auxiliary lemmas. The first lemma states that concrete evaluations can be performed using the rule generated for the corresponding evaluation edge.

Lemma 4.9 (Correctness of *rule*(e) for evaluation edges)**.** *Let s, s' be two states in a termination graph connected by an evaluation edge e. Let t be a concrete state with $t \sqsubseteq s$ and let t' result from t by one **JBC** evaluation step (thus, $t' \sqsubseteq s'$, cf. (2.1)). Then we have* $\mathsf{f}_s(\mathrm{ts}(t)) \hookrightarrow^+_{\{rule(e)\}} \mathsf{f}_{s'}(\mathrm{ts}(t'))$.

Proof. Let $rule(e) = \ell \to r \mid b$ where $\ell = \mathsf{f}_s(\mathrm{ts}(s))$. By Lemma 3.5, there exists a substitution σ with $\ell\sigma = \mathsf{f}_s(\mathrm{ts}(s))\sigma = \mathsf{f}_s(\mathrm{ts}(t))$. We will now show that σ can be extended to a substitution $\overline{\sigma}$ such that $\ell\overline{\sigma} = \ell\sigma$, $b\overline{\sigma} \to^*_{\mathcal{PD}} \mathsf{true}$, and $r\overline{\sigma} \to^*_{\mathcal{PD}} \mathsf{f}_{s'}(\mathrm{ts}(t'))$. We perform a case analysis as in Def. 3.6.

- First, let e be labeled by a statement of the form $o_1 = o_2 \oplus o_3$ where $\oplus \in \mathcal{A}rith\mathcal{O}p$. The concrete state t' results from t by removing the two top elements from its operand stack, applying the operation $\oplus$ to them, and pushing the result on top of the operand stack again. Moreover, one moves to the next instruction of the program. A similar relationship holds for $\mathrm{ts}(t)$ and $\mathrm{ts}(t')$: $\mathrm{ts}(t')$ is obtained from $\mathrm{ts}(t)$ by removing the last two elements from the tuple and adding the result of the $\oplus$ operation as new last element of the tuple. This is clearly also done by first applying the rule $\ell \to r$ to $\mathsf{f}_s(\mathrm{ts}(t))$ and applying one rule from $\mathcal{PD}$ afterwards to evaluate the operation $\oplus$. So here $\overline{\sigma} = \sigma$ and $b = \mathsf{true}$.

- Now let e be labeled by a condition $o_1 \oplus o_2$ where $\oplus \in \mathcal{R}el\mathcal{O}p$. Again, we use $\overline{\sigma} = \sigma$. It remains to show that the constraint of the rule $rule(e)$ is fulfilled. This follows from the fact that otherwise, the evaluation of t to t' would not have taken place.
- Now we consider the case where the instruction in the state s is a `putfield` operation which writes to a field of reference o. Since t' is an instance of s', by Lemma 3.5 there is a substitution σ' with $\mathsf{f}_{s'}(\mathrm{ts}(s'))\sigma' = \mathsf{f}_{s'}(\mathrm{ts}(t'))$. Note that r differs from $\mathsf{f}_{s'}(\mathrm{ts}(s'))$, by containing fresh variables $\overline{o'}$ instead of o' whenever $o \searrow\!\swarrow o'$. We extend σ' to these fresh variables by defining $\sigma'(\overline{o'}) = \sigma'(o')$. Now we can define $\overline{\sigma}$ as follows:
$$\overline{\sigma}(x) = \begin{cases} \sigma(x) & \text{if } x \text{ is a variable of } \ell \\ \sigma'(x) & \text{otherwise} \end{cases}$$
With this substitution, we obviously have $\ell\overline{\sigma} = \ell\sigma = t$. It remains to show $r\overline{\sigma} = \mathsf{f}_{s'}(\mathrm{ts}(t'))$ or, equivalently, $\sigma(x) = \sigma'(x)$ for all variables x of r. This is clear for those variables which are not from ℓ. Hence, it remains to show that for all x from both ℓ and r, we have $\sigma(x) = \sigma'(x)$. As x occurs in ℓ, the reference x is reachable in the state s and $x \neq \mathtt{null}$. Note that since x is still in r, we cannot have $o \searrow\!\swarrow x$ in s. Hence, the heaps of s and s' do not differ at the address x, and analogously, the heaps of t and t' do not differ either. Thus, $\sigma(x) = \sigma'(x)$.
- Finally, we consider all other evaluation edges. Here, we again choose $\overline{\sigma} = \sigma$ and since the evaluation from t to t' is exactly modeled in s and s', $\mathrm{ts}(s)\sigma = \mathrm{ts}(t)$ implies $\mathrm{ts}(s')\sigma = \mathrm{ts}(t')$.

∎

The next lemma shows that every concrete evaluation step can be simulated by rewriting with the ITRS corresponding to the termination graph.

Lemma 4.10 (Simulation of concrete evaluations by ITRS)**.** *Let s be a state in the termination graph, let t be a concrete state with $t \sqsubseteq s$, and let t' result from t by one JBC evaluation step. Then for the corresponding ITRS $\mathcal{R}$, we have $\mathsf{f}_s(t) \hookrightarrow_{\mathcal{R}}^{+} \mathsf{f}_{s'}(t')$ for some s' in the termination graph.*

Proof. According to (2.1) there is a path $(s = s_1, s_2, \ldots, s_n = s')$ in the termination graph such that $t' \sqsubseteq s'$ and such that for all $1 \leq i \leq n-2$, the edge between s_i and s_{i+1} is a refinement, split, or instance edge and the edge between s_{n-1} and s_n is an evaluation edge. Here, we have $t \sqsubseteq s_i$ for all $1 \leq i \leq n-1$.

By Lemma 4.9, $\mathsf{f}_{s_{n-1}}(\mathrm{ts}(t)) \hookrightarrow_{\mathcal{R}}^{+} \mathsf{f}_{s_n}(\mathrm{ts}(t'))$. We now show that $\mathsf{f}_{s_i}(\mathrm{ts}(t)) \hookrightarrow_{\mathcal{R}} \mathsf{f}_{s_{i+1}}(\mathrm{ts}(t))$ for all $1 \leq i \leq n-2$.

If the edge from s_i to s_{i+1} is an instantiation edge, then the corresponding rule is $\mathsf{f}_{s_i}(\mathrm{ts}(s_i)) \to \mathsf{f}_{s_{i+1}}(\mathrm{ts}(s_i))$. Since $t \sqsubseteq s_i$, by Lemma 3.5 there exists a substitution σ with $\mathrm{ts}(s_i)\sigma = t$. Hence, by instantiating the rule with σ, we indeed obtain $\mathsf{f}_{s_i}(\mathrm{ts}(t)) \hookrightarrow_{\mathcal{R}} \mathsf{f}_{s_{i+1}}(\mathrm{ts}(t))$.

If the edge from s_i to s_{i+1} is a refinement edge, then the corresponding rule is $\mathsf{f}_{s_i}(\mathrm{ts}(s_{i+1})) \to \mathsf{f}_{s_{i+1}}(\mathrm{ts}(s_{i+1}))$. Since $t \sqsubseteq s_{i+1}$, by Lemma 3.5 there exists a substitution σ with $\mathrm{ts}(s_{i+1})\sigma = t$. Hence, by instantiating the rule with σ, we again have $\mathsf{f}_{s_i}(\mathrm{ts}(t)) \hookrightarrow_{\mathcal{R}} \mathsf{f}_{s_{i+1}}(\mathrm{ts}(t))$.

Finally, if the edge from s_i to s_{i+1} is a split edge, then the corresponding rule is $\mathsf{f}_{s_i}(\mathrm{ts}(s_i)) \to \mathsf{f}_{s_{i+1}}(\mathrm{ts}(s_{i+1}))$. However, here $\mathrm{ts}(s_i) = \mathrm{ts}(s_{i+1})$. As before, by Lemma 3.5

there exists a substitution σ with $\mathrm{ts}(s_i)\sigma = t$. Hence, by instantiating the rule with σ, we get $\mathsf{f}_{s_i}(\mathrm{ts}(t)) \hookrightarrow_{\mathcal{R}} \mathsf{f}_{s_{i+1}}(\mathrm{ts}(t))$. ∎

Now we can finally prove our main theorem.

Theorem 3.7 (Proving termination of JBC by ITRSs)**.** *If the ITRS corresponding to a termination graph is terminating, then the termination graph is terminating as well. Hence, then the original JBC program is also terminating for all concrete states t where $t \sqsubseteq s$ for some abstract state s in the termination graph.*

Proof. Assume that the termination graph has an infinite computation path $s_1^1, \ldots, s_1^{n_1}, s_2^1, \ldots, s_2^{n_2}, \ldots$. Thus, there is an infinite computation sequence $t_1, t_2, \ldots$ of concrete states where $t_i \sqsubseteq s_i^1$ for all i. By Lemma 4.10, this results in the following infinite rewrite sequence w.r.t. the ITRS $\mathcal{R}$ corresponding to the termination graph:

$$\mathsf{f}_{s_1^1}(\mathrm{ts}(t_1)) \hookrightarrow_{\mathcal{R}}^+ \mathsf{f}_{s_2^1}(\mathrm{ts}(t_2)) \hookrightarrow_{\mathcal{R}}^+ \mathsf{f}_{s_3^1}(\mathrm{ts}(t_3)) \hookrightarrow_{\mathcal{R}}^+ \ldots$$

∎

Aachener Informatik-Berichte

This list contains all technical reports published during the past five years. A complete list of reports dating back to 1987 is available from `http://aib.informatik.rwth-aachen.de/`. To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: biblio@informatik.rwth-aachen.de

2005-01 *	Fachgruppe Informatik: Jahresbericht 2004
2005-02	Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"
2005-03	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions
2005-04	Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem
2005-05	Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots
2005-06	Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information
2005-07	Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks
2005-08	Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut
2005-09	Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures
2005-10	Benedikt Bollig: Automata and Logics for Message Sequence Charts
2005-11	Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture
2005-12	Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems
2005-13	Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments
2005-14	Felix C. Freiling, Sukumar Ghosh: Code Stabilization
2005-15	Uwe Naumann: The Complexity of Derivative Computation
2005-16	Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)
2005-17	Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)
2005-18	Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"
2005-19	Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers

2005-20 Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.
2005-21 Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited
2005-22 Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins
2005-23 Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves
2005-24 Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks
2006-01 * Fachgruppe Informatik: Jahresbericht 2005
2006-02 Michael Weber: Parallel Algorithms for Verification of Large Systems
2006-03 Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler
2006-04 Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation
2006-05 Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F
2006-06 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color
2006-07 Thomas Colcombet, Christof Löding: Transforming structures by set interpretations
2006-08 Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs
2006-09 Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking
2006-10 Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed
2006-11 Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers
2006-12 Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning
2006-13 Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities
2006-14 Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"
2006-15 Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices

2006-16	Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness
2006-17	Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines
2007-01 *	Fachgruppe Informatik: Jahresbericht 2006
2007-02	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations
2007-03	Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase
2007-04	Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation
2007-05	Uwe Naumann: On Optimal DAG Reversal
2007-06	Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking
2007-07	Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications
2007-08	Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches
2007-09	Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption
2007-10	Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes
2007-11	Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke
2007-12	Uwe Naumann: An L-Attributed Grammar for Adjoint Code
2007-13	Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs
2007-14	Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes
2007-15	Volker Stolz: Temporal assertions for sequential and concurrent programs
2007-16	Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks
2007-17	René Thiemann: The DP Framework for Proving Termination of Term Rewriting
2007-18	Uwe Naumann: Call Tree Reversal is NP-Complete
2007-19	Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control
2007-20	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems
2007-21	Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains

2007-22 Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets
2008-01 * Fachgruppe Informatik: Jahresbericht 2007
2008-02 Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing
2008-03 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, Harald Zankl: Maximal Termination
2008-04 Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler
2008-05 Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations
2008-06 Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs
2008-07 Alexander Nyßen, Horst Lichter: The MeDUSA Reference Manual, Second Edition
2008-08 George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs
2008-09 George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs
2008-10 George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time
2008-11 George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows
2008-12 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages
2008-13 Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs
2008-14 Bastian Schlich: Model Checking of Software for Microcontrollers
2008-15 Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves
2008-16 Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study
2008-17 Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving
2008-18 Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems
2008-19 Dirk Wilking: Empirical Studies for the Application of Agile Methods to Embedded Systems
2009-02 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications
2009-03 Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices
2009-04 Daniel Klünder: Entwurf eingebetteter Software mit abstrakten Zustandsmaschinen und Business Object Notation

2009-05 George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs

2009-06 George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete

2009-07 Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I

2009-08 Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs

2009-09 Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem

2009-10 Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm

2009-11 Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs

2009-12 Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes

2009-13 Martin Zimmermann: Time-optimal Winning Strategies for Poset Games

2009-14 Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV'09)

2009-15 Joost-Pieter Katoen, Daniel Klink, Martin Neuhäußer: Compositional Abstraction for Stochastic Systems

2009-16 George B. Mertzios, Derek G. Corneil: Vertex Splitting and the Recognition of Trapezoid Graphs

2009-17 Carsten Kern: Learning Communicating and Nondeterministic Automata

2009-18 Paul Hänsch, Michaela Slaats, Wolfgang Thomas: Parametrized Regular Infinite Games and Higher-Order Pushdown Strategies

2010-02 Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time

2010-03 Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering

2010-07 George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de to obtain copies.