RWTH Aachen

Department of Computer Science
Technical Report

Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting

Marc Brockschmidt, Carsten Otto, and Jürgen Giesl

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2011-02

RWTH Aachen · Department of Computer Science · April 2011

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

M. Brockschmidt, C. Otto, J. Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

Abstract

In [4, 14] 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.

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, 5, 6, 7]). However, these methods do not re-use the many existing termination techniques for TRSs and declarative languages. Therefore, in [4, 14] we presented the first rewriting-based approach for proving termination of a real imperative object-oriented language, viz. Java Bytecode [13].

We only know of two other automated methods to analyze JBC termination, implemented in the tools COSTA [2] and Julia [15]. 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 [4, 14] 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 that rewrite techniques can automatically generate well-founded orders comparing arbitrary forms of terms. Moreover, by using TRSs with built-in integers [8], 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

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

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 [4, 14], termination of this TRS implies termination of the original JBC program.

We implemented this approach in our tool AProVE [9] 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, 7, 15]), our approach in [4, 14] 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 [4, 14] 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 [4, 14], 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 [4, 14]:

(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 [4, 14], now we generate TRSs that are amenable to modular termination proofs.

See the appendix for all proofs, and [1] for experimental details and our previous papers [4, 14] (including proofs).

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 [4, 14] 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.

2.1 States

[1] See `http://www.termination-portal.org/wiki/Termination_Competition`.
[2] The bytecode verifier of the JVM [13] 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.

```
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 [4, 14] 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, "$o_2 : \texttt{List}(?)$" 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 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 $0 \leq i \leq n$, $lv_i = \langle l_0, \ldots, l_m \rangle$, $0 \leq j \leq m$. Then $s|_\pi$ is l_j.
- $\pi = \text{OS}_{i,j}$ where $0 \leq i \leq n$, $os_i = \langle o_0, \ldots, o_k \rangle$, $0 \leq j \leq k$. Then $s|_\pi$ is o_j.
- $\pi = \text{IN}_{i,\tau}$ where $0 \leq i \leq n$ 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 $\texttt{List}(?)$ of o_2 does not provide enough information to evaluate `ifnonnull`. Thus, we perform an *instance refinement* [4, 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 $\texttt{List}(\texttt{n} = o_5)$. 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., "$o_1 : \texttt{List}(\texttt{n} = o_2)$" 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* [4, Def. 1], leading to states F (if `i <= 0`) and H.

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

[3] We rename references that are refined to ease the formal definition of the refinements, cf. [4].

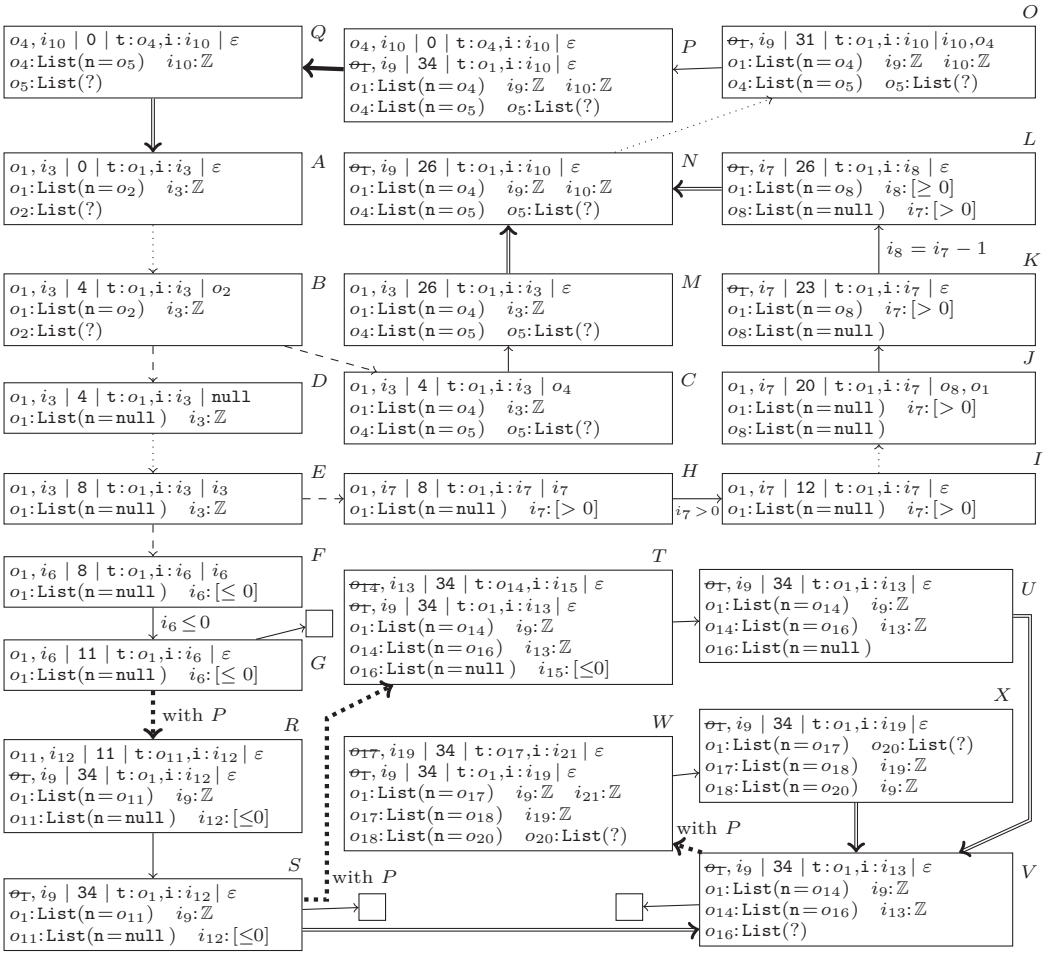


Figure 2 Termination Graph of appE

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. 3 on the soundness of termination graphs later on, in Def. 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 $\mathit{Ref}(s)$ and $\mathit{Ref}(\overline{s})$ 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., $o_1^G \in \mathit{Ref}(G)$ and $o_1^P \in \mathit{Ref}(P)$).

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 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.

---

[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.

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 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** (Context Concretization). Let $s = (\langle \mathit{fr}_0, \ldots, \mathit{fr}_n \rangle, h)$ and let $\overline{s} = (\langle \overline{\mathit{fr}}_0, \ldots, \overline{\mathit{fr}}_m \rangle, \overline{h})$ be a call state where fr_n and $\overline{\mathit{fr}}_0$ correspond to the same method. (So $\overline{\mathit{fr}}_0$ is at the start of the method and fr_n can be at any position of the method.) Let in_n resp. $\overline{\mathit{in}}_0$ be the input arguments of fr_n resp. $\overline{\mathit{fr}}_0$, and let $\mathit{Ref}(s) \cap \mathit{Ref}(\overline{s}) = \varnothing$. For every input argument $(\overline{r}, \tau, \overline{b}) \in \overline{\mathit{in}}_0$ there must be a *corresponding* input argument $(r, \tau, b) \in \mathit{in}_n$ (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 $r \in Ref(s)$, $\overline{r} \in Ref(\overline{s})$, $r \equiv \overline{r}$, 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 $\rho : \textsc{Refs}/_{\equiv} \to \textsc{Refs}$ be an injective mapping to fresh references $\notin Ref(s) \cup Ref(\overline{s})$ and let $\sigma(r) = \rho([r]_{\equiv})$ for all $r \in Ref(s) \cup Ref(\overline{s})$. 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 $[r]_{\equiv} \cap Ref(s) = \{r_1, \ldots, r_k\}$, $[r]_{\equiv} \cap Ref(\overline{s}) = \{\overline{r}_1, \ldots, \overline{r}_d\}$, and there is no input argument $(r_i, \tau, \mathit{false}) \in in_n$
- $h(r_1) \cap \ldots \cap h(r_k)$, if $[r]_{\equiv} \cap Ref(s) = \{r_1, \ldots, r_k\}$, 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 $r_i \in Ref(s_i)$ 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 $h_i(r_i) = (a_i, b_i) \in \textsc{Integers}$ and $\max(a_1, \ldots, a_k) + 1 < \min(b_1, \ldots, b_k)$
- `null`, if all $h_i(r_i) \in \textsc{Unknown} \cup \{\texttt{null}\}$ and at least one of them is `null`
- `null`, if all $h_i(r_i) \in \textsc{Unknown}$ 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 $k = 2$, $s_1 \neq s_2$, $h_1(r_1) = (cl, f_1), h_2(r_2) = (cl, f_2) \in \textsc{Instances}$. 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 $h_1(r_1) \in \textsc{Instances}$ and $h_2(r_2) \in \textsc{Unknown}$ 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 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).

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

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.

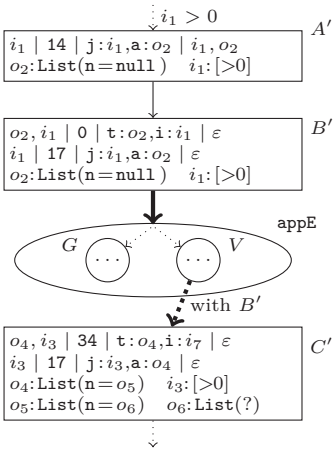
In general, a state s' is an *instance* of a state s (denoted $s' \sqsubseteq s$) if all concrete states represented by s' are also represented by s. For a formal definition of "$\sqsubseteq$", we refer to [4, Def. 3] and [14, Def. 2.3]. The only condition that has to be added to this definition is that 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 [4, 14], $s' \sqsubseteq s$ 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 [4, 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 [4, 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 "`j`:i_1, `a`:o_2, `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` $\mid$ `t`:o_1, `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 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:

---

[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).

- 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 $s \overset{SyEv}{\longrightarrow} s'$, 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 [4, Thm. 10] we proved that on *concrete* states, our notion of symbolic evaluation $\overset{SyEv}{\longrightarrow}$ is equivalent to evaluation in JBC. Thm. 3 shows that symbolic evaluation of *abstract* states correctly simulates the evaluation of concrete states (and hence, of JBC).

▶ **Theorem 3** (Soundness of Termination Graphs). *Let c, c' be concrete states where c can be evaluated to c' (i.e., $c \overset{SyEv}{\longrightarrow} c'$). 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. 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 $c_1 \overset{SyEv}{\longrightarrow} c_2 \overset{SyEv}{\longrightarrow} \ldots$ If c_1 is represented in the termination graph, then by Thm. 3 there is an infinite computation path in the termination graph. In Thm. 6, 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*) [8]. 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 $\mathsf{4} + \mathsf{2} \to \mathsf{6}$ and $\mathsf{2} < \mathsf{3} \to \mathsf{true}$. 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 "$\mathsf{f}(x) \to \mathsf{g}(x, y) \mid x > \mathsf{2}$", then $\mathsf{f}(\mathsf{4} + \mathsf{2}) \hookrightarrow_{\mathcal{R}} \mathsf{f}(\mathsf{6}) \hookrightarrow_{\mathcal{R}} \mathsf{g}(\mathsf{6}, \mathsf{23})$. TRS termination techniques can easily be adapted to ITRSs as well [8].

As in [14, 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 $\mathsf{5}$. 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 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

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

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 [4, 14], 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. 4, 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 4** (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 [14], 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. 4, 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 5.** *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 "$1\,1\,\ldots\,1$" (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.

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 [14], if this edge is labeled

[10] In general, tr also takes the class hierarchy into account. To simplify the presentation, we refer to [14, 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. 6 are easily adapted accordingly (but it complicates the presentation).

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 5, 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. 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 [14], 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".

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

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

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

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

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

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 [4, 14], 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 $\mathsf{f}_{B',17}(\underline{\mathsf{f}_{V,34}(\mathsf{L}(\mathsf{L}(o_6)), i_3)}, \ldots) \to \mathsf{f}_{C',17}(\mathsf{f}_{C',34}(\ldots), \ldots)$. The rules for $\mathsf{f}_{A,0}$ and the other function symbols from `appE` remain

[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.

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. [10, 11, 12]. Note that while COSTA and Julia can prove termination of `appE`, they fail on `cappE`.

Using Lemma 5, we can now prove that every computation path in a termination graph can be simulated by a rewrite sequence with the corresponding ITRS.

▶ **Theorem 6** (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. 6 with Thm. 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 [4, 14], 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 [4, 14]. 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 [4, 14], 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 [9] 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 [4, 14]),[15] Julia [15], 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, 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].

5 M. Colón and H. Sipma. Practical methods for proving program termination. In *Proc. CAV '02*, LNCS 2404, pages 442–454, 2002.

6 B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. PLDI '06*, pages 415–426. ACM Press, 2006.

7 B. Cook, A. Podelski, and A. Rybalchenko. Summarization for termination: No return! *Formal Methods in System Design*, 35(3):369–387, 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, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(1):21–58, 2002.

11 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

12 N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1,2):172–199, 2005.

13 T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Prentice Hall, 1999.

14 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].

15 F. Spoto, F. Mesnard, and É. Payet. A termination analyser for Java Bytecode based on path-length. *ACM TOPLAS*, 32(3), 2010.

A Proofs for Sect. 2

We first recapitulate the definition of the *instance* relation on states [4, Def. 3], which we extend to input arguments as explained in Sect. 2.

▶ **Definition 7** (Instance). Let $s' = (\langle fr'_0, \ldots, fr'_n\rangle, h')$ and $s = (\langle fr_0, \ldots, fr_m\rangle, h)$. Then s' is an *instance* of s (denoted $s' \sqsubseteq s$) iff $n \geq m$ and there is a state $\tilde{s}$ that can be obtained from s by (repeated) context concretization with call states from the termination graph, where $\tilde{s} = (\langle \tilde{fr}_0, \ldots, \tilde{fr}_n\rangle, \tilde{h})$, $\tilde{fr}_i = (\tilde{in}_i, \tilde{pp}_i, \tilde{lv}_i, \tilde{os}_i)$, and $fr'_i = (in'_i, pp'_i, lv'_i, os'_i)$, such that for all $\pi \neq \pi' \in \text{SPos}(s')$ the following conditions hold:

(a) $\tilde{pp}_i = pp'_i$ for all $0 \leq i \leq n$.
(b) if $(r', \tau', b') \in in'_i$, then $(\tilde{r}, \tau', \tilde{b}) \in \tilde{in}_i$ and if $b' = \mathit{false}$, then $\tilde{b} = \mathit{false}$.
(c) if $s'|_\pi = s'|_{\pi'}$ and $h'(s'|_\pi) \in \text{Instances} \cup \text{Unknown}$, then $\pi, \pi' \in \text{SPos}(\tilde{s})$ and $\tilde{s}|_\pi = \tilde{s}|_{\pi'}$.
(d) if $s'|_\pi \neq s'|_{\pi'}$ and $\pi, \pi' \in \text{SPos}(\tilde{s})$, then $\tilde{s}|_\pi \neq \tilde{s}|_{\pi'}$.
(e) if $h'(s'|_\pi) \in \text{Integers}$ and $\pi \in \text{SPos}(\tilde{s})$, then $h'(s'|_\pi) \subseteq \tilde{h}(\tilde{s}|_\pi) \in \text{Integers}$.
(f) if $h'(s'|_\pi) = \texttt{null}$ and $\pi \in \text{SPos}(\tilde{s})$, then $\tilde{h}(\tilde{s}|_\pi) = \texttt{null}$ or $\tilde{h}(\tilde{s}|_\pi) \in \text{Unknown}$.
(g) if $h'(s'|_\pi) = (cl', ?) \in \text{Unknown}$ and $\pi \in \text{SPos}(\tilde{s})$, then $\tilde{h}(\tilde{s}|_\pi) = (cl, ?) \in \text{Unknown}$ and cl' is cl or a subtype of cl.
(h) if $h'(s'|_\pi) = (cl', f') \in \text{Instances}$ and $\pi \in \text{SPos}(\tilde{s})$, then $\tilde{h}(\tilde{s}|_\pi) = (cl, ?)$ or $\tilde{h}(\tilde{s}|_\pi) = (cl', f) \in \text{Instances}$, where cl' must be cl or a subtype of cl.

To prove the soundness of termination graphs (Thm. 3), we need to show that if there is a concrete state c and an abstract state s in our graph with $c \sqsubseteq s$, then a direct successor of s either represents c or the state obtained by evaluating c. In particular, if s is connected to its successor $\tilde{s}$ by an *instance edge* (i.e., $s \sqsubseteq \tilde{s}$), we need to ensure that $c \sqsubseteq \tilde{s}$ holds. For this, we have to show the transitivity of $\sqsubseteq$. To this end, we first prove that $\sqsubseteq$ is stable under context concretization with the same call state.

▶ **Lemma 8** (Stability of $\sqsubseteq$ under Context Concretization). *Let $s, s' \in \text{States}$ with $|s'| = |s|$, $s' \sqsubseteq s$, and let $\overline{s}$ be a call state. If there is a context concretization $\tilde{s}'$ of s' with $\overline{s}$, then there is also a context concretization $\tilde{s}$ of s with $\overline{s}$, and we have $\tilde{s}' \sqsubseteq \tilde{s}$.*

Proof. Let $s' = (\langle fr'_0, \ldots, fr'_n\rangle, h')$, $s = (\langle fr_0, \ldots, fr_n\rangle, h)$ and $\overline{s} = (\langle \overline{fr}_0, \ldots, \overline{fr}_m\rangle, \overline{h})$.

We first prove that there is indeed a context concretization of s with $\overline{s}$. Since there exists a context concretization of s' with $\overline{s}$, $\overline{fr}_0$ is at position $\texttt{0}$ in some method and fr'_n is at some position in the same method. By Def. 7(a), fr_n is at the same position as fr'_n. Similarly, for each input argument $(\overline{r}, \tau, \overline{b})$ in $\overline{fr}_0$, there is an input argument (r', τ, b') in fr'_n and by Def. 7(b), there is also an input argument (r, τ, b) in fr_n.

To prove that there is a context concretization of s with $\overline{s}$, we now need to show that none of the used intersections are empty. For that, assume that there are $\pi_1, \ldots, \pi_u \in \text{SPos}(s)$ and $\overline{\pi}_1, \ldots, \overline{\pi}_v \in \text{SPos}(\overline{s})$ such that $s|_{\pi_1} \equiv \ldots \equiv s|_{\pi_u} \equiv \overline{s}|_{\overline{\pi}_1} \equiv \ldots \equiv \overline{s}|_{\overline{\pi}_v}$[16] Remember that $s' \sqsubseteq s$ implies $\pi_i \sqsubseteq \text{SPos}(s')$ for all $1 \leq i \leq u$, cf. [14, Lemma 4.1]. Furthermore, by Def. 7(c), (d), and (h), we know that then also $s'|_{\pi_1} \equiv \ldots \equiv s'|_{\pi_u} \equiv \overline{\pi}_1 \equiv \ldots \equiv \overline{\pi}_v$. From Def. 7(b) we also have that if there is no input argument $(s|_{\pi_i}, \tau, \mathit{false})$ of fr_n, then there is no input argument $(s'|_{\pi_i}, \tau, \mathit{false})$ in fr'_n, i.e., if we need to intersect the values of one

[16] We assume that $s|_{\pi_i} \neq s|_{\pi_j}$ for all i, j. If references appear twice, one can always ignore one of the copies.

equivalence class for concretizing s, then we also needed to intersect them for concretizing s'. We now check each of the conditions of Def. 2:

- If $h(s|_{\pi_j}) = V_j \in \textsc{Integers}$, then by Def. 7(e), $h'(s'|_{\pi_j}) = V'_j \in \textsc{Integers}$ with $V'_j \subseteq V_j$. Let $\overline{V}_i = \overline{h}(\overline{s}|_{\overline{\pi}_i})$ and $\overline{V} = \overline{V}_1 \cap \ldots \cap \overline{V}_v$. Then, as $V'_1 \cap \ldots \cap V'_u \cap \overline{V}$ is non-empty, we know that $V_1 \cap \ldots \cap V_u \cap \overline{V}$ is also non-empty.
- If $h(s|_{\pi_j}) = \texttt{null}$ for some j, then by Def. 7(f), $h'(s'|_{\pi_j}) = \texttt{null}$. Thus, as $h'(s'|_{\pi_1}) \cap \ldots \cap h'(s'|_{\pi_u}) \cap \overline{h}(\overline{s}|_{\overline{\pi}_1}) \cap \ldots \cap \overline{h}(\overline{s}|_{\overline{\pi}_v})$ is non-empty (actually, it is $\texttt{null}$), we know that $h'(s'|_{\pi_1}) \ldots h'(s'|_{\pi_u}) \in \textsc{Unknown} \cup \{\texttt{null}\}$ and thus by Def. 7(f) and (g), also $h(s|_{\pi_1}) \ldots h(s|_{\pi_u}) \in \textsc{Unknown} \cup \{\texttt{null}\}$. Then, $h'(s'|_{\pi_1}) \cap \ldots \cap h'(s'|_{\pi_u}) \cap \overline{h}(\overline{s}|_{\overline{\pi}_1}) \cap \ldots \cap \overline{h}(\overline{s}|_{\overline{\pi}_v})$ is also $\texttt{null}$.
- If $\overline{h}(\overline{s}|_{\overline{\pi}_{\overline{j}}}) = \texttt{null}$ for some $\overline{j}$ and $h'(s'|_{\pi_1}) \cap \ldots \cap h'(s'|_{\pi_u}) \cap \overline{h}(\overline{s}|_{\overline{\pi}_1}) \cap \ldots \cap \overline{h}(\overline{s}|_{\overline{\pi}_v})$ is non-empty, then $h'(s'|_{\pi_1}) \ldots h'(s'|_{\pi_u}) \in \textsc{Unknown} \cup \{\texttt{null}\}$ and $h(s|_{\pi_1}) \ldots h(s|_{\pi_u}) \in \textsc{Unknown} \cup \{\texttt{null}\}$ as before. Then also $h'(s'|_{\pi_1}) \cap \ldots \cap h'(s'|_{\pi_u}) \cap \overline{h}(\overline{s}|_{\overline{\pi}_1}) \cap \ldots \cap \overline{h}(\overline{s}|_{\overline{\pi}_v}) = \texttt{null}$.
- If $h(s|_{\pi_1}) \ldots h(s|_{\pi_u}), \overline{h}(\overline{s}|_{\overline{\pi}_1}) \ldots \overline{h}(\overline{s}|_{\overline{\pi}_v}) \in \textsc{Unknown}$, then the intersection is always non-empty (it at least contains $\texttt{null}$).
- If $h(s|_{\pi_j}) = (cl, f) \in \textsc{Instances}$ for some j, then by Def. 7(h) also $h'(s'|_{\pi_j}) = (cl, f') \in \textsc{Instances}$. As the intersection $h'(s'|_{\pi_1}) \cap \ldots \cap h'(s'|_{\pi_u}) \cap \overline{h}(\overline{s}|_{\overline{\pi}_1}) \cap \ldots \cap \overline{h}(\overline{s}|_{\overline{\pi}_v})$ is non-empty, we know that $u = 1$ and $v = 1$ and that $\overline{h}(\overline{s}|_{\overline{\pi}_1}) = (\overline{cl}, ?) \in \textsc{Unknown}$ or $\overline{h}(\overline{s}|_{\overline{\pi}_1}) = (cl, \overline{f}) \in \textsc{Instances}$.
  Assume that $\overline{h}(\overline{s}|_{\overline{\pi}_1}) = (\overline{cl}, ?) \in \textsc{Unknown}$. Then, as cl and $\overline{cl}$ were not orthogonal in the concretization of s', we know that $h(s|_{\pi_1}) \cap \overline{h}(\overline{s}|_{\overline{\pi}_1}) = (\min(cl, \overline{cl}), \tilde{f})$.
  Otherwise, assume $\overline{h}(\overline{s}|_{\overline{\pi}_1}) = (cl, \overline{f}) \in \textsc{Instances}$. Then, the intersection is trivially non-empty.
- If $h(s|_{\pi_j}) = (cl, ?) \in \textsc{Unknown}$ for all j and there is a $\overline{j}$ with $\overline{h}(\overline{s}|_{\overline{\pi}_{\overline{j}}}) = (\overline{cl}, \overline{f}) \in \textsc{Instances}$, then we can conclude, as above, that $u = 1$ and $v = 1$ and that $h(s|_{\pi_j}) \neq \texttt{null}$. Furthermore, $h'(s'|_{\pi_1}) \in \textsc{Unknown} \cup \textsc{Instances}$ by Def. 7(g) and (h).
  Assume that $h'(s'|_{\pi_1}) = (cl', ?) \in \textsc{Unknown}$. Then, as cl' is a subtype of cl, we also know that $\overline{cl}$ is a subtype of cl.[17] Thus, the intersection is non-empty.
  Otherwise, assume $h'(s'|_{\pi_1}) = (\overline{cl}, f')$. Then $\overline{cl}$ is obviously a subtype of cl and the intersection is non-empty.

We now need to show that $\tilde{s}' \sqsubseteq \tilde{s}$ holds. For that, we check each of the conditions of Def. 7. From $|s'| = |s|$, we conclude $|\tilde{s}'| = |\tilde{s}|$. Let $\tilde{s}' = (\langle \tilde{fr}'_0, \ldots, \tilde{fr}'_k \rangle, \tilde{h}')$, $\tilde{s} = (\langle \tilde{fr}_0, \ldots, \tilde{fr}_k \rangle, \tilde{h})$, $\tilde{fr}'_i = (\tilde{in}'_i, \tilde{pp}'_i, \tilde{lv}'_i, \tilde{os}'_i)$, $\tilde{fr}_i = (\tilde{in}_i, \tilde{pp}_i, \tilde{lv}_i, \tilde{os}_i)$ and $\overline{fr}_i = (\overline{in}_i, \overline{pp}_i, \overline{lv}_i, \overline{os}_i)$. Let $\pi \neq \pi' \in \textsc{SPos}(\tilde{s}')$:

(a) $\tilde{pp}'_i = pp'_i = pp_i = \tilde{pp}_i$ for all $0 \leq i \leq n$ and $\tilde{pp}'_i = \overline{pp}_{i-n} = \tilde{pp}_i$ for all $n < i < n + m$.

(b) If $(\tilde{r}', \tau, \tilde{b}') \in \tilde{in}'_i$ for some $0 \leq i \leq n$, then $(r', \tau, b') \in in'_i$ and by $s' \sqsubseteq s$, there is a $(r, \tau, b) \in in_i$ and consequently also a $(\tilde{r}, \tau, \tilde{b}) \in \tilde{in}_i$. If $\tilde{b}' = b' = \mathit{false}$, then also by $s' \sqsubseteq s$, $b = \tilde{b} = \mathit{false}$.
  If $(\tilde{r}', \tau, \tilde{b}') \in \tilde{in}'_i$ for some $n < i < n + m$, then there is also $(\tilde{r}, \tau, \tilde{b}) \in \tilde{in}_i$, as both are derived from $(\overline{r}', \tau, \overline{b}') \in \overline{in}_{i-n}$. If $\overline{b}' \neq \tilde{b}' = \mathit{false}$, then $\overline{r}'$ reaches some input argument $(\overline{r}, \tau, \overline{b})$ in $\overline{h}$ that corresponds to an input argument $(r', \tau, \mathit{false}) \in in'_n$. As $s' \sqsubseteq s$, there is a similar input argument $(r, \tau, \mathit{false}) \in in_n$ and thus $\tilde{b}$ was also set to false when performing the context concretization of s with $\overline{s}$.

[17] Remember that Java does not allow multiple inheritance and we do not consider interfaces in this paper.

(c) If $\tilde{s}'|_\pi = \tilde{s}'|_{\pi'}$, $\tilde{h}'(\tilde{s}'|_\pi) \in \text{INSTANCES} \cup \text{UNKNOWN}$ and $\pi, \pi' \in \text{SPOS}(s')$, then either $s'|_\pi = s'|_{\pi'}$ and by $s' \sqsubseteq s$ also $s|_\pi = s|_{\pi'}$, thus $\tilde{s}|_\pi = \tilde{s}|_{\pi'}$ or $\tilde{h}'(\tilde{s}'|_\pi) = h'(s'|_\pi) \cap \overline{h}(r)$ and $\tilde{h}'(\tilde{s}'|_{\pi'}) = h'(s'|_{\pi'}) \cap \overline{h}(r')$ with $s'|_\pi \equiv s'|_{\pi'}$. But then, also $s|_\pi \equiv s|_{\pi'}$ and thus $\tilde{s}|_\pi = \tilde{s}|_{\pi'}$.
If $\tilde{s}'|_\pi = \tilde{s}'|_{\pi'}$, $\tilde{h}'(\tilde{s}'|_\pi) \in \text{INSTANCES} \cup \text{UNKNOWN}$ and $\pi, \pi' \notin \text{SPOS}(s')$, then the references $\tilde{s}'|_\pi, \tilde{s}'|_{\pi'}$ were created from references at the corresponding positions $\overline{\pi}, \overline{\pi}'$ in $\overline{s}$ and we had $\overline{s}|_{\overline{\pi}} \equiv \overline{s}|_{\overline{\pi}'}$. This means that we either had $\overline{s}|_{\overline{\pi}} = \overline{s}|_{\overline{\pi}'}$ or there were $\tau, \tau' \in \text{SPOS}(s')$ such that $s'|_\tau \equiv \overline{s}|_{\overline{\pi}}$ and $s'|_{\tau'} \equiv \overline{s}|_{\overline{\pi}'}$. As the positions π, π' exist in $\tilde{s}$, we also have $\tau, \tau' \in \text{SPOS}(s)$ and $s|_\tau \equiv s|_{\tau'}$. Consequently, we choose the same fresh identifiers for both references in the concretization and thus $\tilde{s}|_\pi = \tilde{s}|_{\pi'}$.

(d) Analogously to (c).

(e) If $\tilde{h}'(\tilde{s}'|_\pi) = \tilde{V}' \in \text{INTEGERS}$ and $\pi \in \text{SPOS}(\tilde{s})$, then $\tilde{h}(\tilde{s}|_\pi) = \tilde{V}$. If $\pi \in \text{SPOS}(s')$, then $\tilde{V}' = h'(s'|_\pi) \cap M$ for some M (where $M = \mathbb{Z}$ if there was no intersection involved). Then $\tilde{V} = h(s|_\pi) \cap M$ for that same M and as $h'(s'|_\pi) \subseteq h(s|_\pi)$, we have $\tilde{V}' \subseteq \tilde{V}$.
If $\pi \notin \text{SPOS}(s')$, then $\tilde{V}' = \overline{h}(\overline{s}|_{\overline{\pi}}) \cap M$ where $M = \mathbb{Z}$ or $M = h'(s'|_\tau)$ for some $\tau \in \text{SPOS}(s')$. If $\tau \in \text{SPOS}(s)$, then $h'(s'|_\tau) \subseteq h(s|_\tau)$ and thus, as $\tilde{V} = \overline{h}(\overline{s}|_{\overline{\pi}}) \cap M$, we have $\tilde{V}' \subseteq \tilde{V}$. If $\tau \notin \text{SPOS}(s)$, then $\tilde{V} = \overline{h}(\overline{s}|_{\overline{\pi}})$ and obviously, $\tilde{V}' \subseteq \tilde{V}$.

(f),(g),(h) Analogously to (e).

◀

Using Lemma 8, transitivity of $\sqsubseteq$ can now easily proved by reducing it to the case of states with call stacks of the same size. (For this case, we proved transitivity of $\sqsubseteq$ already in [4, Lemma 13].)

▶ **Lemma 9** ($\sqsubseteq$ transitive). *Let $s, s', s'' \in \text{STATES}$ with $s'' \sqsubseteq s'$ and $s' \sqsubseteq s$. Then $s'' \sqsubseteq s$.*

Proof. We have $|s''| \geq |s'| \geq |s|$. From $s'' \sqsubseteq s'$, we can conclude that there is a state $\tilde{s}'$ that can be obtained by repeated context concretization of s' such that $|s''| = |\tilde{s}'|$ and $s'' \sqsubseteq \tilde{s}'$. Let $\tilde{s}$ be the state resulting from s by performing the same context concretizations. Thus, $|\tilde{s}'| \geq |\tilde{s}|$ and by Lemma 8, we have $\tilde{s}' \sqsubseteq \tilde{s}$.

Hence, by further repeated context concretization of $\tilde{s}$, we can obtain a state $\hat{s}$ with $|\tilde{s}'| = |\hat{s}|$ and $\tilde{s}' \sqsubseteq \hat{s}$. Hence, we now have $|s''| = |\tilde{s}'| = |\hat{s}|$ and $s'' \sqsubseteq \tilde{s}' \sqsubseteq \hat{s}$. Thus, [4, Lemma 13] implies $s'' \sqsubseteq \hat{s}$. Since $\hat{s}$ was obtained by repeated context concretization from s, this also implies $s'' \sqsubseteq s$. ◀

After having shown the soundness of instance edges, we now prove the soundness of *context concretization edges*.

▶ **Lemma 10** (Soundness of Context Concretization Edges). *Let $c \sqsubseteq s$ for a return state s and $c \xrightarrow{SyEv} c'$. Then there exists a context concretization $\tilde{s}$ of s with a call state $\overline{s}$ such that $c \sqsubseteq \tilde{s}$.*

Proof. As $c \xrightarrow{SyEv} c'$, c cannot be a program end. As the top stack frames of c and s are at the same program position (i.e., at a `return` instruction), we obtain $|c| \geq 2$ and thus, $|c| > |s|$. Hence, according to Def. 7, there exists a state $\hat{s}$ obtained by repeated context concretization from s such that $|c| = |\hat{s}|$ and $c \sqsubseteq \hat{s}$. Since $|c| > |s|$, we must perform at least one context concretization step from s to $\hat{s}$. Let $\tilde{s}$ be the result of performing the first of these context concretizations on s. Then by Def. 7, we also have $c \sqsubseteq \tilde{s}$. ◀

Next we now prove the soundness of *refinement edges*. Let c be a concrete state (i.e., a state where the heap maps all integer references to singleton intervals and no object reference is mapped to UNKNOWN). If $c \sqsubseteq s$ and s has refinement edges to $s_1, \ldots, s_n$ (i.e., we say that

$\{s_1, \ldots, s_n\}$ *is a refinement of* s), then there is an s_i with $c \sqsubseteq s_i$. We call such refinements *valid*. The following two lemmas adapt our proofs for the validity of the integer and the instance refinement from [4] to the setting of the current paper.

▶ **Lemma 11.** *The integer refinement is valid.*

Proof. Let c be a concrete state, $c \sqsubseteq s$, and let $\{s_1, \ldots, s_n\}$ be an integer refinement of s. As $c \sqsubseteq s$, there is a state $\tilde{s}$ obtained by context concretization steps with $\overline{s}_1, \ldots, \overline{s}_k$ from s such that $|c| = |\tilde{s}|$ and $c \sqsubseteq \tilde{s}$. Let h_c, h, $\tilde{h}$, $\overline{h}_1, \ldots, \overline{h}_k$ be the heaps of c, s, $\tilde{s}$, $\overline{s}_1, \ldots, \overline{s}_k$. We want to prove that there is an $s_i \in \{s_1, \ldots, s_n\}$ such that $c \sqsubseteq s_i$.

Let $r \in \textsc{Refs}$ be the reference on which the refinement was performed and let $r_1, \ldots, r_n$ be the fresh references used in $s_1, \ldots, s_n$ to replace r in s. Let $\Pi = \{\pi \in \textsc{SPos}(\tilde{s}) \mid \tilde{h}(\tilde{s}|_\pi) = h(r) \cap \overline{h}_1(\overline{r}_{1,1}) \cap \ldots \cap \overline{h}_1(\overline{r}_{1,d_1}) \cap \ldots \cap \overline{h}_k(\overline{r}_{k,1}) \cap \ldots \cap \overline{h}_k(\overline{r}_{k,d_k})\}$ be the set of positions in $\tilde{s}$ at which an integer is stored that was created by intersecting $h(r)$ in the concretization process.[18]

By [14, Lemma 4.1], $c \sqsubseteq \tilde{s}$ implies $\Pi \subseteq \textsc{SPos}(c)$. By construction, $\tilde{s}|_\pi = \tilde{s}|_{\pi'}$ for all $\pi, \pi' \in \Pi$. Then, by Def. 7(d), we also have $c|_\pi = c|_{\pi'}$ and thus we can now choose s_i by taking *the* singleton integer interval $h_c(c|_\pi)$ from c at one of these positions and by choosing the state $s_i \in \{s_1, \ldots, s_n\}$ whose heap h_i satisfies $h_c(c|_\pi) \subseteq h_i(r_i)$. As $h_c(c|_\pi)$ is a singleton interval and the values $h_1(r_1), \ldots, h_n(r_n)$ are a partition of $h(r)$, such an s_i exists. Then, we can define $\tilde{s}_i$ as the result of applying context concretization of s_i with the same call states $\overline{s}_1, \ldots, \overline{s}_k$. This is still possible, because the refinement from s to s_i did not change program positions and it also did not change the number of input arguments or the positions used in input arguments. Furthermore, all intersections are non-empty: Let $\tilde{h}_i$ be the heap of $\tilde{s}_i$. We only consider the position π from above, as all other values have not changed. In $\tilde{s}$, we had $\tilde{h}(\tilde{s}|_\pi) = h(r) \cap \overline{h}_1(\overline{r}_{1,1}) \cap \ldots \cap \overline{h}_1(\overline{r}_{1,d_1}) \cap \ldots \cap \overline{h}_k(\overline{r}_{k,1}) \cap \ldots \cap \overline{h}_k(\overline{r}_{k,d_k})$ and $h_c(c|_\pi) \subseteq \tilde{h}(\tilde{s}|_\pi)$. Thus, $h_c(c|_\pi)$ is contained in all sets $\overline{h}_j(\overline{r}_{j,\ell})$ in this intersection. By the choice of s_i, we have $h_c(c|_\pi) \subseteq h_i(r_i)$ and thus, the intersection $\tilde{h}_i(\tilde{s}_i|_\pi) = h_i(r_i) \cap \overline{h}_1(\overline{r}_{1,1}) \cap \ldots \cap \overline{h}_1(\overline{r}_{1,d_1}) \cap \ldots \cap \overline{h}_k(\overline{r}_{k,1}) \cap \ldots \cap \overline{h}_k(\overline{r}_{k,d_k})$ is also non-empty, as it at least contains $h_c(c|_\pi)$. Therefore, Def. 7(e) is also satisfied. This implies $c \sqsubseteq \tilde{s}_i \sqsubseteq s_i$. ◀

▶ **Lemma 12.** *The instance refinement is valid.*

Proof. Let c be a concrete state, $c \sqsubseteq s$, and let $\{s_0, \ldots, s_n\}$ be an instance refinement of s. As $c \sqsubseteq s$, there is a state $\tilde{s}$ obtained by context concretization steps from s such that $|c| = |\tilde{s}|$ and $c \sqsubseteq \tilde{s}$. Let h_c, h, $\tilde{h}$, $h_0, \ldots, h_n$ be the heaps of c, s, $\tilde{s}$, $s_0, \ldots, s_n$. We want to prove that there is an $s_i \in \{s_0, \ldots, s_n\}$ such that $c \sqsubseteq s_i$.

Let $r \in \textsc{Refs}$ be the reference on which the refinement was performed and let $r_1, \ldots, r_n$ be the fresh references used in $s_1, \ldots, s_n$ to replace r in s. So $h(r) = (cl, ?)$ and $s_1, \ldots, s_n$ result from replacing r by r_i where $h_0(r_0) = \texttt{null}$ and for $1 \leq i \leq n$, $h_i(r_i) = (cl_i, f_i)$ where the cl_i are all (not necessarily proper) subtypes of cl and f_i assigns unknown values to all fields of cl_i and its superclasses.

Analogously to Lemma 11, we can construct a set Π of positions that are affected by the refinement. Here, we do not always intersect the values of s and all call states, since the input arguments may have the flag *false*. However, this does not affect the structure of the set Π. Again, we can conclude that $c|_\pi = c|_{\pi'}$ for all $\pi, \pi' \in \Pi$.

[18] Remember that integers are immutable in the sense that each change to them creates a fresh reference (i.e., integer input arguments always have the flag *true*). Thus, when performing context concretization, integers are always intersected.

In the first case, we assume $h_c(c|_\pi) = \texttt{null}$. We show that then we have $c \sqsubseteq s_0$. As $\tilde{h}(\tilde{s}|_\pi)$ represents $h_c(c|_\pi)$ (i.e., $c \sqsubseteq \tilde{s}$ implies that $\tilde{h}(\tilde{s}|_\pi)$ is `null` or in UNKNOWN by Def. 2), was `null` or UNKNOWN, we know that by construction all values used in the (repeated) intersections in the construction of $\tilde{h}(\tilde{s}|_\pi)$ are also either `null` or UNKNOWN. Let $\tilde{s}_0$ result from s_0 by applying the same context concretization steps that were used to create $\tilde{s}$ from s. This context concretization exists, since neither program positions nor the names or the positions of input arguments were changed in the refinement from s to s_0. Let $\tilde{h}_0$ be the heap of $\tilde{s}_0$. As intersecting `null` with UNKNOWN or `null` always results in `null`, the intersection is non-empty and we have $\tilde{h}_0(\tilde{s}_0|_\pi) = \texttt{null}$. Accordingly, $c \sqsubseteq \tilde{s}_0 \sqsubseteq s_0$.

In the second case, we assume $h_c(c|_\pi) = (cl_c, f_c)$. By Def. 7(h), cl_c is a (not necessarily proper) subtype of cl. Hence, there exists an s_i with $h_i(r_i) = (cl_i, f_i)$ and $cl_i = cl_c$. As $\tilde{h}(\tilde{s}|_\pi)$ represents $h_c(c|_\pi)$, by Def. 2 we can conclude $\tilde{h}(\tilde{s}|_\pi)$ is $(cl, \tilde{f}) \in$ INSTANCES or $(\tilde{cl}, ?) \in$ UNKNOWN where cl is a subtype of $\tilde{cl}$. Thus, we know that by construction all values used in the intersection when constructing $\tilde{h}(\tilde{s}|_\pi)$ were of the form $(cl, \overline{f}) \in$ INSTANCES or $(\overline{cl}, ?)$ where cl is a subtype of $\overline{cl}$. Let $\tilde{s}_i$ result from s_i by applying the same context concretization steps that were used to create $\tilde{s}$ from s. This context concretization exists, since neither program positions nor the names or the positions of input arguments were changed in the refinement from s to s_i. Let $\tilde{h}_i$ be the heap of $\tilde{s}_i$. As intersecting $h_i(r_i) = (cl, f_i) \in$ INSTANCES always results in INSTANCES of the same type cl again, the intersection is non-empty and we have $\tilde{h}_i(\tilde{s}_i|_\pi) = (cl, \tilde{f}_i)$. Accordingly, $c \sqsubseteq \tilde{s}_i \sqsubseteq s_i$. ◀

To prove the soundness of *call edges*, we show that if a call state $\overline{s}$ has a call edge to s (i.e., s contains only the top stack frame of $\overline{s}$), then this call stack split can be undone by context concretization of s with the call state $\overline{s}$. Hence, $\overline{s} \sqsubseteq s$ and thus, $c \sqsubseteq \overline{s}$ implies $c \sqsubseteq s$ by the transitivity of $\sqsubseteq$ (Lemma 9).

▶ **Lemma 13** (Soundness of Call Edges). *Let $\overline{s} = (\langle \overline{fr}_0, \ldots, \overline{fr}_n \rangle, \overline{h})$ be a call state and let $s = (\langle \overline{fr}_0 \rangle, \overline{h})$. Then $\overline{s} \sqsubseteq s$.*

Proof. Context concretization of s with $\overline{s}$ results in a state that is identical to $\overline{s}$ (up to renaming of variables). This implies $\overline{s} \sqsubseteq s$ by Def. 7. ◀

Now we have proved the soundness of instance, context concretization, refinement, and call edges. It remains to show the soundness of *evaluation edges*, i.e., we still have to prove that our symbolic evaluation can simulate the evaluation of JBC. In other words, we have to show that whenever $c \xrightarrow{SyEv} c'$ for concrete states where $c \sqsubseteq s$ and where $s \xrightarrow{SyEv} s'$, then we also have $c' \sqsubseteq s'$. For the non-modular case, we already proved this in [4, Lemma 19]. To extend this to the modular case, one only has to consider references in c which can reach a reference which can also be reached from the top stack frame. Otherwise the reference's value is not influenced by the evaluation step.

Consider a reference r which is both reachable from one of c's stack frames that is not explicitly represented by s and from the top stack frame of c. Then we need to prove that when applying context concretization to s', then this changed value is correctly propagated to the stack frames added by the context concretization. We know that if a reference is reachable from the top frame *and* from one of the lower stack frames, then this reference (or a reference which reaches it) was passed to the upper stack frames through the parameters of method calls. Here, our input arguments play an important role. For every position of these parameters and the references reachable from them where the position starts in a lower stack frame, our requirements on the input arguments of top frames ensure that there is a corresponding input argument. Thus, when an object is modified by the evaluation

of the method in the top frame, then the modified value is used when performing context concretization and this modified value is copied to the corresponding positions in the lower stack frames created by context concretization. Thus, we only have to show for each position of a manipulated object where the position starts in a lower stack frame, that there is a corresponding input argument in the top frame. Then, we know by definition of our context concretization that the modified new value is copied.

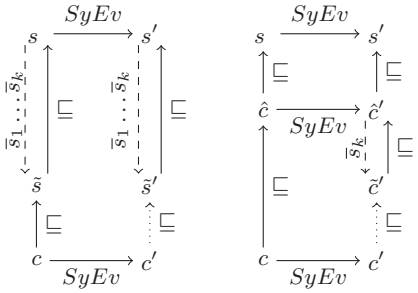


Figure 3 Proof of Lemma 14

The left-hand side of Fig. 3 shows a sketch of the situation that we have when proving the soundness of evaluation edges. By $c \sqsubseteq s$, we know that there is a context concretization $\tilde{s}$ of s with $|\tilde{s}| = |c|$ and $c \sqsubseteq \tilde{s}$. This state $\tilde{s}$ is constructed by performing k context concretizations of s with call states $\overline{s}_1 \dots \overline{s}_k$. We prove the lemma by by induction on the number k of these context concretization steps.

We show that we can perform the same concretization on s' to obtain a state $\tilde{s}'$. Our goal is to show that $c' \sqsubseteq \tilde{s}'$ holds. To this end, we consider the states $\hat{c}$ and $\hat{c}'$. These are constructed from c resp. c', by removing the stack frames corresponding to the last concretization step with call state $\overline{s}_k$, i.e., we simply cut off the bottommost $|\overline{s}_k|$ stack frames. Clearly, we have $\hat{c} \overset{SyEv}{\longrightarrow} \hat{c}'$ and $\hat{c} \sqsubseteq s$ (as we only removed some positions from c). Hence, we can apply the induction hypothesis and conclude that $\hat{c}' \sqsubseteq s'$ holds. We then prove that we can concretize $\hat{c}'$ with $\overline{s}_k$ to obtain $\tilde{c}'$, i.e., that we can re-add those stack frames that were removed in the construction of $\hat{c}'$ from c' by concretization. We then have $|\tilde{c}'| = |c'|$ and prove $c' \sqsubseteq \tilde{c}'$, from which we can conclude from the transitivity of $\sqsubseteq$ that $c' \sqsubseteq s'$. This situation is presented on the right-hand side of Fig. 3.

▶ **Lemma 14** (Soundness of Symbolic Evaluation). *Let c be a concrete state and let $c \sqsubseteq s$. If $c \overset{SyEv}{\longrightarrow} c'$ and $s \overset{SyEv}{\longrightarrow} s'$, then $c' \sqsubseteq s'$.*

Proof. For all instructions not modifying the heap, the claim is obvious, as they only concern the references in local variables or the operand stack. Therefore, we now only consider an evaluation of the `putfield` instruction.

As $c \sqsubseteq s$, there is some $\tilde{s}$ obtained by k concretizations of s with call states $\overline{s}_1, \dots, \overline{s}_k$ such that $|c| = |\tilde{s}|$ and $c \sqsubseteq \tilde{s}$. We prove a slightly stronger statement than just the lemma. More precisely, we show that not only $c' \sqsubseteq s'$, but that we can use the same contexts $\overline{s}_1, \dots, \overline{s}_k$ to concretize s' to obtain a $\tilde{s}'$ with $c' \sqsubseteq \tilde{s}'$. The statement is shown by induction on k. To simplify the presentation, we assume that c and s have the same input arguments, which can easily be achieved by adding missing input arguments to c with values taken from s.

Case $k = 0$:

This was already proved in [4, Lemma 19].

Case $k > 0$:

Assume that the claim holds for $k-1 \geq 0$. Let $c = (\langle fr_0, \dots, fr_n \rangle, h_c)$, $c' = (\langle fr'_0, \dots, fr'_n \rangle, h'_c)$. When repeatedly performing context concretization to transform s to $\tilde{s} = (\langle \tilde{fr}_0, \dots, \tilde{fr}_n \rangle, \tilde{h}_s)$, we obtain states $s = \tilde{s}_0, \tilde{s}_1, \dots, \tilde{s}_{k-1}, \tilde{s}_k = \tilde{s}$ such that $\tilde{s}_{i+1}$ results from context concretization of $\tilde{s}_i$ with $\overline{s}_{i+1}$. Let $\hat{s} = \tilde{s}_{k-1} = (\langle \tilde{fr}_0, \dots, \tilde{fr}_\ell \rangle, \hat{h})$. Now let $\hat{c} = (\langle fr_0, \dots, fr_\ell \rangle, h_c)$ be the "abridged" version of c where we removed the stack frames corresponding to the last context concretization step from $\hat{s}$ to $\tilde{s}$. From $c \sqsubseteq \tilde{s}$, we can conclude that also $\hat{c} \sqsubseteq \hat{s}$. For this, remember that $\text{SPos}(\hat{c}) \subseteq \text{SPos}(c)$. For all positions π from $\text{SPos}(\hat{c}) \cap \text{SPos}(\hat{s})$, $c \sqsubseteq \tilde{s}$ implies that $h_c(s|_\pi)$ is represented by the corresponding value $\tilde{h}_s(\tilde{s}|_\pi)$. However, $\tilde{h}_s(\tilde{s}|_\pi)$

is obtained by either copying the value from $\hat{h}(\hat{s}|_\pi)$ or by intersecting it with other values. Thus, $h_c(s|_\pi)$ is also represented by the $\hat{h}(\hat{s}|_\pi)$ and we have $\hat{c} \sqsubseteq \hat{s}$. As $\hat{s} \sqsubseteq s$ by construction, the transitivity of $\sqsubseteq$ (Lemma 9) implies $\hat{c} \sqsubseteq s$.

We can construct, similar to $\hat{c}$, a state $\hat{c}' = (\langle fr'_0, \ldots, fr'_\ell \rangle, h'_c)$ that corresponds to the "abridged" version of c'. Clearly, $\hat{c} \overset{SyEv}{\longrightarrow} \hat{c}'$ follows from $c \overset{SyEv}{\longrightarrow} c'$. As we only need $k-1$ concretization steps on s to obtain a state $\hat{s}$ with $\hat{c} \sqsubseteq \hat{s}$, we can apply our induction hypothesis and conclude that $\hat{c}' \sqsubseteq s'$ and that we can concretize s' with $\overline{s}_1 \ldots \overline{s}_{k-1}$ to obtain a $\tilde{s}'_{k-1}$ with $\hat{c}' \sqsubseteq \tilde{s}'_{k-1}$. We now prove that one can concretize $\hat{c}'$ with $\overline{s}_k$ to obtain $\tilde{c}'$ and that $c' \sqsubseteq \tilde{c}'$. Then, by transitivity, $c' \sqsubseteq s'$. Furthermore, we can then apply Lemma 8 to conclude that we can also concretize $\tilde{s}'_{k-1}$ (i.e., the result of concretizing s' with $\overline{s}_1 \ldots \overline{s}_{k-1}$, which is possible by our induction hypothesis) with $\overline{s}_k$ to obtain a $\tilde{s}'_k$ with $c' \sqsubseteq \tilde{s}'_k$.

First, we show that that one can perform context concretization of $\hat{c}'$ with $\overline{s}_k$ which yields a state $\tilde{c}'$ with heap $\tilde{h}'$. Let $\hat{h}', \overline{h}_k, \hat{h}$ and $\tilde{h}'_{k-1}$ be the heaps of $\hat{c}', \overline{s}_k, \hat{s}$ and $\tilde{s}'_{k-1}$. As $\overline{s}_k$ could be used to concretize $\tilde{s}_{k-1}$, we know that $\overline{s}_k$'s top stack frame is at the first program position of the method of the bottom stack frame of $\tilde{s}_{k-1}$. As $\hat{c} \sqsubseteq \tilde{s}_{k-1}$, $\hat{c}$'s bottom stack frame is at that same position. Furthermore, the program position of the bottom stack frame of $\hat{c}$ does not change when evaluating it to $\hat{c}'$. Thus, $\overline{s}_k$ is at the right program position to be used in the context concretization of $\hat{c}'$. Furthermore, by our assumption about the input arguments of c and s, its input arguments are also represented by $\hat{c}'$. Now assume that there is an $\equiv$ equivalence class $\{r_1, \ldots, r_u, r_{u+1}, \ldots, r_m\}$ of references such that $r_1 \ldots r_u \in \mathit{Ref}(\hat{c}')$ and $r_{u+1} \ldots r_m \in \mathit{Ref}(\overline{s}_k)$. If $m > 1$ and none of them corresponds to an input argument $(r_i, \tau, \mathit{false})$ of fr'_ℓ, we need to check whether their intersection is empty. However, in this case, their values were not modified in the evaluation. Let $\pi_1, \ldots, \pi_u$ be such that $\hat{c}'|_{\pi_j} = r_j$. Then we also have $\hat{c}|_{\pi_j} = r_j$. As $\hat{c} \sqsubseteq \hat{s}$, we know by Def. 7(c) that the references at positions $\pi_1 \ldots \pi_u$ are also in the same equivalence class when concretizing $\hat{s}$ with $\overline{s}_k$ and none of the elements of this equivalence class was corresponding to a changed input argument. As the intersection of $\hat{s}$ with $\overline{s}_k$ for this equivalence class was non-empty and the resulting value represented $h(c|_{\pi_j})$, we can conclude that the intersection with $h(c|_{\pi_j})$ is also non-empty. Thus, $\tilde{c}'$ exists.

As $\tilde{c}' \sqsubseteq \hat{c}'$, it suffices to prove $c' \sqsubseteq \tilde{c}'$. Note that $|c'| = |\tilde{c}'|$. We now check the conditions of Def. 7 for all $\pi \neq \pi' \in \mathrm{SPos}(c')$:

(a) Given by construction.

(b) Given by construction.

(c) Let $c'|_\pi = c'|_{\pi'}$. We perform the following case analysis.

Case 1: neither π nor π' were affected by the evaluation

Then we also have $c|_\pi = c|_{\pi'}$. Thus, $\hat{c}|_\pi = \hat{c}|_{\pi'}$ and also $\hat{c}'|_\pi = \hat{c}'|_{\pi'}$, and therefore $\tilde{c}'|_\pi = \tilde{c}'|_{\pi'}$.

Case 2: $\pi, \pi' \in \mathrm{SPos}(\hat{c}')$ and no $\tau \in \mathrm{SPos}(c') \setminus \mathrm{SPos}(\hat{c}')$ with $c'|_\tau = c'|_\pi$ or $c'|_\tau = c'|_{\pi'}$ exists

In this case, the considered positions were not influenced by the concretization and thus, we are done by our induction hypothesis.

Case 3: Only $c'|_\pi$ has been affected and it is not in $\mathrm{SPos}(\hat{c}')$

As $c'|_\pi$ is affected, we know that there is some decomposition $\pi = \tau v \mu$ with $c|_\tau = c|_{\mathrm{os}_{0,1}}$ (i.e., the reference whose field v is modified in the evaluation of `putfield`) and τ of minimal length. Whenever a reference from a lower stack frame is passed to a higher stack frame, then this is due to a method call and the corresponding reference is included in the input arguments of the called method. As the reference $c|_\tau$ is used in the top stack frame of c and the position τ begins as local variable, operand stack entry or input variable of a stack frame below fr_ℓ, we have an input argument (r, ρ, b) of fr_ℓ such that there is a

ν with $c'|_{\text{IN}_{\ell,\rho}\,\nu} = c'|_\tau$. Thus, as τ is by choice not modified by the evaluation, we have $c|_\tau = c|_{\text{IN}_{\ell,\rho}\,\nu} = c'|_{\text{IN}_{\ell,\rho}\,\nu}$. But as $\text{IN}_{\ell,\rho} \in \text{SPOS}(\hat{c})$, we also have $\text{IN}_{\ell,\rho} \in \text{SPOS}(\hat{c}')$ and thus, $\hat{c}|_{\text{OS}_{0,1}} = \hat{c}|_{\text{IN}_{\ell,\rho}\,\nu} = \hat{c}'|_{\text{IN}_{\ell,\rho}\,\nu}$. Here, we can append η again and obtain $\hat{c}'|_{\text{IN}_{\ell,\rho}\,\nu\eta} = c'|_\pi$.

Case 3(a): $\pi' \in \text{SPOS}(\hat{c}')$

In this case, we now know that $c|_{\text{OS}_{0,0}\,\mu} = c|_{\pi'}$ implies $\hat{c}'|_{\text{IN}_{\ell,\rho}\,\nu v\mu} = \hat{c}'|_{\pi'}$. Then, we know that $\hat{c}'|_{\text{IN}_{\ell,\rho}\,\nu v\mu} \equiv \hat{c}'|_{\pi'}$ and thus after the concretization, we have $\tilde{c}'|_{\text{IN}_{\ell,\rho}\,\nu\mu} = \tilde{c}'|_{\pi'}$. By choice of $\text{IN}_{\ell,\rho}$, we also have $\tilde{c}'|_{\text{IN}_{\ell,\rho}\,\nu v\mu} = \tilde{c}'|_\pi$.

Case 3(b): $\pi' \notin \text{SPOS}(\hat{c}')$

In this case, we can construct a decomposition $\alpha\beta$ of π' such that $c|_\alpha = c|_{\text{IN}_{\ell,\rho'}}$, as $c|_{\pi'}$ is reachable from the top stack frame. We then also have $c|_{\text{OS}_{0,0}\,\mu} = c|_{\text{IN}_{\ell,\rho'}\,\beta}$ and consequently, $\hat{c}'|_{\text{IN}_{\ell,\rho}\,\nu v\mu} = \hat{c}'|_{\text{IN}_{\ell,\rho'}\,\beta}$. Then, by choice of $\text{IN}_{\ell,\rho}$ and $\text{IN}_{\ell,\rho'}$, we also have $\tilde{c}'|_\pi = \tilde{c}'|_{\text{IN}_{\ell,\rho}\,\nu v\mu} = \tilde{c}'|_{\text{IN}_{\ell,\rho'}\,\beta} = \tilde{c}'|_{\pi'}$.

Case 4: Only $c|_{\pi'}$ has been affected and it is not in $\text{SPOS}(\hat{c}')$

Symmetrical to case 3.

Case 5: Both $c|_\pi, c|_{\pi'}$ have been affected and are not in $\text{SPOS}(\hat{c}')$

As in case 3, we construct $\text{IN}_{\ell,\rho}\,\nu v\mu$ and $\text{IN}_{\ell,\rho'}\,\nu' v\mu'$ such that $c'|_{\text{IN}_{\ell,\rho}\,\nu v\mu} = c'|_\pi$ and $c'|_{\text{IN}_{\ell,\rho'}\,\nu' v\mu'} = c'|_{\pi'}$. Then, we can conclude that we already have $\hat{c}'|_{\text{IN}_{\ell,\rho}\,\nu v\mu} = \hat{c}'|_{\text{IN}_{\ell,\rho'}\,\nu' v\mu'}$ and thus, after concretization, we have $\tilde{c}'|_\pi = \tilde{c}'|_{\text{IN}_{\ell,\rho}\,\nu v\mu} = \tilde{c}'|_{\text{IN}_{\ell,\rho'}\,\nu' v\mu'} = \tilde{c}'|_{\pi'}$.

(d) Analogous to (c).

(e) Trivial for $\pi \in \text{SPOS}(\hat{c}')$. The case that π was not modified by the evaluation is also trivial. Assume $\pi \notin \text{SPOS}(\hat{c}')$ and π was modified by the evaluation. Then, as above, there are ρ, ν, η such that $c'|_\pi = c'|_{\text{IN}_{\ell,\rho}\,\nu\eta} = \hat{c}'|_{\text{IN}_{\ell,\rho}\,\nu\eta}$. By our induction hypothesis, the value at $\hat{c}'|_{\text{IN}_{\ell,\rho}\,\nu\eta}$ is correct, i.e., the same as obtained by concrete evaluation. The boolean flag of the input position $\text{IN}_{\ell,\rho}$ is *false*, as some successor was modified by the evaluation and thus, this correct value of $c'|_{\text{IN}_{\ell,\rho}\,\nu\eta}$ is copied when concretizing, such that $\tilde{h}'(\tilde{c}'|_\pi) = \hat{h}'(\hat{c}'|_{\text{IN}_{\ell,\rho}\,\nu\eta})$. Consequently, $h'(c'|_\pi) \subseteq \tilde{h}'(\tilde{c}'|_\pi)$.

(f),(g),(h) Analogous to (e).

◀

Using these lemmas, we can now prove the soundness of termination graphs, i.e., that every concrete JBC-evaluation corresponds to a computation path in the termination graph.

▶ **Theorem 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 $c' \sqsubseteq s'$.*

Proof. We prove the theorem by induction on the sum of the lengths of all paths from s to the next evaluation edge. This sum is always finite, since we required that every cycle of a termination graph must contains at least one evaluation edge.

We perform a case analysis on the type of the outgoing edges of s. If s has a *call edge* or an *instantiation edge* to $\tilde{s}$, then $s \sqsubseteq \tilde{s}$ by Lemma 13. Hence, we obtain $c \sqsubseteq \tilde{s}$ by transitivity of $\sqsubseteq$ (Lemma 9) and the claim follows from the induction hypothesis.

If the outgoing edges of s are *context concretization* or *refinement edges* to $s_1, \ldots, s_n$, we know by Lemma 10, Lemma 11 and Lemma 12 that there is an s_i with $c \sqsubseteq s_i$. Again, then the claim follows from the induction hypothesis.

Finally, if there is an *evaluation edge* from s to s' (i.e., $s \xrightarrow{SyEv} s'$), we know by Lemma 14 that $c' \sqsubseteq s'$. ◀

B Proofs for Sect. 3

▶ **Lemma 5.** *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 "$1\,1\,\ldots\,1$" (i times).*

Proof. This is essentially a copy of the proof in [14, Lemma 3.5], just changed to the new encoding of stack frames and the addition of input arguments. ◀

For the proof of Thm. 6, we need to show that each evaluation of a concrete state that is represented in the termination graph can be simulated by the rules to generate termination graphs. To this end, we prove two auxiliary lemmas.

▶ **Lemma 15.** *Let $c_0 \overset{SyEv}{\longrightarrow} \ldots \overset{SyEv}{\longrightarrow} c_n$ be an evaluation of concrete states with $n > 0$, let there be a path from a state s_0 to a state s_m in a termination graph G with $c_0 \sqsubseteq s_0$ and $c_n \sqsubseteq s_m$. Let $|c_0| = |c_n|$ and $|c_0| \leq |c_i|$ for all $1 \leq i \leq n$. Then, for the ITRS $\mathcal{R}$ corresponding to the termination graph G, we have $ts_{s_0}(c_0)|_{1^k} \hookrightarrow^+_{\mathcal{R}} ts_{s_m}(c_n)|_{1^k}$, where $k = |c_0| - |s_0|$.*

Proof. We prove the lemma by induction over the number ℓ of call edges on the path from s_0 to s_m.

Case $\ell = 0$: There are no call edges and consequently, no context concretization edges. In this case, we can apply [14, Lemma 4.10] to prove the claim.

Case $\ell > 0$: We assume that the lemma holds for all values smaller that ℓ. Let $s_0, s_1, \ldots, s_m$ be the sequence of states on the path from s_0 to s_m and let i, j such that the edge from s_i to s_{i+1} is the first call edge on this path and the edge from s_j to s_{j+1} is the context concretization edge corresponding to the return from the method called between s_i and s_{i+1}, i.e., where we concretized s_j with the call state s_i to obtain s_{j+1}.

Let $c_{i'}$ and $c_{j'}$ be the states of the concrete evaluation such that $c_{i'} \sqsubseteq s_i$ and $c_{j'} \sqsubseteq s_j$ (thus, also $c_{j'} \sqsubseteq s_{j+1}$). The correspondence between the method call and return implies that we have $|c_{i'}| = |c_{j'}|$.

For the subsequence $s_0, \ldots, s_i$, there is no call edge and we can apply [14, Lemma 4.10], as in the base case, to obtain $\mathrm{ts}_{s_0}(c_0)|_{1^k} \hookrightarrow^*_{\mathcal{R}} \mathrm{ts}_{s_i}(c_{i'})|_{1^k}$. For the call edge from s_i to s_{i+1}, we generate the rule $\mathrm{ts}_{s_i}(s_{i+1}) \to \mathrm{ts}(s_{i+1})$. As s_{i+1}'s call stack is just a copy of s_i's top stack frame, we have $\mathrm{ts}_{s_i}(s_{i+1}) = \mathrm{ts}(s_i)|_{1^a}$ where $a = |s_i| - 1$. From $c_{i'} \sqsubseteq s_i$ we can conclude by Lemma 5 that there is a substitution σ_i such that $\mathrm{ts}(s_i)\sigma_i = \mathrm{ts}_{s_i}(c_{i'})|_{1^k}$ (since $|c_{i'}| - |s_i| = |c_0| - |s_0| = k$). Consequently, we can apply the generated rule to continue the rewrite sequence, i.e.,

$$\begin{array}{ll} & \mathrm{ts}_{s_0}(c_0)|_{1^k} \\ \hookrightarrow^*_{\mathcal{R}} & \mathrm{ts}_{s_i}(c_{i'})|_{1^k} \\ = & \mathrm{ts}(s_i)\sigma_i \\ = & (\mathrm{ts}_{s_i}(c_{i'})[\mathrm{ts}_{s_i}(s_{i+1})\sigma_i]_{1^a})|_{1^k} \\ \hookrightarrow_{\mathcal{R}} & (\mathrm{ts}_{s_i}(c_{i'})[\mathrm{ts}(s_{i+1})\sigma_i]_{1^a})|_{1^k} \end{array}$$

However, as we have $c_{i'} \sqsubseteq s_{i+1}$ by construction, we can use the induction hypothesis for the evaluation $c_{i'} \overset{SyEv}{\longrightarrow} \ldots \overset{SyEv}{\longrightarrow} c_{j'}$, where $c_{i'} \sqsubseteq s_{i+1}$ and $c_{j'} \sqsubseteq s_j$, as we have fewer call edges on this path. Thus, we can rewrite $\mathrm{ts}_{s_{i+1}}(c_{i'})|_{1^{a+k}}$ to $\mathrm{ts}_{s_j}(c_{j'})|_{1^{a+k}}$. Hence, the above rewrite sequence can be continued:

$$\begin{array}{ll} & \mathrm{ts}_{s_0}(c_0)|_{1^k} \\ \hookrightarrow^+_{\mathcal{R}} & (\mathrm{ts}_{s_i}(c_{i'})[\mathrm{ts}(s_{i+1})\sigma_i]_{1^a})|_{1^k} \\ \hookrightarrow_{\mathcal{R}} & (\mathrm{ts}_{s_i}(c_{i'})[\mathrm{ts}_{s_j}(c_{j'})|_{1^{a+k}}]_{1^a})|_{1^k} \end{array}$$

For the context concretization edge from s_j to s_{j+1}, we generate the rule

$$(\mathrm{ts}(s_i)[\mathrm{ts}(s_j)]_{1^a})\rho \to \mathrm{ts}(s_{j+1}), \tag{6}$$

where ρ is the identification substitution from the concretization of s_j with s_i. As we have $c_{j'} \sqsubseteq s_j$, there is some substitution σ_j such that $\mathrm{ts}(s_j)\sigma_j = \mathrm{ts}_{s_j}(c_{j'})|_{1^{a+k}}$ (cf. Lemma 5). We can apply the rule to our term as ρ only merges those identifiers that correspond to the same reference in $c_{j'}$ anyway. Thus, these identifiers need to be instantiated with the same terms by σ_i and σ_j. Thus, the following substitution ρ' is well defined:

$$\rho' = \{\rho(r)/\sigma_i(r) \mid r \in \mathit{Ref}(s_i)\} \cup \{\rho(r)/\sigma_j(r) \mid r \in \mathit{Ref}(s_j)\}$$

The substitution ρ' is a matcher such that $(\mathrm{ts}(s_i)[\mathrm{ts}(s_j)]_{1^a})\rho\rho' = (\mathrm{ts}_{s_i}(c_{i'})[\mathrm{ts}_{s_j}(c_{j'})|_{1^{a+k}}]_{1^a})|_{1^k}$ and thus we can apply rule (6) to rewrite our term to $\mathrm{ts}(s_{j+1})\rho'$. With that, we can rewrite $\mathrm{ts}_{s_0}(c_0)|_{1^k} \hookrightarrow_{\mathcal{R}}^+ \mathrm{ts}_{s_{j+1}}(c_{j'})|_{1^k}$. If there are no further call edges on the path from s_{j+1} to s_m, we can apply the base case to rewrite $ts_{s_{j+1}}(c_{j'})|_{1^k}$ to $ts_{s_m}(c_n)|_{1^k}$ and thus prove the lemma. If there are further call edges, we would need to use the same proof as for the edge following s_i until finally, there is no call edge anymore. ◀

Using this, we can prove Thm. 6 only for terminating cases. For the case that the control flow never returns from a called method to the caller, we need the following lemma.

▶ **Lemma 16.** *Let $c_0 \overset{SyEv}{\longrightarrow} c_1$ be an evaluation step of concrete states such that $|c_0| < |c_1|$ and let there be a corresponding computation path from s_0 to s_1 in a termination graph G with $c_0 \sqsubseteq s_0$ and $c_1 \sqsubseteq s_1$. Then, for the corresponding ITRS $\mathcal{R}$, we have $ts_{s_0}(c_0)|_{1^k} \hookrightarrow_{\mathcal{R}} ts_{s_1}(c_1)|_{1^\ell}$, where $k = |c_0| - |s_0|$ and $\ell = |c_1| - |s_1|$.*

Proof. This is just the first part of the proof of Lemma 15. ◀

Based on these two lemmas, we can now prove our main result:

▶ **Theorem 6** (Soundness of ITRS Translation)**.** *If the ITRS corresponding to a termination graph G is terminating, then G has no infinite computation path.*

Proof. We assume that there is some infinite computation path $s_0^0, s_0^1, \ldots, s_1^0, s_1^1, \ldots$ in G, i.e., that there is an infinite computation sequence $c_0 \overset{SyEv}{\longrightarrow} c_1 \overset{SyEv}{\longrightarrow} \ldots$ of concrete states with $c_i \sqsubseteq s_i^j$ for all i, j. Without loss of generality, we assume that $|c_0| \leq |c_i|$ for all $i > 0$.[19]

We now show how to obtain an infinite rewrite sequence starting with $\mathrm{ts}_{s_0^0}(c_0)|_{1^a}$ from this computation path resp. from the infinite computation sequence $c_0 \overset{SyEv}{\longrightarrow} c_1 \overset{SyEv}{\longrightarrow} \ldots$, where $a = |c_0| - |s_0^0|$.

If we have $|c_0| < |c_n|$ for all $n > 0$, then we know that from c_0 to c_1, there is a method call which never returns. By Lemma 16, we have $ts_{s_0^0}(c_0)|_{1^a} \hookrightarrow_{\mathcal{R}} ts_{s_1^0}(c_1)|_{1^b}$, where $a = |c_0| - |s_0^0|$ and $b = |c_1| - |s_1^0|$. Now the construction is continued for the infinite computation sequence $c_1 \overset{SyEv}{\longrightarrow} c_2 \overset{SyEv}{\longrightarrow} \ldots$

Otherwise, we consider the smallest $n > 0$ with $|c_0| = |c_n|$. Here, we apply Lemma 15 to obtain $ts_{s_0^0}(c_0)|_{1^a} \hookrightarrow_{\mathcal{R}}^+ ts_{s_n^0}(c_n)|_{1^b}$, where $a = |c_0| - |s_0^0|$ and $b = |c_n| - |s_n^0|$. Now the construction is continued for the infinite computation sequence $c_n \overset{SyEv}{\longrightarrow} c_{n+1} \overset{SyEv}{\longrightarrow} \ldots$ ◀

[19] Otherwise, one can simply start at a later point in the the infinite sequence.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three 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`

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

2010-04 René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme

2010-05 Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme

2010-06 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata

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

2010-08 Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting

2010-09 George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs

2010-10 Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut

2010-11 Martin Zimmermann: Parametric LTL Games

2010-12 Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut

2010-13 Ashraf Armoush: Design Patterns for Safety-Critical Embedded Systems

2010-14 Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination

2010-15 Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode

2010-16 Christian Berger: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles

2010-17 Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten

2010-18 Ibrahim Armaç: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit

2010-19 Felix Reidl: Experimental Evaluation of an Independent Set Algorithm

2010-20 Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games

2011-04 Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.