Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting*

Marc Brockschmidt, Carsten Otto, and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

Abstract

In [5, 15] we presented an approach to prove termination of non-recursive Java Bytecode (JBC) programs automatically. Here, JBC programs are first transformed to finite *termination graphs* which represent all possible runs of the program. Afterwards, the termination graphs are translated to term rewrite systems (TRSs) such that termination of the resulting TRSs implies termination of the original JBC programs. So in this way, existing techniques and tools from term rewriting can be used to prove termination of JBC automatically. In this paper, we improve this approach substantially in two ways:

(1) We extend it in order to also analyze *recursive* JBC programs. To this end, one has to represent call stacks of arbitrary size.

(2) To handle JBC programs with several methods, we *modularize* our approach in order to re-use termination graphs and TRSs for the separate methods and to prove termination of the resulting TRS in a modular way.

We implemented our approach in the tool AProVE. Our experiments show that the new contributions increase the power of termination analysis for JBC significantly.

1998 ACM Subject Classification D.1.5 - Object-oriented Programming, D.2.4 - Software/Program Verification, D.3.3 - Language Constructs and Features, F.3 - Logics and Meanings of Programs, F.4.2 - Grammars and Other Rewriting Systems, I.2.2 - Automatic Programming

Keywords and phrases termination, Java Bytecode, term rewriting, recursion

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.155

Category Regular Research Paper

1 Introduction

While termination of TRSs and logic programs was studied for decades, recently there have also been many results on termination of *imperative programs* (e.g., [3, 6, 7, 8]). However, these methods do not re-use the many existing termination techniques for TRSs and declarative languages. Therefore, in [5, 15] we presented the first rewriting-based approach for proving termination of a real imperative object-oriented language, viz. Java Bytecode [14].

We only know of two other automated methods to analyze JBC termination, implemented in the tools COSTA [2] and Julia [16]. They transform JBC into a constraint logic program by abstracting objects of dynamic data types to integers denoting their path-length (e.g., list objects are abstracted to their length). While this fixed mapping from objects to integers leads to high efficiency, it also restricts the power of these methods.

In contrast, in [5, 15] we represent data objects not by integers, but by *terms* which express as much information as possible about the objects. For example, list objects are represented by terms of the form $\mathsf{List}(t_1, \mathsf{List}(t_2, \ldots \mathsf{List}(t_n, \mathsf{null}) \ldots))$. In this way, we benefit from the fact

* Supported by the DFG grant GI 274/5-3 and the G.I.F. grant 966-116.6.

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 155–170

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

that rewrite techniques can automatically generate well-founded orders comparing arbitrary forms of terms. Moreover, by using TRSs with built-in integers [9], our approach is not only powerful for algorithms on user-defined data structures, but also for algorithms on pre-defined data types like integers. To obtain TRSs that are suitable for termination analysis, our approach first transforms a JBC program into a *termination graph* which represents all possible runs of the program. These graphs handle all aspects of JBC that cannot easily be expressed in term rewriting (e.g., side effects, cyclic data objects, object-orientation, etc.). Afterwards, a TRS is generated from the termination graph. As proved in [5, 15], termination of this TRS implies termination of the original JBC program.

We implemented this approach in our tool AProVE [10] and in the *International Termination Competitions*,[1] AProVE achieved competitive results compared to Julia and COSTA.

However, a significant drawback was that (in contrast to techniques that abstract objects to integers [2, 8, 16]), our approach in [5, 15] could not deal with *recursion*. The problem is that for recursive methods, the size of the call stack usually depends on the input arguments. Hence, to represent all possible runs, this would lead to termination graphs with infinitely many states (since [5, 15] used no abstraction on call stacks). An abstraction of call stacks is non-trivial due to possible aliasing between references in different stack frames.

In the current paper, we solve these problems. Instead of directly generating a termination graph for the whole program as in [5, 15], in Sect. 2 we construct a separate termination graph for each method. These graphs can be combined afterwards. Similarly, one can also combine the TRSs resulting from these "method graphs" (Sect. 3). As demonstrated by our implementation in AProVE (Sect. 4), our new approach has two main advantages over [5, 15]:

(1) We can now analyze *recursive* methods, since our new approach can deal with call stacks that may grow unboundedly due to method calls.

(2) We obtain a *modular* approach, because one can re-use a method graph (and the rewrite rules generated from it) whenever the method is called. So in contrast to [5, 15], now we generate TRSs that are amenable to modular termination proofs.

See [4] for all proofs, and see [1] for experimental details and our previous papers [5, 15].

2 From Recursive JBC to Modular Termination Graphs

To analyze termination of a set of desired initial (concrete) program states, we represent this set by a suitable *abstract state* which is the initial node of the termination graph. Then this state is *evaluated symbolically*, which leads to its child nodes in the termination graph.

Our approach is restricted to verified[2] sequential JBC programs. To simplify the presentation in this paper, we exclude arrays, static class fields, interfaces, and exceptions. We also do not describe the annotations introduced in [5, 15] to handle complex sharing effects. With such annotations one can for example also model "unknown" objects with arbitrary sharing behavior as well as cyclic objects. Extending our approach to such constructs is easily possible and has been done for our implementation in the termination prover AProVE. However, currently our implementation has only minimal support for features like floating point arithmetic, strings, static initialization of classes, instances of `java.lang.Class`, reflection, etc.

Sect. 2.1 presents our notion of *states*. Sect. 2.2 introduces *termination graphs* for one method and Sect. 2.3 shows how to re-use these graphs for programs with many methods.

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

[2] The bytecode verifier of the JVM [14] ensures certain properties of the code that are useful for our analysis, e.g., that there is no overflow or underflow of the operand stack.

2.1 States

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

```
00: aload_0       // load this to opstack
01: getfield n    // load this.n to opstack
04: ifnonnull 26  // jump to 26 if n is not null
07: iload_1       // load i to opstack
08: ifgt 12       // jump to 12 if i > 0
11: return        // return (without value)
12: aload_0       // load this to opstack
13: new List      // create new List object
16: dup           // duplicate top stack entry
17: invokespecial <init> // invoke constructor
20: putfield n    // write new List to field n
23: iinc 1, -1    // decrement i by 1
26: aload_0       // load this to opstack
27: getfield n    // load this.n to opstack
30: iload_1       // load i to opstack
31: invokevirtual appE   // recursive call
34: return        // return (without value)
```

Consider the recursive method `appE` (presented in both Java and JBC). We use a class `List` where the field `n` points to the next list element. For brevity, we omitted a field for the value of a list element. The method `appE` recursively traverses the list to its end, where it attaches `i` fresh elements (if `i > 0`).

Fig. 1 displays an abstract state of `appE`. A state consists of a sequence of *stack frames* and the *heap*, i.e., States = SFrames* ×Heap. The state in Fig. 1 has just a single stack frame "o_1, i_3 | `0` | `t`:o_1,`i`:i_3 | ε" which consists of four components. Its first component o_1, i_3 are the *input arguments*, i.e., those objects that are "visible" from outside the analyzed method. This component is new compared to [5, 15] and it is needed to denote later on which of these objects have been modified by side effects during the execution of the method. In our example, `appE` has two input arguments, viz. the implicit formal parameter `this` (whose value is o_1) and the formal parameter `i` with value i_3. In contrast to JBC, we also represent integers by references and adapt the semantics of all instructions to handle this correctly. So $o_1, i_3 \in$ Refs, where Refs is an infinite set of names for addresses on the heap.

o_1, i_3 | `0` | `t`:o_1,`i`:i_3 | ε
o_1:`List`(`n`=o_2) i_3:$\mathbb{Z}$
o_2:`List`(?)

Figure 1 State

The second component `0` of the stack frame is the *program position* (from ProgPos), i.e., the index of the next instruction. So `0` means that evaluation continues with `aload_0`.

The third component is the list of values of *local variables*, i.e., LocVar = Refs*. To ease readability, we do not only display the values, but also the variable names. For example, the name of the first local variable `this` is shortened to `t` and its value is o_1.

The fourth component is the *operand stack* to store temporary results, i.e., OpStack = Refs*. Here, ε is the empty stack and "o_8, o_1" denotes a stack with o_8 on top.

So the set of all *stack frames* is SFrames = InpArgs × ProgPos × LocVar × OpStack. As mentioned, the *call stack* of a state can consist of several stack frames. If a method calls another method, then a new frame is put on top of the call stack.

In addition to the call stack, a state contains information on the *heap*. The heap is a partial function mapping references to their value, i.e., Heap = Refs → Integers ∪ Instances ∪ Unknown ∪ {`null`}. We depict a heap by pairs of a reference and a value, separated by ":".

Integers are represented by intervals, i.e., Integers = $\{\{x \in \mathbb{Z} \mid a \leq x \leq b\} \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \leq b\}$. We abbreviate $(-\infty, \infty)$ by $\mathbb{Z}$, $[1, \infty)$ by $[> 0]$, etc. So "$i_3 : \mathbb{Z}$" means that any integer can be at the address i_3. Since current TRS tools cannot handle 32-bit `int`-numbers, we treat all numeric types like `int` as the infinite set of all integers.

To represent Instances (i.e., objects) of some class, we store their type and the values of their fields, i.e., Instances = Classnames ×(FieldIDs → Refs). Classnames contains the names of all classes. FieldIDs is the set of all field names. To prevent ambiguities, in general the FieldIDs also include the respective class name. For all $(cl, f) \in$ Instances, the function f is defined for all fields of cl and of its superclasses. Thus, "o_1 : `List`(`n` = o_2)" means that at the address o_1, there is a `List` object whose field `n` has the value o_2.

UNKNOWN = CLASSNAMES ×{?} represents `null` and all tree-shaped objects for which we only have type information. In particular, UNKNOWN objects are acyclic and do not share parts of the heap with any objects at the other references in the state. For example, "o_2 : `List`(?)" means that o_2 is `null` or an instance of `List` (or a subtype of `List`).

Every *input argument* has a boolean flag, where *false* indicates that it may have been modified (as a side effect) by the current method. Moreover, we store which formal parameter of the method corresponds to this input argument. So in Fig. 1, the full input arguments are $(o_1, \text{LV}_{0,0}, \mathit{true})$ and $(i_3, \text{LV}_{0,1}, \mathit{true})$. Here, $\text{LV}_{i,j}$ is the *position* of the j-th local variable in the i-th stack frame. When the top stack frame (i.e., frame 0) is at program position `0` of a method, then its 0-th and 1-st local variables (at positions $\text{LV}_{0,0}$ and $\text{LV}_{0,1}$) correspond to the first and second formal parameter of the method. Formally, $\text{INPARGS} = 2^{\text{REFS} \times \text{SPOS} \times \mathbb{B}}$.

A *state position* $\pi \in \text{SPOS}(s)$ is a sequence starting with $\text{LV}_{i,j}$, $\text{OS}_{i,j}$ (for operand stack entries), or $\text{IN}_{i,\tau}$ (for input arguments (r, τ, b) in the i-th stack frame), followed by a sequence of FIELDIDS. This sequence indicates how to access a particular object.

▶ **Definition 2.1** (State Positions). Let $s = (\langle fr_0, \ldots, fr_n \rangle, h) \in \text{STATES}$ where $fr_i = (in_i, pp_i, lv_i, os_i)$. Then $\text{SPOS}(s)$ is the smallest set containing all the following sequences π:

- $\pi = \text{LV}_{i,j}$ where $0 \le i \le n$, $lv_i = \langle l_0, \ldots, l_m \rangle$, $0 \le j \le m$. Then $s|_\pi$ is l_j.
- $\pi = \text{OS}_{i,j}$ where $0 \le i \le n$, $os_i = \langle o_0, \ldots, o_k \rangle$, $0 \le j \le k$. Then $s|_\pi$ is o_j.
- $\pi = \text{IN}_{i,\tau}$ where $0 \le i \le n$ and $(r, \tau, b) \in in_i$. Then $s|_\pi$ is r.
- $\pi = \pi' v$ for some $v \in \text{FIELDIDS}$ and some $\pi' \in \text{SPOS}(s)$ where $h(s|_{\pi'}) = (cl, f) \in \text{INSTANCES}$ and where $f(v)$ is defined. Then $s|_\pi$ is $f(v)$.

The *references in the state* s are defined as $\mathit{Ref}(s) = \{s|_\pi \mid \pi \in \text{SPOS}(s)\}$.

So for the state s in Fig. 1, we have $s|_{\text{LV}_{0,0}} = s|_{\text{IN}_{0,\text{LV}_{0,0}}} = o_1$, $s|_{\text{LV}_{0,0}\,\texttt{n}} = s|_{\text{IN}_{0,\text{LV}_{0,0}}\,\texttt{n}} = o_2$, etc.

2.2 Termination Graphs for a Single Method

In Fig. 2, we construct the termination graph of `appE`. The state in Fig. 1 is its initial state A, i.e., we analyze termination of `appE` for acyclic lists of arbitrary length and any integer.

In A, `aload_0` loads the value of the 0-th local variable `this` on the operand stack. So A is connected by an *evaluation edge* to a state with program position `1` (omitted from Fig. 2 due to space reasons, i.e., dotted arrows abbreviate *several* steps). Then "`getfield n`" replaces o_1 on the operand stack by the value o_2 of its field `n`, resulting in state B. The value `List`(?) of o_2 does not provide enough information to evaluate `ifnonnull`. Thus, we perform an *instance refinement* [5, Def. 5] resulting in C and D, i.e., a case analysis whether o_2's value is `null`. *Refinement edges* are denoted by dashed lines. In C, we assume that o_2's value is not `null`. Thus, we replace o_2 by a fresh[3] reference o_4, which points to `List`(`n` = o_5). Hence, we can now evaluate `ifnonnull` and jump to instruction `26` in state M.

In D, we assume that o_2's value is `null`, i.e., "o_1 : `List`(`n` = o_2)" and "o_2 : `null`". To ease the presentation, in such states we simply replace all occurrences of o_2 with `null`. After evaluating the instruction "`ifnonnull 26`", in the next state (which we omitted from Fig. 2 for space reasons), the instruction "`iload_1`" loads the value of `i` on the operand stack. This results in state E. Now again we do not have enough information to evaluate `ifgt`. Thus, we perform an *integer refinement* [5, Def. 1], leading to states F (if `i <= 0`) and H.

[3] We rename references that are refined to ease the formal definition of the refinements, cf. [5].

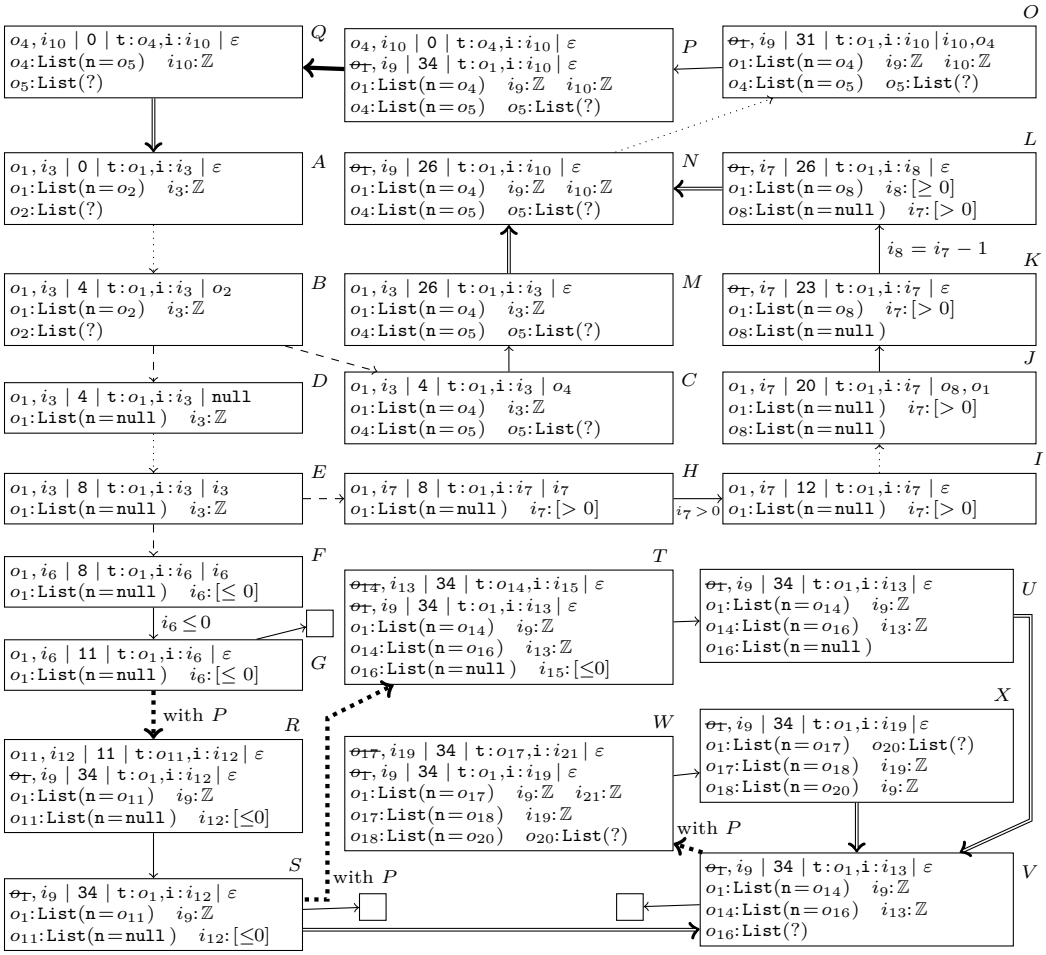


Figure 2 Termination Graph of `appE`

In F, we evaluate `ifgt`, leading to G. We label the edge from F to G with the condition $i_6 \leq 0$ of this case. This label will be used when generating a TRS from the termination graph. States like G that have only a single stack frame which is at a `return` position are called *return states*. Thus, we reach a *program end*, denoted by $\square$. From H, we jump to instruction `12` in I and label the edge with $i_7 > 0$. In I, o_1 is pushed on the operand stack. Afterwards, we create another list element o_8, where we skipped the constructor call in Fig. 2. In K, o_8 has been written to the field `n` of o_1. This is a *side effect* on an object that is visible from outside the method (since o_1 is an input argument). Hence, in K we set the boolean flag for o_1 to *false* (depicted by crossing out the input argument o_1).

In L, the value of the 1-st local variable `i` is decremented by 1. In contrast to JBC, we represent primitive data types by references. Hence, we introduce a fresh reference i_8, pointing to the adapted value. Since i_7's value did not change, i_7 is not crossed out.

State L is similar to the state M we obtained from the other branch of our first refinement. To simplify the graph, we create a *generalized* state N, which represents a superset of all concrete states represented by L or M. N is almost like M (up to renaming of references) and only differs in the information about input arguments, which is taken from L. We draw *instance edges* (double arrows) from L and M to N and only consider N in the remainder.

In O, we have loaded `this.n` and `i` on the operand stack and invoke `appE` on these values.

So in P, a second stack frame is pushed on top of the previous one. States like P that contain at least two frames where the top frame is at the start of a method are *call states*.

We now introduce a new approach to represent call stacks of arbitrary size by *splitting up* call stacks. Otherwise, for recursive methods the call stack could grow unboundedly and we would obtain an infinite termination graph. So P has a *call edge* (thick arrow) to Q which only contains P's top stack frame. Since Q is identical to A (modulo renaming), we do not have to analyze `appE` again, but simply draw an instance edge from Q to A.

Up to now A only represented concrete states where `appE` was called "directly". However, now A can also be reached from a "method call" in P. Hence, now A and the other abstract states s of `appE`'s termination graph also represent states where `appE` was called "recursively", i.e., where below the stack frames of s, one has the stack frames of P (only P's top frame is replaced by the frames of s).[4] For each *return state* we now consider two cases: Either there are no further frames below the top frame (then one reaches a leaf of the termination graph) or else, there are further frames below the top (which result from the method call in P). Hence, for every return state like G, we now create an additional successor state R (the *context concretization of G with P*), connected by a *context concretization edge* (a thick dotted arrow). R has the same stack frame as G (up to renaming), but below we add the call stack of P (without P's top frame that corresponded to the method call).

In R, `appE`'s recursive call has just reached the `return` statement at index `11`. Here, we identified o_1 and i_6 from state G with o_4 and i_{10} from P and renamed them to o_{11} and i_{12}. We now consider which information we have about R's heap. According to state G, the input arguments of `appE`'s recursive call were not modified during the execution of this recursive call. Thus, for the input arguments o_{11} and i_{12} in R, we can use *both* the information on o_1 and i_6 in G and on o_4 and i_{10} in P. According to G, o_1 is a list of length 1 and $i_6 \leq 0$. According to P, o_4 has at least length 1 and i_{10} is arbitrary. Hence, in R we can take the *intersection* of this information and deduce that o_{11} has length 1 and $i_{12} \leq 0$. (So in this example, the intersection of G's and P's information coincides with the information in G.)

When constructing termination graphs, context concretization is only needed for return states. But to formulate Thm. 2.3 on the soundness of termination graphs later on, in Def. 2.2 we introduce context concretization for arbitrary states $s = (\langle fr_0, \ldots, fr_n \rangle, h)$. So s results from evaluating the method in the bottom frame fr_n (i.e., fr_{n-1} was created by a call in fr_n, fr_{n-2} was created by a call in fr_{n-1}, etc.). Context concretization of s with a call state $\overline{s} = (\langle \overline{fr}_0, \ldots, \overline{fr}_m \rangle, \overline{h})$ means that we consider the case where fr_n results from a call in $\overline{fr}_1$. Thus, the top frame $\overline{fr}_0$ of $\overline{s}$ is at the start of some method and the bottom frame fr_n of s must be at an instruction of the *same* method. Moreover, for all input arguments $(\overline{r}, \tau, \overline{b})$ in $\overline{fr}_0$ there must be a *corresponding* input argument (r, τ, b) in fr_n.[5] To ease the formalization, let $Ref(s)$ and $Ref(\overline{s})$ be disjoint. For instance, if s is G and $\overline{s}$ is P, we can mark the references by ^{G} and ^{P} to achieve disjointness (e.g., $o_1^G \in Ref(G)$ and $o_1^P \in Ref(P)$).

Then we add the frames $\overline{fr}_1, \ldots, \overline{fr}_m$ of the call state $\overline{s}$ below the call stack of s to obtain a new state $\tilde{s}$ with the call stack $\langle fr_0\sigma, \ldots, fr_n\sigma, \overline{fr}_1\sigma, \ldots, \overline{fr}_m\sigma \rangle$. The *identification substitution* σ identifies every input argument $\overline{r}$ of $\overline{fr}_0$ with the corresponding input argument r of fr_n. If the boolean flag for the input argument r in s is *false*, then this object may have changed during the evaluation of the method and in $\tilde{s}$, we should only use the information

[4] For example, A now represents all states with call stacks $\langle fr^A, fr_1^P, fr_1^P, \ldots, fr_1^P \rangle$ where fr^A is A's stack frame and $fr_1^P, fr_1^P, \ldots, fr_1^P$ are copies of P's bottom frame (in which references may have been renamed). So A represents states where `appE` was called within an arbitrary high context of recursive calls.

[5] This obviously holds for all input arguments corresponding to formal parameters of the method, but Sect. 2.3 will illustrate that sometimes $\overline{fr}_0$ may have additional input arguments.

from s. But if the flag is *true*, then the object did not change. Then, both the information in s and in $\overline{s}$ about this object is correct and for $\tilde{s}$, we take the intersection of this information. In our example, $\sigma(o_1^G) = \sigma(o_4^P) = o_{11}^R$ and $\sigma(i_6^G) = \sigma(i_{10}^P) = i_{12}^R$. Since the flags of the input arguments o_1^G and i_6^G are *true*, for o_{11}^R and i_{12}^R, we intersect the information from G and P.

If we identify r and $\overline{r}$, and both point to INSTANCES, then we may also have to identify the references in their fields. To this end, we define an equivalence relation $\equiv \subseteq \text{REFS} \times \text{REFS}$ where "$r \equiv \overline{r}$" means that r and $\overline{r}$ are identified. Let $r \equiv \overline{r}$ and let r be no input argument in s with the flag *false*. If r points to (cl, f) in s and $\overline{r}$ points to $(cl, \overline{f})$ in $\overline{s}$, then all references in the fields v of cl and its superclasses also have to be identified, i.e., $f(v) \equiv \overline{f}(v)$.

To illustrate this in our example, note that we abbreviated the information on G's heap in Fig. 2. In reality we have "$o_1^G : \texttt{List}(\texttt{n} = o_2^G)$", "$o_2^G : \texttt{null}$", and "$i_6^G : [\leq 0]$". Hence, we do not only obtain $i_6^G \equiv i_{10}^P$ and $o_1^G \equiv o_4^P$, but since o_1^G's boolean flag is not *false*, we also have to identify the references at the field `n` of the object, i.e., $o_2^G \equiv o_5^P$.

Let ρ be an injective function that maps each $\equiv$-equivalence class to a fresh reference. We define the *identification substitution* σ as $\sigma(r) = \rho([r]_\equiv)$ for all $r \in \mathit{Ref}(s) \cup \mathit{Ref}(\overline{s})$. So we map equivalent references to the same new reference and we map non-equivalent references to different references. To construct $\tilde{s}$, if $r \in \mathit{Ref}(s)$ points to an object which was not modified by side effects during the execution of the called method (i.e., where the flag is not *false*), we intersect all information in s and $\overline{s}$ on the references in $[r]_\equiv$. For all other references in $\mathit{Ref}(s)$ resp. $\mathit{Ref}(\overline{s})$, we only take the information from s resp. $\overline{s}$ and apply σ.

In our example, we have the equivalence classes $\{o_1^G, o_4^P\}$, $\{o_2^G, o_5^P\}$, $\{i_6^G, i_{10}^P\}$, $\{o_1^P\}$, and $\{i_9^P\}$. For these classes we choose the new references o_{11}^R, o_2^R, i_{12}^R, o_1^R, i_9^R, and obtain $\sigma = \{o_1^G/o_{11}^R, o_4^P/o_{11}^R, o_2^G/o_2^R, o_5^P/o_2^R, i_6^G/i_{12}^R, i_{10}^P/i_{12}^R, o_1^P/o_1^R, i_9^P/i_9^R\}$. The information for o_{11}^R, o_2^R, and i_{12}^R is obtained by intersecting the respective information from G and P. The information for o_1^R and i_9^R is taken over from P (by applying σ).

Def. 2.2 also introduces the concept of *intersection* formally. If $r \in \mathit{Refs}(s)$, $\overline{r} \in \mathit{Refs}(\overline{s})$, and h resp. $\overline{h}$ are the heaps of s resp. $\overline{s}$, then intuitively, $h(r) \cap \overline{h}(\overline{r})$ consists of those values that are represented by both $h(r)$ and $\overline{h}(\overline{r})$. For example, if $h(r) = [\geq 0] = (-1, \infty)$ and $\overline{h}(\overline{r}) = [\leq 0] = (-\infty, 1)$, then the intersection is $(-1, 1) = [0, 0]$. Similarly, if $h(r)$ or $\overline{h}(\overline{r})$ is `null`, then their intersection is again `null`. If $h(r), \overline{h}(\overline{r})$ are UNKNOWN instances of classes cl_1, cl_2, then their intersection is an UNKNOWN instance of the more special class $\min(cl_1, cl_2)$. Here, $\min(cl_1, cl_2) = cl_1$ if cl_1 is a (not necessarily proper) subtype of cl_2 and $\min(cl_1, cl_2) = cl_2$ if cl_2 is a subtype of cl_1. Otherwise, cl_1 and cl_2 are called *orthogonal*. If $h(r) \in$ UNKNOWN and $\overline{h}(\overline{r}) \in$ INSTANCES, then their intersection is from INSTANCES using the more special type. Finally, if both $h(r), \overline{h}(\overline{r}) \in$ INSTANCES with the same type, then their intersection is again from INSTANCES. For the references in its fields, we use the identification substitution σ that renames equivalent references to the same new reference.

Note that one may also have to identify different references in the *same* state. For example, $\overline{s}$ could have the input arguments $(\overline{r}, \tau_1, \overline{b})$ and $(\overline{r}, \tau_2, \overline{b})$ with the corresponding input arguments (r_1, τ_1, b_1) and (r_2, τ_2, b_2) in s. Then $\overline{r} \equiv r_1 \equiv r_2$. Note that if $r_1 \neq r_2$ are references from the *same* state where $h(r_1) \in$ INSTANCES, then they point to different objects (i.e., then $h(r_1) \cap h(r_2)$ is empty). Similarly, if $h(r_1), h(r_2) \in$ UNKNOWN, then they also point to different objects or to `null` (i.e., then $h(r_1) \cap h(r_2)$ is `null`).

▶ **Definition 2.2** (Context Concretization). Let $s = (\langle fr_0, \ldots, fr_n \rangle, h)$ and let $\overline{s} = (\langle \overline{fr}_0, \ldots, \overline{fr}_m \rangle, \overline{h})$ be a call state where fr_n and $\overline{fr}_0$ correspond to the same method. (So $\overline{fr}_0$ is at the start of the method and fr_n can be at any position of the method.) Let in_n resp. $\overline{in}_0$ be the input arguments of fr_n resp. $\overline{fr}_0$, and let $\mathit{Ref}(s) \cap \mathit{Ref}(\overline{s}) = \varnothing$. For every input argument $(\overline{r}, \tau, \overline{b}) \in \overline{in}_0$ there must be a *corresponding* input argument $(r, \tau, b) \in in_n$ (i.e., with the same

position τ), otherwise there is no context concretization of s with $\overline{s}$. Let $\equiv \subseteq \text{REFS} \times \text{REFS}$ be the smallest equivalence relation which satisfies the following two conditions:

- if $(\overline{r}, \tau, \overline{b}) \in \overline{in}_0$ and $(r, \tau, b) \in in_n$, then $r \equiv \overline{r}$.
- if $r \in \mathit{Ref}(s)$, $\overline{r} \in \mathit{Ref}(\overline{s})$, $r \equiv \overline{r}$, and there is no $(r, \tau, \mathit{false}) \in in_n$, then $h(r) = (cl, f)$ and $\overline{h}(\overline{r}) = (cl, \overline{f})$ implies that $f(v) \equiv \overline{f}(v)$ holds for all fields v of cl and its superclasses.

Let $\rho : \text{REFS}/_{\equiv} \to \text{REFS}$ be an injective mapping to fresh references $\notin \mathit{Ref}(s) \cup \mathit{Ref}(\overline{s})$ and let $\sigma(r) = \rho([r]_{\equiv})$ for all $r \in \mathit{Ref}(s) \cup \mathit{Ref}(\overline{s})$. Then the *context concretization of s with $\overline{s}$* is the state $\tilde{s} = (\langle \mathit{fr}_0\sigma, \ldots, \mathit{fr}_n\sigma, \overline{\mathit{fr}}_1\sigma, \ldots, \overline{\mathit{fr}}_m\sigma \rangle, \tilde{h})$. Here, we define $\tilde{h}(\sigma(r))$ to be

- $h(r_1) \cap \ldots \cap h(r_k) \cap \overline{h}(\overline{r}_1) \cap \ldots \cap \overline{h}(\overline{r}_d)$, if $[r]_{\equiv} \cap \mathit{Ref}(s) = \{r_1, \ldots, r_k\}$, $[r]_{\equiv} \cap \mathit{Ref}(\overline{s}) = \{\overline{r}_1, \ldots, \overline{r}_d\}$, and there is no input argument $(r_i, \tau, \mathit{false}) \in in_n$
- $h(r_1) \cap \ldots \cap h(r_k)$, if $[r]_{\equiv} \cap \mathit{Ref}(s) = \{r_1, \ldots, r_k\}$, and there is an $(r_i, \tau, \mathit{false}) \in in_n$

If the intersection is empty, then there is no concretization of s with $\overline{s}$. Moreover, whenever there is an input argument $(\overline{r}, \tau, \overline{b}) \in \overline{in}_0$ with corresponding input argument $(r, \tau, \mathit{false}) \in in_n$, then for all input arguments $(\overline{r}', \tau', \overline{b}')$ in lower stack frames of $\overline{s}$ where $\overline{r}'$ reaches[6] $\overline{r}$ in $\overline{h}$, the flag $\overline{b}'$ must be replaced by *false* when creating the context concretization $\tilde{s}$. In other words, in the lower stack frame of $\tilde{s}$, we then have the input argument $(\overline{r}'\sigma,\ \tau',\ \mathit{false})$.

Finally, for all $s_1, \ldots, s_k \in \{s, \overline{s}\}$ where h_i is the heap of s_i, and for all pairwise different references $r_1, \ldots, r_k$ with $r_i \in \mathit{Ref}(s_i)$ where $r_1 \equiv \ldots \equiv r_k$, we define $h_1(r_1) \cap \ldots \cap h_k(r_k)$ to be $h_1(r_1)\sigma$ if $k = 1$. Otherwise, $h_1(r_1) \cap \ldots \cap h_k(r_k)$ is

- $(\max(a_1, \ldots, a_k), \min(b_1, \ldots, b_k))$, if all $h_i(r_i) = (a_i, b_i) \in \text{INTEGERS}$ and $\max(a_1, \ldots, a_k) + 1 < \min(b_1, \ldots, b_k)$
- `null`, if all $h_i(r_i) \in \text{UNKNOWN} \cup \{\texttt{null}\}$ and at least one of them is `null`
- `null`, if all $h_i(r_i) \in \text{UNKNOWN}$ and there are $j \neq j'$ with $s_j = s_{j'}$
- `null`, if $k = 2$, $h_1(r_1) = (cl_1, ?)$, $h_2(r_2) = (cl_2, ?)$ and cl_1, cl_2 are orthogonal
- $(\min(cl_1, cl_2), ?)$, if $k = 2$, $s_1 \neq s_2$, $h_1(r_1) = (cl_1, ?)$, $h_2(r_2) = (cl_2, ?)$, and cl_1, cl_2 are not orthogonal
- (cl, f), if $k = 2$, $s_1 \neq s_2$, $h_1(r_1) = (cl, f_1), h_2(r_2) = (cl, f_2) \in \text{INSTANCES}$. Here, $f(v) = \sigma(f_1(v)) = \sigma(f_2(v))$ for all fields v of cl and its superclasses.
- $(\min(cl_1, cl_2), f)$, if $k = 2$, $s_1 \neq s_2$, $h_1(r_1) = (cl_1, ?)$, $h_2(r_2) = (cl_2, f_2)$, and cl_1, cl_2 are not orthogonal. Here, $f(v) = \sigma(f_2(v))$ for all fields v of cl_2 and its superclasses. If cl_1 is a subtype of cl_2, then for those fields v of cl_1 and its superclasses where f_2 is not defined, $f(v)$ returns a fresh reference r_v where $\tilde{h}(r_v) = (-\infty, \infty)$ if the field v has an integer type and $\tilde{h}(r_v) = (cl_v, ?)$ if the type of the field v is some class cl_v. The case where $h_1(r_1) \in \text{INSTANCES}$ and $h_2(r_2) \in \text{UNKNOWN}$ is analogous.

In all other cases, $h_1(r_1) \cap \ldots \cap h_k(r_k)$ is empty.

We continue with constructing `appE`'s termination graph. When evaluating R, the top frame is removed from the call stack and due to the lower stack frame, we now reach a new return state S. As above, for every return state, we have to create a new context concretization T which is like the call state P, but where P's top stack frame is replaced by the stack frame of the return state S. We use an identification substitution σ which maps o_1^S and o_4^P to o_{14}^T, i_9^S and i_{10}^P to i_{13}^T, i_{12}^S to i_{15}^T, o_{11}^S to o_{16}^T, o_1^P to o_1^T, and i_9^P to i_9^T. The value of o_{14}^T (i.e., o_1^S and o_4^P) may have changed during the execution of the top frame (as o_1^S is

[6] We say that $\overline{r}'$ *reaches* $\overline{r}$ in $\overline{h}$ iff there is a position $\pi_1\,\pi_2 \in \text{SPOS}(\overline{s})$ such that $\overline{s}|_{\pi_1} = \overline{r}'$ and $\overline{s}|_{\pi_1\,\pi_2} = \overline{r}$.

crossed out). Hence, we only take the value from S, i.e., o_{14}^T is a list of length 2. For i_{13}^T, we intersect the information on i_9^S and on i_{10}^P. The information on i_{15}^T is taken from i_{12}^S and the information on o_1^T resp. i_9^T is taken from o_1^P resp. i_9^P (where σ is applied).

When evaluating T, the top frame is removed and we reach a new return state U. If we continued in this way, we would perform context concretization on U again, etc. Then the construction would not finish and we would get an infinite termination graph.

To obtain finite graphs, we use the heuristic to generalize all return states with the same program position to one common state, i.e., only one of them may have no outgoing instance edge. Then this generalized state can be used instead of the original ones. In S, `this` is a list of length 2, whereas in U, `this` has length 3. Moreover, `i` ≤ 0 in S, whereas `i` is arbitrary in U. Therefore, we generalize S and U to a new state V where `this` has length ≥ 2 and `i` is arbitrary. Now T and U are not needed anymore and could be removed.

As V is a return state, we have to create a new successor W by context concretization, which is like the call state P, but where P's top frame is replaced by V's frame (analogous to the construction of T). Evaluating W leads to X, which is an instance of V. Thus, we draw an instance edge from X to V and the termination graph construction is finished.

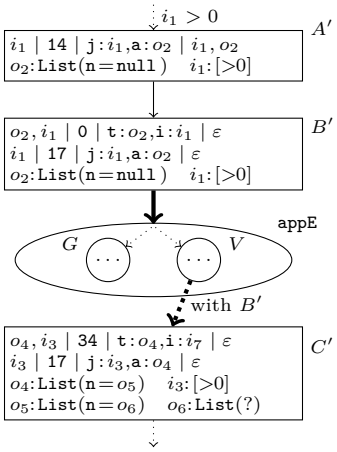
In general, a state s' is an *instance* of a state s (denoted $s' \sqsubseteq s$) if all concrete states represented by s' are also represented by s. For a formal definition of "$\sqsubseteq$", we refer to [5, Def. 3] and [15, Def. 2.3]. The only condition that has to be added to this definition is that for every input argument (r', τ, b') in the i-th frame of s', there must also be a corresponding input argument (r, τ, b) in the i-th frame of s, where $b' = \mathit{false}$ implies $b = \mathit{false}$.

However in [5, 15], $s' \sqsubseteq s$ only holds if s' and s have the same call stack size. In contrast, we now also allow larger call stacks in s' and define $s' \sqsubseteq s$ iff a state $\tilde{s}$ can be obtained by repeated context concretization from s, where s' and $\tilde{s}$ have the same call stack size and $s' \sqsubseteq \tilde{s}$. For example, $P \sqsubseteq A$, although P has two and A only has one stack frame, since context concretization of A (with P) yields a state $\tilde{A}$ which is a renaming of P (thus, $P \sqsubseteq \tilde{A}$).

2.3 Termination Graphs for Several Methods

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

Termination graphs for a method can be re-used whenever the method is called. To illustrate this, consider a method `cappE` which calls `appE`. It constructs a new `List a`, checks if the formal parameter `j` is `> 0`, and calls `a.appE(j)` to append `j` elements to `a`. Then, if `a.n` is `null`, one enters a non-terminating loop. But as `j > 0`, our analysis can detect that after the call `a.appE(j)`, the list `a.n` is not `null`. Hence, the loop is never executed and `cappE` is terminating.



In `cappE`'s termination graph, after constructing the new `List` and checking `j > 0`, one reaches A'. The call of `appE` leads to the call state B', whose top frame is at position `0` of `appE`. As in the step from P to Q in Fig. 2, we now split the call stack. The resulting state (with only B''s top frame) is connected by an instance edge to the initial state A of `appE`'s termination graph, i.e., we re-use the graph of Fig. 2. Recall that for every call state $\overline{s}$ that calls `appE` and each return state s in `appE`'s termination graph, we perform context concretization of s with $\overline{s}$. In fact, one can restrict this to return states $\overline{s}$ without outgoing instance edges (i.e., to G and V).

Now we have another call state B' which calls `appE`. G has no context concretization with B', as the second input

argument is ≤ 0 in G and > 0 in B' (i.e., the intersection is empty). Context concretization of V with B' yields state C'. Here, $i_3^{C'}$ results from intersecting i_9^V and $i_1^{B'}$, whereas $o_4^{C'}$ is taken over from o_1^V (thus in C', `a.n` is not `null` and hence, the `while` loop is not executed).[7]

To define termination graphs formally, in [5, Def. 6] we extended JBC-evaluation to abstract states, i.e., "$s \overset{SyEv}{\longrightarrow} s'$" means that s *symbolically evaluates* to s'. We now extend [5, Def. 6] to handle *input arguments*. Input arguments remain unchanged by symbolic evaluation, except when evaluating `putfield` or invoking a method. If evaluation of a `putfield` instruction changes an object at a position $\text{IN}_{i,\tau}\ \pi$, then we set the boolean flag b of the input argument (r, τ, b) in the i-th stack frame to *false* (cf. $J \overset{SyEv}{\longrightarrow} K$ in Fig. 2).

Now we explain how to create the input arguments for new stack frames which are generated when invoking a method. In general, one may need more input arguments than the method's formal parameters. To see this, consider a variant of `cappE`, where before the call of `appE`, we add the instruction "`List b = a.n = new List();`". Thus, now `a` is a list of length 2 and `b` also points to `a`'s second element. Hence, in state A' we now have the local variables "$\texttt{j}{:}i_1$, $\texttt{a}{:}o_2$, $\texttt{b}{:}o_3$" where "$o_2 : \texttt{List}(\texttt{n} = o_3)$" and "$o_3 : \texttt{List}(\texttt{n} = \texttt{null})$". As before, `appE` is called with the arguments o_2 and i_1 and its execution modifies the object at o_2 as a side effect. However, due to this, the object at o_3 is modified *as well*. We have to take this into account, because after the execution of `appE`, the object at o_3 is still accessible via the local variable `b`. So here the execution of a called method has a side effect on objects that are visible from lower frames of the call stack.

Recall that the purpose of the *input arguments* is to describe which objects may have changed (as a side effect) during the execution of the method. Therefore in B', we now have to add o_3 as an additional input argument when calling `appE`. More precisely, the three input arguments of B' would be $(o_2, \text{LV}_{0,0}, \mathit{true})$, $(i_1, \text{LV}_{0,1}, \mathit{true})$, and $(o_3, \text{LV}_{0,0}\ \texttt{n}, \mathit{true})$ (corresponding to the field `n` of `appE`'s first formal parameter).

Consequently, we now have to re-process the termination graph of `appE` to obtain a variant where the states have three input arguments. The stack frame of V would then be "$\sout{o_1}, i_9, \sout{o_{14}}, \mid 34 \mid \texttt{t}{:}o_1, \texttt{i}{:}i_{13} \mid \varepsilon$". Hence, in the context concretization of V with B' (where o_{14}^V is identified with $o_3^{B'}$), the information on $o_3^{B'}$ is longer valid, but instead one has to use o_{14}^V. Thus in C', the value of `b` is no longer "$o_3 : \texttt{List}(\texttt{n} = \texttt{null})$", but "$o_5 : \texttt{List}(\texttt{n} = o_6^{C'})$", where $o_6^{C'}$'s value is a copy of V's value for o_{16}^V, i.e., `List`(?).

So for any call state[8] $\overline{s}$, if there is a number i and a $\tau \in \text{FieldIDs}^*$ such that $\overline{s}|_{\text{LV}_{0,i}\,\tau} = r$, then $(r, \text{LV}_{0,i}\,\tau, \mathit{true})$ should be included in the input arguments of the top stack frame. The only exception are references r that are no *top references* and where all *predecessors* of r can also be reached from some formal parameter $\overline{s}|_{\text{LV}_{0,j}}$ of the called method. The reason is that then r is only reachable from other input arguments of $\overline{s}$ and hence, their flags suffice to indicate whether the object at r has changed. Here, r is a *top reference* iff $\overline{s}|_\pi = r$ holds for some position π with $|\pi| = 1$ (i.e., π has the form $\text{LV}_{i,j}$, $\text{OS}_{i,j}$, or $\text{IN}_{i,\tau}$). A reference r' is a *predecessor* of r iff $\overline{s}|_\pi = r'$ and $\overline{s}|_{\pi\,v} = r$ for some $\pi \in \text{SPos}(\overline{s})$ and some $v \in \text{FieldIDs}$.

For P in Fig. 2, o_4, i_{10}, and o_5 are at positions of the form $\text{LV}_{0,i}\,\tau$. However, only o_4 and

[7] When methods modify objects as a side effect, the exact result of this modification is often not expressible if objects are abstracted to integers. Therefore tools like Julia and COSTA often do not try to express such modifications and fail if this would have been crucial for the termination proof. Indeed, for `cappE`'s termination, one needs information about the object `a` *after* it was modified by `a.appE(j)`. Therefore, while Julia and COSTA can prove termination of `appE`, they fail on `cappE` (although in this example, the effect of the modification would even be expressible when using the path-length abstraction to integers).

[8] In fact, this requirement also has to be imposed for initial states of method graphs, i.e., states with just one stack frame and program position `0` (i.e., at the start of a method).

i_{10} must be input arguments (o_5 is not at a top position and its only predecessor is o_4).

Finally, we can explain how to construct termination graphs in general:

- Each call state $(\langle \overline{fr}_0, \ldots, \overline{fr}_m \rangle, \overline{h})$ is connected to $(\langle \overline{fr}_0 \rangle, \overline{h})$ by a *call edge*.
- Each return state $s = (\langle fr \rangle, h)$ has an edge to the *program end* (ε, h) and *context concretization edges* to all context concretizations of s with call states of the termination graph.
- For all other states s, if $s \xrightarrow{SyEv} s'$, then we connect s to s' by an *evaluation edge*.
- If evaluation is impossible, we use integer or instance refinement (using *refinement edges*).
- To get finite graphs,[9] we use a heuristic which sometimes introduces more general states (e.g., when a program position is visited twice). If $s' \sqsubseteq s$, then s' can be connected to s by an *instance edge*. However, all cycles of the graph must contain an evaluation edge.
- In a termination graph, all nodes except *program ends* must have outgoing edges.

In [5, Thm. 10] we proved that on *concrete* states, our notion of symbolic evaluation $\xrightarrow{SyEv}$ is equivalent to evaluation in JBC. Thm. 2.3 shows that symbolic evaluation of *abstract* states correctly simulates the evaluation of concrete states (and hence, of JBC).

▶ **Theorem 2.3** (Soundness of Termination Graphs). *Let c, c' be concrete states where c can be evaluated to c' (i.e., $c \xrightarrow{SyEv} c'$). If a termination graph contains an abstract state s which represents c (i.e., $c \sqsubseteq s$), then the graph has a path from s to a state s' with $c' \sqsubseteq s'$.*

Paths in the termination graph that correspond to repeated evaluations of concrete states are called *computation paths*. Note that Thm. 2.3 can be used to prove the soundness of our approach: Suppose there is an infinite JBC-computation, i.e., an infinite evaluation of concrete states $c_1 \xrightarrow{SyEv} c_2 \xrightarrow{SyEv} \ldots$ If c_1 is represented in the termination graph, then by Thm. 2.3 there is an infinite computation path in the termination graph. In Thm. 3.3, we will show that then the TRS resulting from the termination graph is not terminating.

3 From Modular Termination Graphs to Term Rewriting

We now transform termination graphs into *integer term rewrite system* (*ITRSs*) [9]. These are conditional TRSs where the booleans, integers, standard arithmetic operations $\mathcal{A}rith\mathcal{O}p$ like $+, -, *, /, \ldots$, and standard relations $\mathcal{R}el\mathcal{O}p$ like $>, <, \ldots$ are pre-defined by an infinite set of rules $\mathcal{PD}$. For example, $\mathcal{PD}$ contains $4 + 2 \to 6$ and $2 < 3 \to \mathsf{true}$. The *rewrite relation* $\hookrightarrow_{\mathcal{R}}$ of an ITRS $\mathcal{R}$ is defined as the *innermost* rewrite relation of $\mathcal{R} \cup \mathcal{PD}$, where all variables (including extra variables in conditions or right-hand sides of rules) may only be instantiated by normal forms. So if $\mathcal{R}$ contains "$\mathsf{f}(x) \to \mathsf{g}(x, y) \mid x > 2$", then $\mathsf{f}(4 + 2) \hookrightarrow_{\mathcal{R}} \mathsf{f}(6) \hookrightarrow_{\mathcal{R}} \mathsf{g}(6, 23)$. TRS termination techniques can easily be adapted to ITRSs as well [9].

As in [15, Def. 3.2], a reference r in a state s with heap h is transformed into a term by the function $\mathrm{tr}(s, r)$. If $h(r) \in$ Unknown or $h(r)$ is an integer interval of several numbers, then $\mathrm{tr}(s, r)$ is a variable with the name r. If $h(r)$ is a concrete integer like $[5, 5]$, then $\mathrm{tr}(s, r)$ is the corresponding constant 5. If $h(r) = \mathtt{null}$, then $\mathrm{tr}(s, r)$ is the constant null.

The main advantage of our rewrite-based approach becomes obvious when transforming data objects into terms (i.e., when $h(r) \in$ Instances). The reason is that such data objects essentially *are* terms and hence, our transformation can keep their structure. We use the class names as function symbols, and the arguments of these symbols represent the values of

[9] Indeed, our implementation uses heuristics which guarantee that we automatically generate a finite termination graph for any JBC program.

fields. So to represent objects of the class `List`, we use a unary function symbol List whose argument corresponds to the value of the field `n`. Thus, o_1 in P from Fig. 2 is transformed into the term $\mathrm{tr}(P, o_1) = \mathsf{List}(\mathsf{List}(o_5))$.[10] However, references r pointing to *cyclic* objects are transformed to a variable r in order to represent an "arbitrary unknown" object.

Now we show how to transform states into terms. In [5, 15], for each state s we used a function symbol f_s which had one argument for each top position in the state. In contrast, to model the call and return of methods, we now encode each stack frame on its own. Then a state is represented by nesting the terms for its stack frames.

To encode a stack frame (in, pp, lv, os) of s to a term, we use a function symbol $f_{s,pp}$ whose arguments correspond to the top positions in this frame. To represent the call stack, $f_{s,pp}$ gets an additional first argument, which contains the encoding of the frame *above* the current one, or eos (for "end of stack") if there is no such frame. So the top stack frame is always at an innermost position of the form $1\,1\ldots1$. Thus, state P is encoded as the term

$$\mathrm{ts}(P) \;=\; \mathsf{f}_{P,34}(\,\mathsf{f}_{P,0}(\mathsf{eos},\, \underbrace{\mathsf{List}(o_5)}_{o_4},\, i_{10},\, \underbrace{\mathsf{List}(o_5)}_{o_4},\, i_{10}),\quad \underbrace{\mathsf{List}(\mathsf{List}(o_5))}_{o_1},\, i_9,\, \underbrace{\mathsf{List}(\mathsf{List}(o_5))}_{o_1},\, i_{10})$$

In Def. 3.1, for any sequence $\langle r_1, ..., r_k\rangle$, "$\mathrm{tr}(s, \langle r_1, ..., r_k\rangle)$" stands for "$\mathrm{tr}(s, r_1), \ldots, \mathrm{tr}(s, r_k)$".

▶ **Definition 3.1** (Transforming States)**.** Let $s = (\langle fr_0, \ldots, fr_n\rangle, h)$ with $fr_i = (in_i, pp_i, lv_i, os_i)$ and $in_i = \{(r_{i,0}, \tau_{i,0}, b_{i,0}), \ldots, (r_{i,k_i}, \tau_{i,k_i}, b_{i,k_i})\}$, for all i. We define $\mathrm{ts}(s) = \overline{\mathrm{ts}}(s, n)$, where

$$\overline{\mathrm{ts}}(s,i) = \begin{cases} \mathsf{f}_{s,pp_i}\left(\,\overline{\mathrm{ts}}(s, i-1),\ \mathrm{tr}(s, \langle r_{i,0}\ldots r_{i,k_i}\rangle),\ \mathrm{tr}(s, lv_i),\ \mathrm{tr}(s, os_i)\,\right), & \text{if } i \geq 0 \\ \mathsf{eos}, & \text{otherwise} \end{cases}$$

As in [15], the instance relation on states is related to the matching relation on the corresponding terms. If $s' \sqsubseteq s$ and the call stack of s has size n, then $\mathrm{ts}(s)$ matches the subterm of $\mathrm{ts}(s')$ that encodes the upper n frames of the call stack. Hence, if one generates rewrite rules to evaluate $\mathrm{ts}(s)$, then they can also be applied to $\mathrm{ts}(s')$. Here, one of course has to label the function symbols in $\mathrm{ts}(s)$ and $\mathrm{ts}(s')$ in the same way. To this end, let $\mathrm{ts}_s(s')$ be a copy of $\mathrm{ts}(s')$ where all symbols are labeled by s instead of s'. Consider Fig. 2, where $P \sqsubseteq A$ and where the call stacks of P and A have size 2 and 1, respectively. Here, $\mathrm{ts}(A) = \mathsf{f}_{A,0}(\mathsf{eos}, \mathsf{List}(o_2), i_3, \mathsf{List}(o_2), i_3)$ matches $\mathrm{ts}_A(P)|_1 = \mathsf{f}_{A,0}(\mathsf{eos}, \mathsf{List}(o_5), i_{10}, \mathsf{List}(o_5), i_{10})$.

To ease presentation,[11] we assume that frames of the same method refer to the "same" input arguments. More precisely, let $fr = (pp, in, lv, os)$ and $fr' = (pp', in', lv', os')$ be frames with pp and pp' in the same method. If $in = \{(r_1, \tau_1, b_1), \ldots, (r_k, \tau_k, b_k)\}$, then we assume that $in' = \{(r'_1, \tau_1, b'_1), \ldots, (r'_k, \tau_k, b'_k)\}$ for the *same* positions $\tau_1, \ldots, \tau_k$. When encoding fr and fr' to terms t and t' in Def. 3.1, we fix a total order on positions $\tau_1, \ldots, \tau_k$. Then the argument positions that correspond to (r_i, τ_i, b_i) in t and to (r'_i, τ_i, b'_i) in t' are the same.

▶ **Lemma 3.2.** *Let $s' \sqsubseteq s$ and let $i = |s'| - |s|$ be the difference of their call stack sizes. Then there is a substitution σ with $\mathrm{ts}(s)\sigma = \mathrm{ts}_s(s')|_{1^i}$. Here, "$1^i$" means "$1\,1\ldots1$" (i times).*

Now we construct ITRSs whose termination implies termination of the original programs. To this end, we transform the edges of the termination graph into rewrite rules.

[10] In general, tr also takes the class hierarchy into account. To simplify the presentation, we refer to [15, Def. 3.3] for details and use the above representation in the illustrating examples.

[11] Without this assumption, $s' \sqsubseteq s$ would not imply that $\mathrm{ts}(s)$ matches a subterm of $\mathrm{ts}_s(s')$. Instead, one first would have to *expand* $\mathrm{ts}_s(s')$ by the additional input arguments of s that are missing in s'. The remaining construction and Thm. 3.3 are easily adapted accordingly (but it complicates the presentation).

If there is an *evaluation edge* from s to $\tilde{s}$, then we generate the rule $\mathrm{ts}(s) \to \mathrm{ts}(\tilde{s})$ which rewrites any instance of s to the corresponding instance of $\tilde{s}$. As in [15], if this edge is labeled with $o_1 = o_2 \circ o_3$ where $\circ \in \mathcal{A}rith\mathcal{O}p$, then in $\mathrm{ts}(\tilde{s})$ we replace o_1 by $\mathrm{tr}(s, o_2) \circ \mathrm{tr}(s, o_3)$. If the edge is labeled by $o_1 \circ o_2$ where $\circ \in \mathcal{R}el\mathcal{O}p$, then we add the condition $\mathrm{tr}(s, o_1) \circ \mathrm{tr}(s, o_2)$ to the generated rule. So the edge from H to I in Fig. 2 results in

$$\mathsf{f}_{H,8}(\mathsf{eos}, \mathsf{List}(\mathsf{null}), i_7, \mathsf{List}(\mathsf{null}), i_7, i_7) \;\to\; \mathsf{f}_{I,12}(\mathsf{eos}, \mathsf{List}(\mathsf{null}), i_7, \mathsf{List}(\mathsf{null}), i_7) \quad | \quad i_7 > 0$$

If there is an *instance edge* from s to $\tilde{s}$, then in the resulting rule we keep all information that we already have for the specialized state s and continue rewriting with the rules we already created for $\tilde{s}$. So instead of $\mathrm{ts}(s) \to \mathrm{ts}(\tilde{s})$, we generate the rule $\mathrm{ts}(s) \to \mathrm{ts}_{\tilde{s}}(s)$. For example, for the instance edge from L to N, we generate the rule

$$\mathsf{f}_{L,26}(\mathsf{eos}, \mathsf{List}(\mathsf{List}(\mathsf{null})), i_7, \mathsf{List}(\mathsf{List}(\mathsf{null})), i_8) \;\to\; \mathsf{f}_{N,26}(\mathsf{eos}, \mathsf{List}(\mathsf{List}(\mathsf{null})), i_7, \mathsf{List}(\mathsf{List}(\mathsf{null})), i_8)$$

Similarly, if there is a *refinement edge* from s to $\tilde{s}$, then $\tilde{s}$ is a specialized version of s. These edges represent a case analysis and hence, some instances of s are also instances of $\tilde{s}$, but others are no instances of $\tilde{s}$. By Lemma 3.2, we can use pattern matching to perform the necessary case analysis. Thus, instead of $\mathrm{ts}(s) \to \mathrm{ts}(\tilde{s})$ we generate the rule $\mathrm{ts}_s(\tilde{s}) \to \mathrm{ts}(\tilde{s})$. As an example, the instance refinement from B to D results in the rule

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

If there is a *call edge* from s to $\tilde{s}$, then $\tilde{s}$ only contains the top frame of the call stack of s. Here, we also generate the rule $\mathrm{ts}_s(\tilde{s}) \to \mathrm{ts}(\tilde{s})$. So for the edge from P to Q, we get

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

Now this rule and the other appE-rules can be applied in terms like $\mathsf{f}_{P,34}(\underline{\mathsf{f}_{P,0}(\mathsf{eos}, \ldots)}, \ldots)$ to rewrite the underlined subterm that represents a recursive call of appE. By applying all rules corresponding to the edges from Q up to P, one then obtains $\mathsf{f}_{P,34}(\mathsf{f}_{P,34}(\mathsf{f}_{P,0}(\mathsf{eos}, \ldots), \ldots), \ldots)$. So the rules resulting from a termination graph can create call stacks of arbitrary size.

For a *context concretization edge* from s to $\tilde{s}$ with the call-state $\overline{s}$, the left-hand side of the corresponding rule should essentially represent the state where the method in the top frame of $\overline{s}$ has been called and its execution reached the `return` statement in s. So the left-hand side should be like $\mathrm{ts}(\overline{s})$, but the subterm at position $\pi = 1^{|\overline{s}|-1}$ (which encodes the top stack frame of $\overline{s}$) is replaced by $\mathrm{ts}(s)$. Hence, we obtain $\mathrm{ts}(\overline{s})[\mathrm{ts}(s)]_\pi$. Note that in the new state $\tilde{s}$, we used the identification substitution σ for the references from s and $\overline{s}$, cf. Def. 2.2. Therefore, in the corresponding rewrite rule, we should use the new names of these references not only on the right-hand side of the rule (which results from encoding $\tilde{s}$), but also on the left-hand side. In other words, we create the rule $(\mathrm{ts}(\overline{s})[\mathrm{ts}(s)]_\pi)\sigma \to \mathrm{ts}(\tilde{s})$.

As an example, let s be the return state V, $\overline{s}$ be the call state P, and $\tilde{s}$ be the context concretization W. We abbreviate "List" by "L". Then for the edge from V to W, we get

$$\begin{array}{l}\mathsf{f}_{P,34}(\,\mathsf{f}_{V,34}(\mathsf{eos}, \mathsf{L}(\mathsf{L}(o_{20}^W)), i_{19}^W, \mathsf{L}(\mathsf{L}(o_{20}^W)), i_{21}^W),\;\; \mathsf{L}(\mathsf{L}(o_5^W)),\quad i_9^W,\; \mathsf{L}(\mathsf{L}(o_5^W)),\quad i_{19}^W) \;\to \\ \mathsf{f}_{W,34}(\,\mathsf{f}_{W,34}(\mathsf{eos}, \mathsf{L}(\mathsf{L}(o_{20}^W)), i_{19}^W, \mathsf{L}(\mathsf{L}(o_{20}^W)), i_{21}^W),\; \mathsf{L}(\mathsf{L}(\mathsf{L}(o_{20}^W))),\; i_9^W,\; \mathsf{L}(\mathsf{L}(\mathsf{L}(o_{20}^W))),\; i_{19}^W)\end{array}$$

Note that on the left-hand side of this rule, for the lower stack frames of P, we still have the values *before* the execution of the method (then, o_1 had the value $\mathsf{L}(\mathsf{L}(o_5))$ in P). The reason is that when simulating the evaluation of states via term rewriting, our rules only modify the subterm corresponding to the top stack frame, until the method of the top frame

reaches a `return`. At that point, we perform all side effects that were caused by the executed method and modify the objects in lower stack frames accordingly. Therefore, the above rule performs the side effect of changing the object at o_1 from $\mathsf{L}(\mathsf{L}(o_5))$ to $\mathsf{L}(\mathsf{L}(\mathsf{L}(o_{20})))$.[12]

As explained in [15], to simplify the resulting TRS, one can often *merge* rules (where essentially, a rule $\ell \to r \mid b$ is used to narrow all right-hand sides where it is applicable and afterwards, the rule is removed). In this way, the termination graph for `appE` of Fig. 2 is transformed into the following ITRS. The rules correspond to the paths from state A via D and F to G (rule (1)), from A via D, H, and P back to A (rule (2)), from A via C and P back to A (rule (3)), from G to V (rule (4)), and from V via W back to V (rule (5)). To ease readability, we omitted "eos" and the arguments for local variables and operand stack entries from the rules. Moreover, we abbreviated "null" by "n".

$$\mathsf{f}_{A,0}(\mathsf{L}(\mathsf{n}),\, i_6) \to \mathsf{f}_{G,11}(\mathsf{L}(\mathsf{n}),\, i_6) \qquad |i_6 \le 0 \quad (1)$$

$$\mathsf{f}_{A,0}(\mathsf{L}(\mathsf{n}),\, i_7) \to \mathsf{f}_{P,34}(\,\mathsf{f}_{A,0}(\mathsf{L}(\mathsf{n}), i_7 - 1),\, \mathsf{L}(\mathsf{L}(\mathsf{n})),\, i_7) \;|i_7 > 0 \quad (2)$$

$$\mathsf{f}_{A,0}(\mathsf{L}(\mathsf{L}(o_5)),\, i_3) \to \mathsf{f}_{P,34}(\,\mathsf{f}_{A,0}(\mathsf{L}(o_5), i_3),\, \mathsf{L}(\mathsf{L}(o_5)),\, i_3) \quad (3)$$

$$\mathsf{f}_{P,34}(\,\mathsf{f}_{G,11}(\mathsf{L}(\mathsf{n}), i_{12}),\, \mathsf{L}(\mathsf{L}(\mathsf{n})),\, i_9) \to \mathsf{f}_{V,34}(\mathsf{L}(\mathsf{L}(\mathsf{n})),\, i_9) \quad (4)$$

$$\mathsf{f}_{P,34}(\,\mathsf{f}_{V,34}(\mathsf{L}(\mathsf{L}(o_{20})), i_{19}),\, \mathsf{L}(\mathsf{L}(o_5)),\, i_9) \to \mathsf{f}_{V,34}(\mathsf{L}(\mathsf{L}(\mathsf{L}(o_{20}))),\, i_9) \quad (5)$$

These rules are a natural representation of the original JBC algorithm as a TRS. Rules (1) and (2) handle the case where the length of the input list is 1 (i.e., `n == null`). If the integer parameter `i` is `<= 0`, then we immediately return (rule (1)). Otherwise, in rule (2) a new element is attached to the input list (i.e., now the input list is $\mathsf{L}(\mathsf{L}(\mathsf{n}))$, and the algorithm is called recursively with the tail of the list (i.e., again with $\mathsf{L}(\mathsf{n})$) and with `i - 1`. In rule (3), the input list has length ≥ 2. Here, the algorithm is called recursively with the tail of the list, whereas the integer parameter is unchanged. Rules (4) and (5) state that after the execution of the recursive call `n.appE(i)`, the list that results from this recursive call (e.g., $\mathsf{L}(\mathsf{L}(o_{20}))$ in rule (5)) is written to the field `n` of the current list as a side effect (e.g., in rule (5), the subterm $\mathsf{L}(o_5)$ in the current list $\mathsf{L}(\underline{\mathsf{L}(o_5)})$ is replaced by $\mathsf{L}(\mathsf{L}(o_{20}))$).

Termination of this ITRS can easily be proved automatically. In the only recursive rules (2) and (3), either the number in the second argument or the length of the list in the first argument of $\mathsf{f}_{A,0}$ decreases. As mentioned before, termination of `appE` can also be proved by Julia and COSTA, because here it suffices to compare arguments by their path-length. However, if lists or other data objects have to be compared in a different way, tools like Julia and COSTA fail, whereas rewrite techniques can compare arbitrary forms of terms, cf. Sect. 4.

Note that in [5, 15], JBC was transformed into TRSs where defined symbols (except pre-defined operations on integers and booleans) only occur on root positions. So instead of a term like $\mathsf{f}_{P,34}(\underline{\mathsf{f}_{A,0}(\mathsf{L}(\mathsf{n}), i_7 - 1)},\, \mathsf{L}(\mathsf{L}(\mathsf{n})),\, i_7)$ on the right-hand side of rule (2), we would generate a term $\mathsf{f}_{PA}(\mathsf{L}(\mathsf{n}), i_7 - 1,\, \mathsf{L}(\mathsf{L}(\mathsf{n})),\, i_7)$ for a new symbol f_{PA}. The disadvantage is that then it is not possible to re-use TRSs and their termination proofs for auxiliary methods that are called in the current method (i.e., one cannot prove termination in a *modular* way).

So for `cappE` from Sect. 2.3, with our new approach the rule for the call of `appE` is $\mathsf{f}_{A',14}(\ldots) \to \mathsf{f}_{B',17}(\underline{\mathsf{f}_{A,0}(\mathsf{L}(\mathsf{n}), i_1)}, \ldots)$ and the rule for its return is $\mathsf{f}_{B',17}(\underline{\mathsf{f}_{V,34}(\mathsf{L}(\mathsf{L}(o_6)), i_3)}, \ldots)$

[12] So for objects that were changed during the execution of the method, the information from $\tilde{s}$ may not be used on the left-hand side of the resulting rewrite rule. However, one could improve the generation of the left-hand-sides by allowing to use the information from $\tilde{s}$ for those references which were not changed by the method (i.e., where the information in $\tilde{s}$ results from the intersection of the corresponding information in s and $\overline{s}$). Then for the edge from G to R, one would obtain a rule where instead of the left-hand side $\mathsf{f}_{P,34}(\mathsf{f}_{G,11}(\ldots), \mathsf{L}(\mathsf{L}(o_2^R)), i_9^R, \mathsf{L}(\mathsf{L}(o_2^R)), i_{12}^R)$ one has the left-hand side $\mathsf{f}_{P,34}(\mathsf{f}_{G,11}(\ldots), \mathsf{L}(\mathsf{L}(\mathsf{null})), i_9^R, \mathsf{L}(\mathsf{L}(\mathsf{null})), i_{12}^R)$. We used this improvement in rule (4) above.

$\to \mathsf{f}_{C',17}(\mathsf{f}_{C',34}(\ldots),\ldots)$. The rules for $\mathsf{f}_{A,0}$ and the other function symbols from `appE` remain unchanged and can be re-used. Hence, their (innermost) termination proof can also be re-used. Since the remaining rules for `cappE` have no recursion, termination of the `cappE`-TRS trivially follows from termination of the `appE`-TRS. This illustrates the advantages of our modular approach which leads to TRSs that form *hierarchical combinations*. Hence, one can benefit from termination methods like the *dependency pair* technique that prove innermost termination of hierarchical combinations in a modular way, cf. [11, 12, 13]. Note that while COSTA and Julia can prove termination of `appE`, they fail on `cappE`.

Using Lemma 3.2, we can now prove that every computation path in a termination graph can be simulated by a rewrite sequence with the corresponding ITRS.

▶ **Theorem 3.3** (Soundness of ITRS Translation). *If the ITRS corresponding to a termination graph G is terminating, then G has no infinite computation path.*

As explained at the end of Sect. 2.3, by combining Thm. 3.3 with Thm. 2.3, we obtain that termination of the resulting ITRS implies termination of the original JBC program for all concrete states represented in the termination graph. Of course, the converse does not hold, i.e., our approach cannot be used to prove non-termination of JBC. Future work will be concerned with using our termination graphs also for non-termination analysis, as well as for other analyses like absence of null pointer exceptions and side effect freeness.

4 Experiments and Conclusion

We presented a new approach to prove termination of JBC programs automatically. In contrast to our earlier work [5, 15], we introduced a technique (based on *context concretizations*) that abstracts from the exact form of the call stack. In this way, we can now also analyze *recursive* methods, which were excluded in [5, 15]. Moreover, we obtain a *modular* approach, since one can now generate termination graphs for different methods separately and re-use them whenever a method is called. In contrast to [5, 15], we now also synthesize TRSs from the termination graphs whose termination can be proved in a modular way.

We implemented our new approach in the termination tool AProVE [10] and evaluated it on a collection of 83 recursive and 133 non-recursive JBC programs. These examples contain the 172 JBC programs from the *Termination Problem Data Base* (used in the *International Termination Competition*)[13] as well as a number of additional typical recursive programs.[14] Below, we compare AProVE 2011 (which contains all contributions of this paper), AProVE 2010 (which implements [5, 15]),[15] Julia [16], and COSTA [2]. We used a runtime of 2 minutes for each example. "**Y**es" indicates how many examples could be proved, "**F**ail" states how often the tool failed in less than 2 minutes, "**T**" indicates how many examples led to a **T**ime-out, and "**R**" gives the average **R**untime in seconds for each example.

	recursion				no recursion			
	Y	**F**	**T**	**R**	**Y**	**F**	**T**	**R**
AProVE 2011	67	0	16	30	108	0	25	27
AProVE 2010	15	3	65	96	103	13	17	23
Julia	57	26	0	3	96	37	0	2
COSTA	47	35	1	6	73	60	0	5

So due to our new modular approach, AProVE 2011 yields the most precise results for the

[13] We removed one controversial example whose termination depends on the handling of integer overflows.
[14] Of course, we also included `appE` and `cappE`, and AProVE 2011 easily proves termination of them.
[15] In addition, whenever a recursive method is called with *fixed* inputs, AProVE 2010 tries to evaluate it. But it cannot prove termination of recursive method for (infinite) *sets* of possible inputs.

recursive JBC programs in the collection. (However, there are also several examples where Julia or COSTA succeed whereas AProVE fails.) On non-recursive programs, AProVE 2010 was already powerful (but the modularity of our new approach helps in large examples). Of course, Julia and COSTA are significantly faster than AProVE. This is because Julia and COSTA use a *fixed* abstraction from objects to integers, whereas AProVE applies rewrite techniques to generate (potentially different) suitable well-founded orders in every termination proof. Nevertheless, the experiments clearly show that rewrite techniques are not only powerful, but also efficient enough for termination of JBC. So a fruitful approach for the future could be to couple the rewrite-based approach of AProVE with the technique of Julia and COSTA to combine their respective advantages. To experiment with our implementation via a web interface and for details on the experiments, we refer to [1].

Acknowledgement. We are grateful to F. Spoto and S. Genaim for help with the experiments and to the referees for many helpful suggestions.

References

1 `http://aprove.informatik.rwth-aachen.de/eval/JBC-Recursion/`.
2 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.
3 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.
4 M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. Technical Report AIB 2011-02, RWTH Aachen, 2011. Available at [1] and at `http://aib.informatik.rwth-aachen.de`.
5 M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl. Termination graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, LNCS 6463, pages 17–37, 2010. Extended version (with proofs) available at [1].
6 M. Colón and H. Sipma. Practical methods for proving program termination. In *Proc. CAV '02*, LNCS 2404, pages 442–454, 2002.
7 B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. PLDI '06*, pages 415–426. ACM Press, 2006.
8 B. Cook, A. Podelski, and A. Rybalchenko. Summarization for termination: No return! *Formal Methods in System Design*, 35(3):369–387, 2009.
9 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.
10 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.
11 J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(1):21–58, 2002.
12 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
13 N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1,2):172–199, 2005.
14 T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Prentice Hall, 1999.
15 C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proc. RTA '10*, LIPIcs 6, pages 259–276, 2010. Extended version (with proofs) available at [1].
16 F. Spoto, F. Mesnard, and É. Payet. A termination analyser for Java Bytecode based on path-length. *ACM TOPLAS*, 32(3), 2010.