Termination Graphs for Java Bytecode[⋆]

M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

Abstract. To prove termination of Java Bytecode (JBC) automatically, we transform JBC to finite *termination graphs* which represent all possible runs of the program. Afterwards, the graph can be translated into "simple" formalisms like term rewriting and existing tools can be used to prove termination of the resulting term rewrite system (TRS). In this paper we show that termination graphs indeed capture the semantics of JBC correctly. Hence, termination of the TRS resulting from the termination graph implies termination of the original JBC program.

1 Introduction

Termination is an important property of programs. Therefore, techniques to analyze termination automatically have been studied for decades [7, 8, 20]. While most work focused on *term rewrite systems* or *declarative programming languages*, recently there have also been many results on termination of *imperative programs* (e.g., [2, 4, 5]). However, these are "stand-alone" methods which do not allow to re-use the many existing termination techniques and tools for TRSs and declarative languages. Therefore, in [15] we presented the first rewriting-based approach for proving termination of a real imperative object-oriented language, viz. Java Bytecode. Related TRS-based approaches had already proved successful for termination analysis of Haskell and Prolog [10, 16].

JBC [13] is an assembly-like object-oriented language designed as intermediate format for the execution of Java by a Java Virtual Machine (JVM). While there exist several static analysis techniques for JBC, we are only aware of two other automated methods to analyze termination of JBC, implemented in the tools COSTA [1] and Julia [18]. They transform JBC into a constraint logic program by abstracting every object of a dynamic data type to an integer denoting its path-length (i.e., the length of the maximal path of references obtained by following the fields of objects). While this fixed mapping from objects to integers leads to a very efficient analysis, it also restricts the power of these methods.

In contrast, in our approach from [15], we represent data objects not by integers, but by *terms* which express as much information as possible about the data objects. In this way, we can benefit from the fact that rewrite techniques can automatically generate suitable well-founded orders comparing arbitrary forms of terms. Moreover, by using TRSs with built-in integers [9], our approach is not only powerful for algorithms on user-defined data structures, but also for algorithms on pre-defined data types like integers.

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

However, it is not easy to transform JBC to a TRS which is suitable for termination analysis. Therefore, we first transform JBC to so-called *termination graphs* which represent all possible runs of the JBC program. These graphs handle all aspects of the programming language that cannot easily be expressed in term rewriting (e.g., side effects, cyclicity of data objects, object-orientation, etc.). Similar graphs are also used in program optimization techniques [17].

To analyze termination of a set $\mathcal{S}$ of desired initial (concrete) program states, we first represent this set by a suitable *abstract* state. This abstract state is the starting node of the termination graph. Then this state is evaluated symbolically, which leads to its child nodes in the termination graph. This symbolic evaluation is repeated until one reaches states that are *instances* of states that already appeared earlier in the termination graph. So while we perform considerably less abstraction than direct termination tools like [1, 18], we also apply suitable abstract interpretations [6] in order to obtain finite representations for all possible forms of the heap at a certain program position.

Afterwards, a TRS is generated from the termination graph whose termination implies termination of the original JBC program for all initial states $\mathcal{S}$. This TRS can then be handled by existing TRS termination techniques and tools.

We implemented this approach in our tool AProVE [11] and in the *International Termination Competitions*,[1] AProVE achieved competitive results compared to Julia and COSTA. So rewriting techniques can indeed be successfully used for termination analysis of imperative object-oriented languages like Java.

However, [15] only introduced termination graphs informally and did not prove that these graphs really represent the semantics of JBC. In the present paper, we give a formal justification for the concept of termination graphs. Since the semantics of JBC is not formally specified, in this paper we do not focus on full JBC, but on JINJA Bytecode [12].[2] JINJA is a small Java-like programming language with a corresponding bytecode. It exhibits the core features of Java, its semantics is formally specified, and the corresponding correctness proofs were performed in the Isabelle/HOL theorem prover [14]. So in the following, "JBC" always refers to "JINJA Bytecode". We present the following new contributions:

- In Sect. 2, we define termination graphs formally and determine how states in these graphs are evaluated symbolically (Def. 6, 7). To this end, we introduce three kinds of edges in termination graphs ($\xrightarrow{\textsc{Eval}}$, $\xrightarrow{\textsc{Ins}}$, $\xrightarrow{\textsc{Ref}}$). In contrast to [15], we extend these graphs to handle also method calls and exceptions.
- In Sect. 3, we prove that on *concrete* states, our definition of "symbolic evaluation" is equivalent to evaluation in JBC (Thm. 10). As illustrated in Fig. 1, there is a mapping $\textsc{trans}$ from JBC program states to our notion of concrete states. Then, Thm. 10 proves that if a program state j_1 of a JBC program is evaluated to a state j_2 (i.e., $j_1 \xrightarrow{jvm} j_2$), then $\textsc{trans}(j_1)$ is evaluated to $\textsc{trans}(j_2)$ using our definitions of "states" and of "symbolic

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

[2] For the same reason, the correctness proof for the termination technique of [18] also regarded a simplified instruction set similar to JINJA instead of full JBC.

$$\begin{tikzcd}
s_1 \arrow[r, "\textsc{Eval}"] & s_2 \arrow[r, "\textsc{Ins}"] & s_2' \arrow[r, "\textsc{Ref}"] & s_2'' \arrow[r, "\textsc{Eval}"] & s_3 & \cdots & \\
 & & & & & & \Big\}\ \text{Thm. 11} \\
c_1 \arrow[uu, "\sqsubseteq"'] \arrow[r, "SyEv"] & c_2 \arrow[uu, "\sqsubseteq"'] \arrow[uur, "\sqsubseteq"] \arrow[uurr, "\sqsubseteq"] \arrow[rrr, "SyEv"] & & & c_3 \arrow[uu, "\sqsubseteq"'] & \cdots & \\
 & & & & & & \Big\}\ \text{Thm. 10} \\
j_1 \arrow[uu, "\textsc{trans}"'] \arrow[r, "jvm"] & j_2 \arrow[uu, "\textsc{trans}"'] \arrow[rrr, "jvm"] & & & j_3 \arrow[uu, "\textsc{trans}"'] & \cdots &
\end{tikzcd}$$

Fig. 1. Relation between evaluation in JBC and paths in the termination graph

evaluation" from Sect. 2 (i.e., $\textsc{trans}(j_1) \xrightarrow{SyEv} \textsc{trans}(j_2)$).

- In Sect. 4, we prove that our notion of symbolic evaluation for *abstract* states correctly simulates the evaluation of concrete states. More precisely, let c_1 be a concrete state which can be evaluated to the concrete state c_2 (i.e., $c_1 \xrightarrow{SyEv} c_2$). Then Thm. 11 states that if the termination graph contains an abstract state s_1 which represents c_1 (i.e., c_1 is an *instance* of s_1, denoted $c_1 \sqsubseteq s_1$), then there is a path from s_1 to another abstract state s_2 in the termination graph such that s_2 represents c_2 (i.e., $c_2 \sqsubseteq s_2$).

Note that Thm. 10 and 11 imply the "soundness" of termination graphs, cf. Cor. 12: Suppose there is an infinite JBC-computation $j_1 \xrightarrow{jvm} j_2 \xrightarrow{jvm} \ldots$ where j_1 is represented in the termination graph (i.e., there is a state s_1 in the termination graph with $\textsc{trans}(j_1) = c_1 \sqsubseteq s_1$). Then by Thm. 10 there is an infinite symbolic evaluation $c_1 \xrightarrow{SyEv} c_2 \xrightarrow{SyEv} \ldots$, where $\textsc{trans}(j_i) = c_i$ for all i. Hence, Thm. 11 implies that there is an infinite so-called computation path in the termination graph starting with the node s_1. As shown in [15, Thm. 3.7], then the TRS resulting from the termination graph is not terminating.

2 Constructing Termination Graphs

To illustrate termination graphs, we regard the method `create` in Fig. 2. `List` is a data type whose `next` field points to the next list element and we omitted the fields for the values of list elements to ease readability. The constructor `List(n)` creates a new list object with `n` as its tail. The method `create(x)` first ensures that `x` is at least 1. Then it creates a list of length `x`. In the end, the list is made cyclic by letting the `next` field of the last list element point to the start of the list. The method `create` terminates as `x` is decreased until it is 1.

After introducing our notion of *states* in Sect. 2.1, we describe the construction of termination graphs in Sect. 2.2 and explain the JBC program of Fig. 2 in parallel. Sect. 2.3 formally defines symbolic evaluation and termination graphs.

2.1 States

The nodes of the termination graph are *abstract states* which represent *sets* of

```
public class List {
 public List next;

 public List(List n) {
    this.next = n;
 }

 public static
 List create(int x) {

   List last;
   List cur;
   if (x <= 0) {
    return null;
   }
   cur = new List(null);
   last = cur;
   while (x != 1) {
    x--;
    cur = new List(cur);
   }
   last.next = cur;
   return cur;

 }
}
```

```
public static List create(int);
    ...                  // return null for x <= 0
    New      List        // create List object
    Push     null        // load null reference
    Invoke   <init> 2    // call constructor
    Store    "cur"       // store into cur
    Load     "cur"       // load cur to opstack
    Store    "last"      // store into last
hd: Load     "x"         // load x to opstack
    Push     1           // load 1 to opstack
    CmpEq                // compare x and 1
    IfFalse  "bd"        // jump to bd if x != 1
    Load     "last"      // load last to opstack
    Load     "cur"       // load cur to opstack
    Putfield next        // set last.next = cur
    Load     "cur"       // load cur to opstack
    Return               // return cur
bd: Load     "x"         // load x to opstack
    Push     -1          // load -1 to opstack
    IAdd                 // add x and -1
    Store    "x"         // store result in x
    New      List        // create List object
    Load     "cur"       // load cur to opstack
    Invoke   <init> 2    // call constructor
    Store    "cur"       // store into cur
    Goto     "hd"        // jump to loop condition
```

Fig. 2. Java Code and a corresponding JINJA Bytecode for the method `create`

concrete states, using a formalization which is especially suitable for a translation into TRSs. Our approach is restricted to verified sequential JBC programs without recursion. To simplify the presentation in the paper, as in JINJA, we exclude floating point arithmetic, arrays, and static class fields. However, our approach can easily be extended to such constructs and indeed, our implementation also handles such programs. We define the set of all states as

$$\textsc{States} = (\textsc{ProgPos} \times \textsc{LocVar} \times \textsc{OpStack})^* \times \\ (\{\bot\} \cup \textsc{References}) \times \textsc{Heap} \times \textsc{Annotations}\,.$$

$\texttt{CmpEq}\ |\ \texttt{x}:i_1, \texttt{l}:o_1, \texttt{c}:o_1\ |\ i_2, i_1$
$i_1 = [1,\infty)\quad i_2 = [1,1]$
$o_1 = \texttt{List}(\texttt{next} = \texttt{null})$

Fig. 3. Abstract state

Consider the state in Fig. 3. Its first component is the program position (from $\textsc{ProgPos}$). In the examples, we represent it by the next program instruction to be executed (e.g., "`CmpEq`").

The second component are the local variables that have a defined value at the current program position, i.e., $\textsc{LocVar} = \textsc{References}^*$. $\textsc{References}$ are addresses in the heap, where we also have $\texttt{null} \in \textsc{References}$. In our representation, we do not store primitive values directly, but indirectly using references to the heap.

In examples we denote local variables by names instead of numbers. Thus, "$\texttt{x}:i_1, \texttt{l}:o_1, \texttt{c}:o_1$" means that the value of the 0^{th} local variable `x` is a reference i_1 for integers and the 1^{st} and 2^{nd} local variables `l` and `c` both reference the address o_1. So different local variables can point to the same address.

The third component is the operand stack that JBC instructions operate on, i.e., $\textsc{OpStack} = \textsc{References}^*$. The empty operand stack is denoted "ε" and "i_2, i_1" denotes a stack with top element i_2 and bottom element i_1.

In contrast to [15], we allow *several* method calls and a triple from ($\textsc{ProgPos}$

$\times$ LocVar $\times$ OpStack) is just one *frame* of the *call stack*. Thus, an abstract state may contain a sequence of such triples. If a method calls another method, then a new frame is put on top of the call stack. This frame has its own program counter, local variables, and operand stack. Consider the state in Fig. 4, where the `List` constructor was called. Hence, the top frame on the call stack corresponds to the first statement of this constructor method. The lower frame corresponds to the statement `Store "cur"` in the method `create`. It will be executed when the constructor in the top frame has finished.

`Load "this"` $|$ `t`$:o_1,$ `n:null` $|\,\varepsilon$
`Store "cur"` $|$ `x`$:i_1\,|\,\varepsilon$
$i_1 = [1,\infty)$
$o_1 =$ `List(next = null)`

Fig. 4. State with 2 frames

The component from $(\{\bot\} \cup \text{References})$ in the definition of States is used for exceptions and will be explained at the end of Sect. 2.2. Here, $\bot$ means that no exception was thrown (we omit $\bot$ in examples to ease readability).

We write the first three components of a state in the first line and separate them by "|". The fourth component Heap is written in the lines below. It contains information about the values of References. We represent it by a partial function, i.e., Heap = References $\to$ Unknown $\cup$ Integers $\cup$ Instances.

The values in Unknown = Classnames $\times\{?\}$ represent tree-shaped (and thus acyclic) objects where we have no information except the type. Classnames are the names of all classes and interfaces. For example, "$o_3 =$ `List`(?)" means that the object at address o_3 is `null` or of type `List` (or a subtype of `List`).

We represent integers as possibly unbounded intervals, i.e. Integers = $\{\{x \in \mathbb{Z} \mid a \le x \le b\} \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \le b\}$. So $i_1 = [1,\infty)$ means that any positive integer can be at the address i_1. Since current TRS termination tools cannot handle 32-bit `int`-numbers as in real Java, we treat `int` as the infinite set of all integers (this is done in JINJA as well).

To represent Instances (i.e., objects) of some class, we describe the values of their fields, i.e., Instances = Classnames $\times$(FieldIDs $\to$ References). To prevent ambiguities, in general the FieldIDs also contain the respective class names. So "$o_1 =$ `List(next = null)`" means that at the address o_1, there is a `List` object and the value of its field `next` is `null`. For all $(cl, f) \in$ Instances, the function f is defined for all fields of the class cl and all of its superclasses.

All sharing information must be explicitly represented. If an abstract state s contains the non-`null` references o_1, o_2 and does not mention that they could be sharing, then s only represents concrete states where o_1 and the references reachable from o_1 are disjoint from o_2 and the references reachable from o_2.

Sharing or aliasing for *concrete* objects can of course be represented easily, e.g., we could have $o_2 =$ `List`(`next` $= o_1$) which means that o_1 and o_2 do not point to disjoint parts of the heap h (i.e., they *join*). But to represent such concepts for *unknown* objects, we use three kinds of *annotations*. Annotations are only built for references $o \ne$ `null` with $h(o) \notin$ Integers.

Equality annotations like "$o_1 \ =^? \ o_2$" mean that the addresses o_1 and o_2 could be equal. Here the value of at least one of o_1 and o_2 must be Unknown. To represent states where two objects "*may join*", we use *joinability annotations* "$o_1 \searrow\!\!\swarrow o_2$". We say that o' is a *direct successor* of o in a state s (denoted $o \to_s o'$)

iff the object at address o has a field whose value is o'. Then "$o_1 \searrow\!\swarrow o_2$" means that if the value of o_1 is UNKNOWN, then there could be an o with $o_1 \rightarrow_s^+ o$ and $o_2 \rightarrow_s^* o$, i.e., o is a proper successor of o_1 and a (possibly non-proper) successor of o_2. Note that $\searrow\!\swarrow$ is symmetric,[3] so "$o_1 \searrow\!\swarrow o_2$" also means that if o_2 is UNKNOWN, then there could be an o' with $o_1 \rightarrow_s^* o'$ and $o_2 \rightarrow_s^+ o'$. Finally, we use *cyclicity annotations* "$o!$" to denote that the object at address o is not necessarily tree-shaped (so in particular, it could be cyclic).[4]

2.2 Termination Graphs, Refinements, and Instances

To build termination graphs, we begin with an abstract state describing all concrete initial states. In our example, we want to know whether all calls of `create` terminate. So in the corresponding initial abstract state, the value of `x` is not an actual integer, but $(-\infty, \infty)$. After symbolically executing the first JBC instructions, one reaches the instruction "`New List`". This corresponds to state A in Fig. 5 where the value of `x` is from $[1, \infty)$.

We can evaluate "`New List`" without further information about `x` and reach the node B via an *evaluation edge*. Here, a new `List` instance was created at address o_1 in the heap and o_1 was pushed on the operand stack. "`New List`" does not execute the constructor yet, but just allocates the needed memory and sets all fields to default values. Thus, the `next` field of the new object is set to `null`.

"`Push null`" pushes `null` on the operand stack. The elements `null` and o_1 on the stack are the arguments for the constructor `<init> 2` that is invoked, where "`2`" means that the constructor with two parameters (`n` and `this`) is used.

This leads to D, cf. Fig. 4. In the top frame, the local variables `this` (abbreviated `t`) and `n` have the values o_1 and `null`. In the second frame, the arguments that were passed to the constructor were removed from the operand stack.

We did not depict the evaluation of the constructor and continue with state E, where the control flow has returned to `create`. So dotted arrows abbreviate several steps. Our implementation of `<init>` returns the newly created object as its result. Therefore, o_1 has been pushed on the operand stack in E.

Evaluation continues to node F, storing o_1 in the local variables `cur` and `last` (abbreviated `c` and `l`). In F one starts with checking the condition of the `while` loop. To this end, `x` and the number 1 are pushed on the operand stack and the instruction `CmpEq` in state G compares them, cf. Fig. 3.

We cannot directly continue the symbolic evaluation, because the control flow depends on the value of the number i_1 in the variable `x`. So we *refine* the information by an appropriate case analysis. This leads to the states H and J where `x`'s value is from $[1, 1]$ resp. $[2, \infty)$. We call this step *integer refinement* and G is connected to H and J by *refinement edges* (denoted by dashed edges in Fig. 5).

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

[4] It is also possible to use an extended notion of annotations which also include sets of FIELDIDs. Then one can express properties like "o may join o' by using only the field `next`" or "o may only have a non-tree structure if one uses *both* fields `next` and `prev`" (such annotations can be helpful to analyze algorithms on doubly-linked lists).

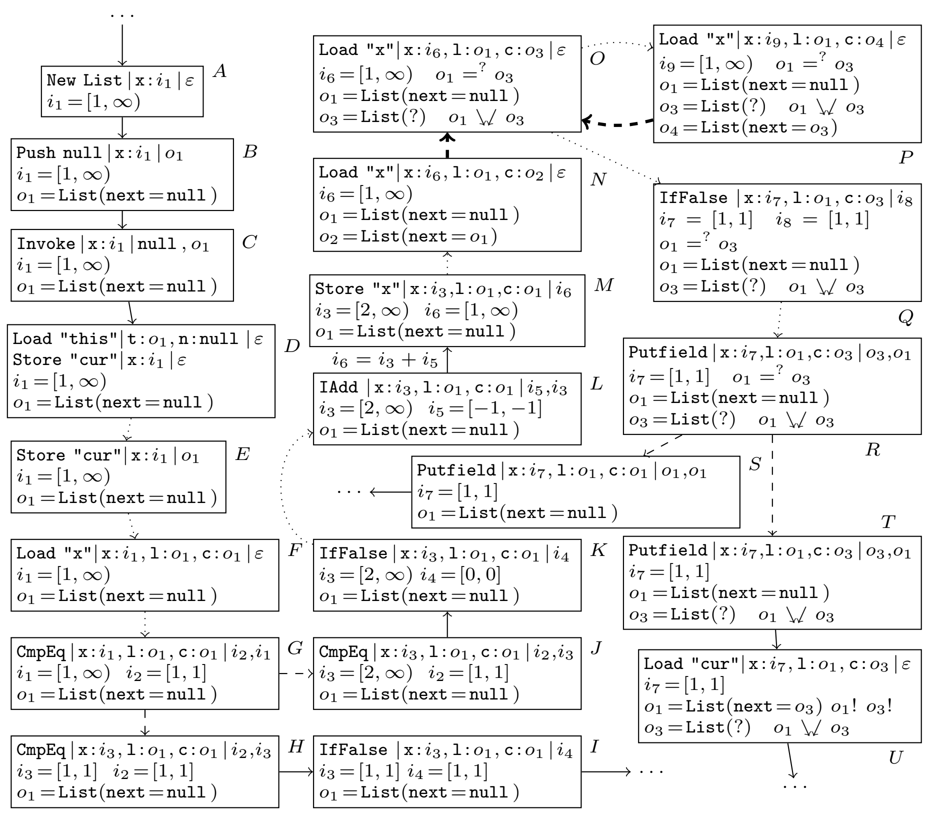

New List | x : i1 | ε i1 = [1, ∞) A
Push null | x : i1 | o1 i1 = [1, ∞) o1 = List(next = null) B
Invoke | x : i1 | null, o1 i1 = [1, ∞) o1 = List(next = null) C
Load "this" | t : o1, n : null | ε Store "cur" | x : i1 | ε i1 = [1, ∞) o1 = List(next = null) D
Store "cur" | x : i1 | o1 i1 = [1, ∞) o1 = List(next = null) E
Load "x" | x : i1, l : o1, c : o1 | ε i1 = [1, ∞) o1 = List(next = null) F
CmpEq | x : i1, l : o1, c : o1 | i2, i1 i1 = [1, ∞) i2 = [1, 1] o1 = List(next = null) G
CmpEq | x : i3, l : o1, c : o1 | i2, i3 i3 = [1, 1] i2 = [1, 1] o1 = List(next = null) H
IfFalse | x : i3, l : o1, c : o1 | i4 i3 = [1, 1] i4 = [1, 1] o1 = List(next = null) I
CmpEq | x : i3, l : o1, c : o1 | i2, i3 i3 = [2, ∞) i2 = [1, 1] o1 = List(next = null) J
IfFalse | x : i3, l : o1, c : o1 | i4 i3 = [2, ∞) i4 = [0, 0] o1 = List(next = null) K
IAdd | x : i3, l : o1, c : o1 | i5, i3 i3 = [2, ∞) i5 = [−1, −1] o1 = List(next = null) L
i6 = i3 + i5
Store "x" | x : i3, l : o1, c : o1 | i6 i3 = [2, ∞) i6 = [1, ∞) o1 = List(next = null) M
Load "x" | x : i6, l : o1, c : o2 | ε i6 = [1, ∞) o1 = List(next = null) o2 = List(next = o1) N
Load "x" | x : i6, l : o1, c : o3 | ε i6 = [1, ∞) o1 =? o3 o1 = List(next = null) o3 = List(?) o1 \/ o3 O
Load "x" | x : i9, l : o1, c : o4 | ε i9 = [1, ∞) o1 =? o3 o1 = List(next = null) o3 = List(?) o1 \/ o3 o4 = List(next = o3) P
IfFalse | x : i7, l : o1, c : o3 | i8 i7 = [1, 1] i8 = [1, 1] o1 =? o3 o1 = List(next = null) o3 = List(?) o1 \/ o3 Q
Putfield | x : i7, l : o1, c : o3 | o3, o1 i7 = [1, 1] o1 =? o3 o1 = List(next = null) o3 = List(?) o1 \/ o3 R
Putfield | x : i7, l : o1, c : o1 | o1, o1 i7 = [1, 1] o1 = List(next = null) S
Putfield | x : i7, l : o1, c : o3 | o3, o1 i7 = [1, 1] o1 = List(next = null) o3 = List(?) o1 \/ o3 T
Load "cur" | x : i7, l : o1, c : o3 | ε i7 = [1, 1] o1 = List(next = o3) o1! o3! o3 = List(?) o1 \/ o3 U

Fig. 5. Termination graph for `create`

To define integer refinements, for any $s \in \textsc{States}$, let $s[o/o']$ be the state obtained from s by replacing all occurrences of the reference o in instance fields, the exception component, local variables, and on the operand stacks by o'. By $s + \{o \mapsto vl\}$ we denote a state which results from s by removing any information about o and instead the heap now maps o to the value vl. So in Fig. 5, J is $(G + \{i_3 \mapsto [2, \infty)\})[i_1/i_3]$. We only keep information on those references in the heap that are reachable from the local variables and the operand stacks.

Definition 1 (Integer refinement). *Let $s \in \textsc{States}$ where h is the heap of s and let $o \in \textsc{References}$ with $h(o) = V \subseteq \mathbb{Z}$. Let $V_1, \ldots, V_n$ be a partition of V (i.e., $V_1 \cup \ldots \cup V_n = V$) with $V_i \in \textsc{Integers}$. Moreover, $s_i = (s + \{o_i \mapsto V_i\})[o/o_i]$ for fresh references o_i. Then $\{s_1, \ldots, s_n\}$ is an* integer refinement *of s.*

In Fig. 5, evaluation of `CmpEq` continues and we push `True` resp. `False` on the operand stack leading to the nodes I and K. To simplify the presentation, in the paper we represent the Booleans `True` and `False` by the integers 1 and 0. In I and K, we can then evaluate the `IfFalse` instruction.

From K on, we continue the evaluation by loading the value of `x` and the constant -1 on the operand stack. In L, `IAdd` adds the two topmost stack elements. To keep track of this, we create a new reference i_6 for the result and label the edge from L to M by the relation between i_6, i_3, and i_5. Such labels are used when constructing rewrite rules from the termination graph [15]. Then, the value of i_6 is stored in `x` and the rest of the loop is executed. Afterwards in state N, `cur` points to a list (at address o_2) where a new element was added in front of the original list at o_1. Then the program jumps back to the instruction `Load "x"` at the label "`hd`" in the program, where the loop condition is evaluated.

However, evaluation had already reached this instruction in state F. So the new state N is a repetition in the control flow. The difference between F and N is that in F, `l` and `c` are the same, while in N, `l` refers to o_1 and `c` refers to o_2, where the list at o_1 is the direct successor (or "tail") of the list at o_2.

To obtain *finite* termination graphs, whenever the evaluation reaches a program position for the second time, we "merge" the two corresponding states (like F and N). This widening result is displayed in node O. Here, the annotation "$o_1 =^? o_3$" allows the equality of the references in `l` and `c`, as in J. But O also contains "$o_1 \searrow\!\!\swarrow o_3$". So `l` may be a successor of `c`, as in N. We connect N to O by an *instance edge* (depicted by a thick dashed line), since the concrete states described by N are a subset of the concrete states described by O. Moreover, we could also connect F to O by an instance edge and discard the states G-N which were only needed to obtain the suitably generalized state O. Note that in this way we maintain the essential invariant of termination graphs, viz. that a node "is terminating" whenever all of its children are terminating.

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

Definition 2 (State positions SPos). *Let $s = (\langle fr_0, \ldots, fr_n \rangle, e, h, a)$ be a state where each stack frame fr_i has the form (pp_i, lv_i, os_i). Then $\textsc{SPos}(s)$ is the smallest set containing all the following sequences π:*

- *$\pi = \textsc{lv}_{i,j}$ where $0 \le i \le n$, $lv_i = o_{i,0}, \ldots, o_{i,m_i}$, $0 \le j \le m_i$. Then $s|_\pi$ is $o_{i,j}$.*
- *$\pi = \textsc{os}_{i,j}$ where $0 \le i \le n$, $os_i = o'_{i,0}, \ldots, o'_{i,k_i}$, $0 \le j \le k_i$. Then $s|_\pi$ is $o'_{i,j}$.*
- *$\pi = \textsc{exc}$ if $e \ne \bot$. Then $s|_\pi$ is e.*
- *$\pi = \pi' v$ for some $v \in \textsc{FieldIDs}$ and some $\pi' \in \textsc{SPos}(s)$ where $h(s|_{\pi'}) = (cl, f) \in \textsc{Instances}$ and where $f(v)$ is defined. Then $s|_\pi$ is $f(v)$.*

For any position π, let $\overline{\pi}_s$ denote the maximal prefix of π such that $\overline{\pi}_s \in \textsc{SPos}(s)$. We write $\overline{\pi}$ if s is clear from the context.

In Fig. 5, $F|_{\textsc{lv}_{0,0}} = i_1, F|_{\textsc{lv}_{0,1}} = F|_{\textsc{lv}_{0,2}} = o_1$. If h is F's heap, then $h(o_1) = (\texttt{List}, f) \in \textsc{Instances}$, where $f(\texttt{next}) = \texttt{null}$. So $F|_{\textsc{lv}_{0,1}\,\texttt{next}} = F|_{\textsc{lv}_{0,2}\,\texttt{next}} = \texttt{null}$.

Intuitively, a state s' is an instance of a state s if they correspond to the same program position and whenever there is a reference $s'|_\pi$, then either the values represented by $s'|_\pi$ in the heap of s' are a subset of the values represented by $s|_\pi$ in the heap of s or else, π is no position in s. Moreover, shared parts of the heap in s' must also be shared in s. Note that since s and s' correspond to the same position in a *verified* JBC program, s and s' have the same number of local variables and their operand stacks have the same size. In Def. 3, the conditions (a)-(d) handle $\textsc{Integers}$, `null`, $\textsc{Unknown}$, and $\textsc{Instances}$, whereas the remaining conditions concern equality and annotations. Here, the conditions (e)-(g) handle the case where two positions π, π' of s' are also in $\mathrm{SP}\textsc{os}(s)$.

Definition 3 (Instance). *Let $s' = (\langle fr'_0, \dots, fr'_n\rangle, e', h', a')$ and $s = (\langle fr_0, \dots, fr_n\rangle, e, h, a)$, where $fr'_i = (pp'_i, lv'_i, os'_i)$ and $fr_i = (pp_i, lv_i, os_i)$. We call s' an* instance *of s (denoted $s' \sqsubseteq s$) iff $pp_i = pp'_i$ for all i and for all $\pi, \pi' \in \mathrm{SP}\textsc{os}(s')$:*

(a) if $h'(s'|_\pi) \in \textsc{Integers}$ and $\pi \in \mathrm{SP}\textsc{os}(s)$, then $h'(s'|_\pi) \subseteq h(s|_\pi) \in \textsc{Integers}$.
(b) if $s'|_\pi = \mathtt{null}$ and $\pi \in \mathrm{SP}\textsc{os}(s)$, then $s|_\pi = \mathtt{null}$ or $h(s|_\pi) \in \textsc{Unknown}$.
(c) if $h'(s'|_\pi) = (cl', ?) \in \textsc{Unknown}$ and $\pi \in \mathrm{SP}\textsc{os}(s)$, then $h(s|_\pi) = (cl, ?) \in \textsc{Unknown}$ and cl' is cl or a subtype of cl.
(d) if $h'(s'|_\pi) = (cl', f') \in \textsc{Instances}$ and $\pi \in \mathrm{SP}\textsc{os}(s)$, then $h(s|_\pi) = (cl, ?)$ or $h(s|_\pi) = (cl', f) \in \textsc{Instances}$, where cl' must be cl or a subtype of cl.
(e) if $s'|_\pi \neq s'|_{\pi'}$ and $\pi, \pi' \in \mathrm{SP}\textsc{os}(s)$, then $s|_\pi \neq s|_{\pi'}$.
(f) if $s'|_\pi = s'|_{\pi'}$ and $\pi, \pi' \in \mathrm{SP}\textsc{os}(s)$ where $h'(s'|_\pi) \in \textsc{Instances} \cup \textsc{Unknown}$, then $s|_\pi = s|_{\pi'}$ or $s|_\pi =^? s|_{\pi'}$.[5]
(g) if $s'|_\pi =^? s'|_{\pi'}$ and $\pi, \pi' \in \mathrm{SP}\textsc{os}(s)$, then $s|_\pi =^? s|_{\pi'}$.
(h) if $\big(s'|_\pi = s'|_{\pi'}$ or $s'|_\pi =^? s'|_{\pi'}$ where $h'(s'|_\pi) \in \textsc{Instances} \cup \textsc{Unknown}\big)$ and $\{\pi, \pi'\} \not\subseteq \mathrm{SP}\textsc{os}(s)$ with $\pi \neq \pi'$, then $s|_{\overline{\pi}} \searrow\!\!\swarrow s|_{\overline{\pi'}}$.
(i) if $s'|_\pi \searrow\!\!\swarrow s'|_{\pi'}$, then $s|_{\overline{\pi}} \searrow\!\!\swarrow s|_{\overline{\pi'}}$.
(j) if $s'|_\pi!$ holds, then $s|_{\overline{\pi}}!$.
(k) if there exist $\rho, \rho' \in \textsc{FieldIDs}^*$ without common prefix where $\rho \neq \rho'$, $s'|_{\pi\rho} = s'|_{\pi\rho'}$, $h'(s'|_{\pi\rho}) \in \textsc{Instances} \cup \textsc{Unknown}$, and ($\{\pi\rho, \pi\rho'\} \not\subseteq \mathrm{SP}\textsc{os}(s)$ or $s|_{\pi\rho} =^? s|_{\pi\rho'}$), then $s|_{\overline{\pi}}!$.

In Fig. 5, we have $F \sqsubseteq O$ and $N \sqsubseteq O$. Symbolic evaluation can continue in the new generalized state O. It again leads to a node like G, where an integer refinement is needed to continue. If the value in `x` is still not 1, eventually one has to evaluate the loop condition again (in node P). Since $P \sqsubseteq O$, we draw an instance edge from P to O and can "close" this part of the termination graph.[6]

If the value in `x` is 1 (which is checked in state Q), we reach state R. Here, the references o_1 and o_3 in `l` and `c` have been loaded on the operand stack and one now has to execute the `Putfield` instruction which sets the `next` field of the object at the address o_1 to o_3. To find out which references are affected by this operation, we need to decide whether $o_1 = o_3$ holds. To this end, we perform an *equality refinement* according to the annotation "$o_1 =^? o_3$".

Definition 4 (Equality refinement). *Let $s \in \textsc{States}$ where h is the heap of s and where s contains "$o =^? o'$". Hence, $h(o) \in \textsc{Unknown}$ or $h(o') \in \textsc{Unknown}$.*

[5] For annotations concerning $s|_\pi$ with $\pi \in \mathrm{SP}\textsc{os}(s)$, we usually do not mention that they are from the $\textsc{Annotations}$ component of s, since s is clear from the context.

[6] If P had not been an instance of O, we would have performed another widening step and created a new node which is more general than O and P. By a suitably aggressive widening strategy, one can ensure that after finitely many widening steps, one always reaches a "fixpoint". Then all states that result from further symbolic evaluation are instances of states that already occurred earlier. In this way, we can automatically generate a finite termination graph for any non-recursive JBC program.

W.l.o.g. let $h(o) \in$ UNKNOWN. *Let* $s_= = s[o/o']$ *and let* $s_{\neq}$ *result from* s *by removing "*$o =^? o'$*". Then* $\{s_=, s_{\neq}\}$ *is an* equality refinement *of* s*.*

In Fig. 5, equality refinement of R results in S (where $o_1 = o_3$) and T (where $o_1 \neq o_3$ and thus, "$o_1 =^? o_3$" was removed). In T's successor U, the `next` field of o_1 has been set to o_3. However, o_1 and o_3 may join due to "$o_1 \searrow\!\!\swarrow o_3$". So in particular, T also represents states where $o_3 \to^+ o_1$. Thus, writing o_3 to a field of o_1 could create a cyclic data object. Therefore, all non-concrete elements in the abstracted object must be annotated with !. Consequently, our symbolic evaluation has to extend our state with "$o_1!$" and "$o_3!$". From U on, the graph construction can be finished directly by evaluating the remaining instructions.

From the termination graph, one could generate the following 1-rule TRS which describes the operations on the cycle of the termination graph.

$$\mathsf{f}_O(i_6, \mathsf{List}(\mathsf{null}), o_3) \ \to\ \mathsf{f}_O(i_6 - 1, \mathsf{List}(\mathsf{null}), \mathsf{List}(o_3)) \ \mid\ i_6 > 0 \wedge i_6 \neq 1 \qquad (1)$$

Here we also took the condition from the states before O into account which ensures that the loop is only executed for numbers `x` that are greater than 0.

As mentioned in Sect. 1, we regard TRSs where the integers and operations like "$-$", "$>$", "$\neq$" are built in [9] and we represent objects by terms. So essentially, for any class `C` with n fields we introduce an n-ary function symbol C whose arguments correspond to the fields of `C`. Hence, the object `List(next = null)` is represented by the term $\mathsf{List}(\mathsf{null})$. A state like O is translated into a term $\mathsf{f}_O(\ldots)$ whose direct subterms correspond to the exception component (if it is not $\perp$), the local variables, and the entries of the operand stack. Hence, Rule (1) describes that in each loop iteration, the value of the 0th local variable decreases from i_6 to $i_6 - 1$, the value of the 1st variable remains $\mathsf{List}(\mathsf{null})$, and the value of the 2nd variable increases from o_3 to $\mathsf{List}(o_3)$. Termination of this TRS is easy to show and indeed, AProVE proves termination of `create` automatically.

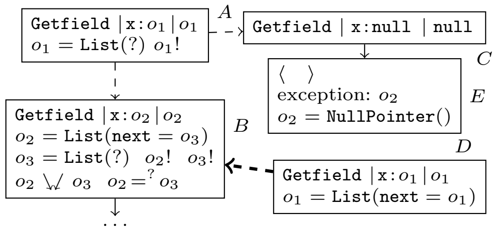


Fig. 6. Instance refinement and exceptions

Finally, we have a third kind of refinement. This *instance refinement* is used if we need information about the existence or the type of an UNKNOWN instance. Consider Fig. 6, where in state A we want to access the `next` field of the `List` object in o_1. However, we cannot evaluate `Getfield`, as the instance in o_1 is UNKNOWN. To refine o_1, we create a successor B where the instance exists and is exactly of type `List` and a state C where o_1 is `null`.

In A the instance may be cyclic, indicated by $o_1!$. For this reason, the instance refinement has to add appropriate annotations to B. For example, state D (where o_1 is a concrete cyclic list) is an instance of B.

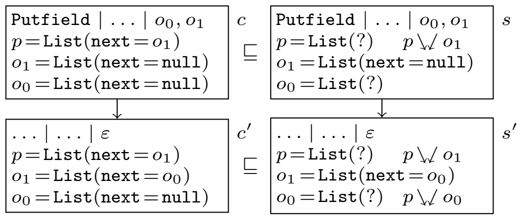
In C, evaluation of `Getfield` throws a `NullPointer` exception. If an exception handler for this type is defined, evaluation would continue there and a reference to the `NullPointer` object is pushed to the operand stack. But here, no such handler exists and E reaches a program end. Here, the call stack is empty and the exception component e is no longer $\perp$, but an object o_2 of type `NullPointer`.

Definition 5 (Instance refinement). *Let $s \in$ STATES where h is the heap of s and $h(o) = (cl, ?)$. Let $cl_1, \ldots, cl_n$ be all non-abstract (not necessarily proper) subtypes of cl. Then $\{s_{\texttt{null}}, s_1, \ldots, s_n\}$ is an* instance refinement *of s. Here, $s_{\texttt{null}} = s[o/\texttt{null}]$ and in s_i, we replace o by a fresh reference o_i pointing to an object of type cl_i. For all fields $v_{i,1} \ldots v_{i,m_i}$ of cl_i (where $v_{i,j}$ has type $cl_{i,j}$), a new reference $o_{i,j}$ is generated which points to the most general value $vl_{i,j}$ of type $cl_{i,j}$, i.e., $(-\infty, \infty)$ for integers and $cl_{i,j}(?)$ for reference types. Then s_i is $(s + \{o_i \mapsto (cl_i, f_i), o_{i,1} \mapsto vl_{i,1}, \ldots, o_{i,m_i} \mapsto vl_{i,m_i}\})[o/o_i]$, where $f_i(v_{i,j}) = o_{i,j}$ for all j. Moreover, new annotations are added in s_i: If s contained $o' \searrow\!\!\swarrow o$, we add $o' =^? o_{i,j}$ and $o' \searrow\!\!\swarrow o_{i,j}$ for all j.[7] If we had $o!$, we also add $o_{i,j}!$, $o_i =^? o_{i,j}$, $o_i \searrow\!\!\swarrow o_{i,j}$, $o_{i,j} =^? o_{i,j'}$, and $o_{i,j} \searrow\!\!\swarrow o_{i,j'}$ for all j, j' with $j \neq j'$.*

2.3 Defining Symbolic Evaluation and Termination Graphs

To define symbolic evaluation formally, for every JINJA instruction, we formulate a corresponding inference rule for symbolic evaluation of our abstract states. This is straightforward for all JINJA instructions except `Putfield`. Thus, in Def. 6 we only present the rules corresponding to a simple JINJA Bytecode instruction (`Load`) and to `Putfield`. We will show in Sect. 3 that on non-abstract states, our inference rules indeed simulate the semantics of JINJA.

For a state s whose topmost frame has m local variables with values $o_0, \ldots, o_m$, "`Load` b" pushes the value o_b of the bth local variable to the operand stack. Executing "`Putfield` v" in a state with the operand stack $o_0, o_1, \ldots, o_k$ means that one wants to write o_0 to the field v of the object at address o_1. This is only allowed if there is no annotation "$o_1 =^? o$" for any o. Then the function f that maps every field of o_1 to its value is updated such that v is now mapped to o_0.



Putfield | ... | o_0, o_1
p = List(next = o_1)
o_1 = List(next = null)
o_0 = List(next = null)
c
$\sqsubseteq$
Putfield | ... | o_0, o_1
p = List(?) $p \searrow\!\!\swarrow o_1$
o_1 = List(next = null)
o_0 = List(?)
s
... | ... | ε
p = List(next = o_1)
o_1 = List(next = o_0)
o_0 = List(next = null)
c'
$\sqsubseteq$
... | ... | ε
p = List(?) $p \searrow\!\!\swarrow o_1$
o_1 = List(next = o_0)
o_0 = List(?) $p \searrow\!\!\swarrow o_0$
s'

Fig. 7. `Putfield` and annotations

However, we may also have to update annotations when evaluating `Putfield`. Consider the concrete state c and the abstract state s in Fig. 7. We have $c \sqsubseteq s$, as the connection between p and o_1 in c (i.e., $p \to_c^* o_1$) was replaced by "$p \searrow\!\!\swarrow o_1$" in s. In both states, we consider a `Putfield` instruction which writes o_0 into the field `next` of o_1. For c, we obtain the state c' where we we now also have $p \to_{c'}^* o_0$. However, to evaluate `Putfield` in the abstract state s, it is not sufficient to just write o_0 to the field `next` of o_1. Then c' would not be an instance of the resulting state s', since s' would not represent the connection between p and o_0. Therefore, we have to add "$p \searrow\!\!\swarrow o_0$" in s'. Now $c' \sqsubseteq s'$ indeed holds. A similar problem was discussed for node U of Fig. 5, where we had to add "!" annotations after evaluating `Putfield`.

[7] Of course, if $cl_{i,j}$ and the type of o' have no common subtype or one of them is `int`, then $o' =^? o_{i,j}$ does not need to be added.

To specify when we need such additional annotations, for any state s let $o \sim_s o'$ denote that "$o =^? o'$" or "$o \searrow\!\swarrow o'$" is contained in s. Then we define $\rightsquigarrow_s$ as $\to_s^* \circ (= \cup \sim_s)$, i.e., $o \rightsquigarrow_s o''$ iff there is an o' with $o \to_s^* o'$, where $o' = o''$ or $o' \sim_s o''$. We drop the index "s" if s is clear from the context. For example, in Fig. 7, we have $p \to_{c'}^* o_1$, $p \to_{c'}^* o_0$ and $p \rightsquigarrow_{s'} o_1$, $p \rightsquigarrow_{s'} o_0$.

Consider a `Putfield` instruction which writes the reference o_0 into the instance referenced by o_1. After evaluation, o_1 may reach any reference q that could be reached by o_0 up to now. Moreover, q cannot only be reached from o_1, but from every reference p that could possibly reach o_1 up to now. Therefore, we must add "$p \searrow\!\swarrow q$" for all p, q with $p \sim o_1$ and $o_0 \rightsquigarrow q$.

Moreover, `Putfield` may create new non-tree shaped objects if there is a reference p that can reach a reference q in several ways after the evaluation. This can only happen if $p \rightsquigarrow q$ and $p \rightsquigarrow o_1$ held before (otherwise p would not be influenced by `Putfield`). If the new field content o_0 could also reach q ($o_0 \rightsquigarrow q$), a second connection from p over o_0 to q may be created by the evaluation. Then we have to add "$p!$" for all p for which a q exists such that $p \rightsquigarrow q$, $p \rightsquigarrow o_1$, and $o_0 \rightsquigarrow q$.[8] It suffices to do this for references p where the paths from p to o_1 and from p to q do not have a common non-empty prefix.

Finally, o_0 could have reached a non-tree shaped object or a reference q marked with !. In this case, we have to add "$p!$" for all p with $p \sim o_1$.

In Def. 6, for any mapping h, let $h + \{k \mapsto d\}$ be the function that maps k to d and every $k' \neq k$ to $h(k')$. For $pp \in \textsc{ProgPos}$, let $pp + 1$ be the position of the next instruction. Moreover, $instr(pp)$ is the instruction at position pp.

Definition 6 (Symbolic evaluation $\xrightarrow{SyEv}$). *For every JINJA instruction, we define a corresponding inference rule for symbolic evaluation of states. We write $s \xrightarrow{SyEv} s'$ if s is transformed to s' by one of these rules. Below, we give the rules for* `Load` *and* `Putfield` *(in the case where no exception was thrown). The rules for the other instructions are analogous.*

$$\frac{\begin{array}{c} s = (\langle (pp, lv, os), fr_1, \ldots, fr_n \rangle, \bot, h, a) \\ instr(pp) = \texttt{Load}\ b \qquad lv = o_0, \ldots, o_m \qquad os = o'_0, \ldots, o'_k \end{array}}{s' = (\langle (pp+1, lv, os'), fr_1, \ldots, fr_n \rangle, \bot, h, a) \qquad os' = o_b, o'_0, \ldots, o'_k}$$

$$\frac{\begin{array}{c} s = (\langle (pp, lv, os), fr_1, \ldots, fr_n \rangle, \bot, h, a) \\ instr(pp) = \texttt{Putfield}\ v \qquad os = o_0, o_1, o_2, \ldots, o_k \\ h(o_1) = (cl, f) \in \textsc{Instances} \qquad a \text{ contains no annotation } o_1 =^? o \end{array}}{\begin{array}{c} s' = (\langle (pp+1, lv, os'), fr_1, \ldots, fr_n \rangle, \bot, h', a') \qquad os' = o_2, \ldots, o_k \\ h' = h + (o_1 \mapsto (cl, f')) \qquad f' = f + (v \mapsto o_0) \end{array}}$$

In the rule for `Putfield`, *a' contains all annotations in a, and in addition:*

- *a' contains "$p \searrow\!\swarrow q$" for all p, q with $p \sim_s o_1$ and $o_0 \rightsquigarrow_s q$*
- *a' contains "$p!$" for all p where $p \rightsquigarrow_s q$, $p \rightsquigarrow_s o_1$, $o_0 \rightsquigarrow_s q$ for some q, and where the paths from p to o_1 and p to q have no common non-empty prefix.*

[8] This happened in state T of Fig. 5 where o_3 was written to the field of o_1. We already had $o_1 \rightsquigarrow_T o_3$ and $o_3 \rightsquigarrow_T o_1$, since T contained the annotation "$o_1 \searrow\!\swarrow o_3$". Hence, in the successor state U of T, we had to add the annotations "$o_1!$" and "$o_3!$".

- *if a contains "$q!$" for some q with $o_0 \to_s^* q$ or if there are π, ρ, ρ' with $\rho \neq \rho'$ where $s|_\pi = o_0$ and $s|_{\pi\rho} = s|_{\pi\rho'}$, then a' contains "$p!$" for all p with $p \sim_s o_1$.*

Finally, we define termination graphs formally. As illustrated, termination graphs are constructed by repeatedly expanding those leaves that do not correspond to program ends (i.e., where the call stack is not empty). Whenever possible, we evaluate the abstract state in a leaf (resulting in an *evaluation edge* $\xrightarrow{\text{EVAL}}$). If evaluation is not possible, we use a refinement to perform a case analysis (resulting in *refinement edges* $\xrightarrow{\text{REF}}$). To obtain a finite graph, we introduce more general states whenever a program position is visited a second time in our symbolic evaluation and add appropriate *instance edges* $\xrightarrow{\text{INS}}$. However, we require all cycles of the termination graph to contain at least one evaluation edge.

Definition 7 (Termination graph). *A graph (N, E) with $N \subseteq$ STATES and $E \subseteq N \times \{\text{EVAL}, \text{REF}, \text{INS}\} \times N$ is a* termination graph *if every cycle contains at least one edge labelled with* EVAL *and one of the following holds for each $s \in N$:*

- *s has just one outgoing edge (s, EVAL, s') and $s \xrightarrow{SyEv} s'$.*
- *There is a refinement $\{s_1, \ldots, s_n\}$ of s according to Def. 1, 4, or 5, and the outgoing edges of s are $(s, \text{REF}, s_1), \ldots, (s, \text{REF}, s_n)$.*
- *s has just one outgoing edge (s, INS, s') and $s \sqsubseteq s'$.*
- *s has no outgoing edge and $s = (\varepsilon, e, h, a)$.*

3 Simulating JBC by Concrete States

In this section we show that if one only regards *concrete* states, the rules for symbolic evaluation in Def. 6 correspond to the operational semantics of JINJA.

Definition 8 (Concrete states). *Let $c \in$ STATES and let h be the heap of c. We call c* concrete *iff c contains no annotations and for all $\pi \in \text{SPOS}(c)$, either $c|_\pi =$ `null` or $h(c|_\pi) \in \text{INSTANCES} \cup \{[z, z] \mid z \in \mathbb{Z}\}$.*

Def. 9 recapitulates the definition of JINJA states from [12] in a formulation that is similar to our states. However, integers are not represented by references, there are no integer intervals, no unknown values, and no annotations.

Definition 9 (JINJA states). *Let $\text{VAL} = \mathbb{Z} \cup \text{REFERENCES}$. Then we define:*

$$\begin{aligned}
\text{JINJASTATES} &= (\text{PROGPOS} \times \text{JINJALOCVAR} \times \text{JINJAOPSTACK})^* \times \\
&\ (\{\bot\} \cup \text{REFERENCES}) \times \text{JINJAHEAP} \\
\text{JINJALOCVAR} &= \text{VAL}^* \\
\text{JINJAOPSTACK} &= \text{VAL}^* \\
\text{JINJAHEAP} &= \text{REFERENCES} \to \text{JINJAINSTANCES} \\
\text{JINJAINSTANCES} &= \text{CLASSNAMES} \times (\text{FIELDIDS} \to \text{VAL})
\end{aligned}$$

To define a function TRANS which maps each JINJA state to a corresponding concrete state, we first introduce a function $tr_{Val} : \text{VAL} \to \text{REFERENCES}$ with $tr_{Val}(o) = o$ for all $o \in$ REFERENCES. Moreover, tr_{Val} maps every $z \in \mathbb{Z}$ to a fresh reference o_z. Later, the value of o_z in the heap will be the interval $[z, z]$.

Now we define $tr_{Ins} : \text{JINJAINSTANCES} \to \text{INSTANCES}$. For any $f :$ FIELDIDS

$\to$ VAL, let $tr_{Ins}(cl, f) = (cl, \widetilde{f})$, where $\widetilde{f}(v) = tr_{Val}(f(v))$ for all $v \in$ FIELDIDS.

Next we define tr_{Heap} : JINJAHEAP $\to$ HEAP. For any $h \in$ JINJAHEAP, $tr_{Heap}(h)$ is a function from REFERENCES to INTEGERS $\cup$ INSTANCES. For any $o \in$ REFERENCES, let $tr_{Heap}(h)(o) = tr_{Ins}(h(o))$. Furthermore, we need to add the new references for integers, i.e., $tr_{Heap}(h)(o_z) = [z, z]$ for all $z \in \mathbb{Z}$.

Let tr_{Frame} : (PROGPOS $\times$ JINJALOCVAR $\times$ JINJAOPSTACK) $\to$ (PROGPOS $\times$ LOCVAR $\times$ OPSTACK) with $tr_{Frame}(pp, lv, os) = (pp, \widetilde{lv}, \widetilde{os})$. If $lv = o_0, \ldots, o_m$, $os = o'_0, \ldots, o'_k$, then $\widetilde{lv} = tr_{Val}(o_0), \ldots, tr_{Val}(o_m)$, $\widetilde{os} = tr_{Val}(o'_0), \ldots, tr_{Val}(o'_k)$.

Finally we define TRANS : JINJASTATES $\to$ STATES. For any $j \in$ JINJASTATES with $j = (\langle fr_0, \ldots, fr_n\rangle, e, h)$, let TRANS$(j) = (\langle tr_{Frame}(fr_0), \ldots, tr_{Frame}(fr_n)\rangle, e', tr_{Heap}(h), \varnothing)$, where $e' = \perp$ if $e = \perp$ and $e' = tr_{Val}(e)$ otherwise.

For $j, j' \in$ JINJASTATES, $j \xrightarrow{jvm} j'$ denotes that evaluating j one step according to the semantics of JINJA [12] leads to j'. Thm. 10 shows that $\xrightarrow{jvm}$ can be simulated by the evaluation of concrete states as defined in Def. 6, cf. Fig. 1.

Theorem 10 (Evaluation of concrete states simulates JINJA evaluation). *For all $j, j' \in$ JINJASTATES, $j \xrightarrow{jvm} j'$ implies* TRANS$(j) \xrightarrow{SyEv}$ TRANS(j').

Proof. We give the proof for the most complex JINJA instruction (i.e., `Putfield` in the case where no exception was thrown). The proof is analogous for the other instructions. Here, $\xrightarrow{jvm}$ is defined by the following inference rule.

$$\frac{\begin{array}{c} j = (\langle (pp, lv, os), fr_1, \ldots, fr_n\rangle, \perp, h) \qquad instr(pp) = \texttt{Putfield}\ v \\ os = o_0, o_1, o_2, \ldots, o_k \qquad h(o_1) = (cl, f) \in \text{JINJAINSTANCES} \end{array}}{\begin{array}{c} j' = (\langle (pp+1, lv, os'), fr_1, \ldots, fr_n\rangle, \perp, h') \qquad os' = o_2, \ldots, o_k \\ h' = h + (o_1 \mapsto (cl, f')) \qquad f' = f + (v \mapsto o_0) \end{array}}$$

Let $j \xrightarrow{jvm} j'$ by the above rule. Then TRANS$(j) = (\langle (pp, \widetilde{lv}, \widetilde{os}), tr_{Frame}(fr_1), \ldots, tr_{Frame}(fr_n)\rangle, \perp, tr_{Heap}(h), \varnothing)$ with $\widetilde{os} = tr_{Val}(o_0), tr_{Val}(o_1), \ldots, tr_{Val}(o_k)$. Note that $tr_{Val}(o_1) = o_1$. Moreover, TRANS$(j') = (\langle (pp+1, \widetilde{lv}, \widetilde{os'}), tr_{Frame}(fr_1), \ldots, tr_{Frame}(fr_n)\rangle, \perp, tr_{Heap}(h'), \varnothing)$ with $\widetilde{os'} = tr_{Val}(o_2), \ldots, tr_{Val}(o_k)$.

On the other hand, by Def. 6 for $c =$ TRANS(j), we have $c \xrightarrow{SyEv} c'$ with $c' = (\langle (pp+1, \widetilde{lv}, \widetilde{os'}), tr_{Frame}(fr_1), \ldots, tr_{Frame}(fr_n)\rangle, \perp, tr_{Heap}(h)', \varnothing)$. It remains to show that $tr_{Heap}(h') = tr_{Heap}(h)'$. For any new reference o_z for integers, we have $tr_{Heap}(h')(o_z) = [z, z] = tr_{Heap}(h)'(o_z)$. For any $o \in$ REFERENCES $\setminus \{o_1\}$, we have $tr_{Heap}(h')(o) = tr_{Ins}(h'(o)) = tr_{Ins}(h(o))$ and $tr_{Heap}(h)'(o) = tr_{Heap}(h)(o) = tr_{Ins}(h(o))$. Finally, $tr_{Heap}(h')(o_1) = tr_{Ins}(h'(o_1)) = tr_{Ins}(cl, f') = (cl, \widetilde{f'})$ where $\widetilde{f'}(v) = tr_{Val}(o_0)$ and $\widetilde{f'}(w) = tr_{Val}(f(w))$ for all $w \in$ FIELDIDS $\setminus \{v\}$. Moreover, $tr_{Heap}(h)'(o_1) = (cl, (\widetilde{f})')$ where $(\widetilde{f})'(v) = tr_{Val}(o_0)$ and $(\widetilde{f})'(w) = \widetilde{f}(w) = tr_{Val}(f(w))$ for all $w \in$ FIELDIDS $\setminus \{v\}$. $\square$

4 Simulating Concrete States by Abstract States

Now we show that our symbolic evaluation on *abstract* states is indeed consistent with the evaluation of all represented *concrete* states, cf. the upper half of Fig. 1.

Theorem 11 (Evaluation of abstract states simulates evaluation of concrete states). *Let $c, c', s \in$ STATES, where c is concrete, $c \xrightarrow{SyEv} c'$, $c \sqsubseteq s$, and s occurs in a termination graph G. Then G contains a path $s(\xrightarrow{\text{INS}} \cup \xrightarrow{\text{REF}})^* \circ \xrightarrow{\text{EVAL}} s'$ such that $c' \sqsubseteq s'$.*

Proof. We prove the theorem by induction on the sum of the lengths of all paths from s to the next $\xrightarrow{\text{EVAL}}$ edge. This sum is always finite, since every cycle of a termination graph contains an evaluation edge, cf. Def. 7. We perform a case analysis on the type of the outgoing edges of s. If there is an edge $s \xrightarrow{\text{INS}} \tilde{s}$, and hence $s \sqsubseteq \tilde{s}$, we prove transitivity of $\sqsubseteq$ (Lemma 13, Sect. 4.1). Then $c \sqsubseteq s$ implies $c \sqsubseteq \tilde{s}$ and the claim follows from the induction hypothesis.

If the outgoing edges of s are $\xrightarrow{\text{REF}}$ edges (i.e., $s \xrightarrow{\text{REF}} s_1, \ldots, s \xrightarrow{\text{REF}} s_n$), we show that our refinements are "*valid*", i.e., $c \sqsubseteq s$ implies $c \sqsubseteq s_j$ for some s_j (Lemmas 14-16, Sect. 4.2). Again, then the claim follows from the induction hypothesis.

Finally, if the first step is an $\xrightarrow{\text{EVAL}}$-step (i.e., $s \xrightarrow{SyEv} s'$), we prove the correctness of the $\xrightarrow{SyEv}$ relation on abstract states (Lemma 19, Sect. 4.3). □

With Thm. 10 and 11, we can prove the "soundness" of termination graphs.

Corollary 12 (Soundness of termination graphs). *Let $j_1 \in$ JINJASTATES have an infinite evaluation $j_1 \xrightarrow{jvm} j_2 \xrightarrow{jvm} \ldots$ and let G be a termination graph with a state s_1^1 such that* TRANS$(j_1) \sqsubseteq s_1^1$. *Then G contains an infinite* computation path $s_1^1, \ldots, s_1^{n_1}, s_2^1, \ldots, s_2^{n_2}, \ldots$ *such that* TRANS$(j_i) \sqsubseteq s_i^1$ *for all i.*

Proof. The corollary follows directly from Thm. 10 and 11, cf. Sect. 1. □

As shown in [15, Thm. 3.7], if the TRS resulting from a termination graph is terminating, then there is no infinite computation path. Thus, Cor. 12 proves the soundness of our approach for automated termination analysis of JBC.

4.1 Transitivity of $\sqsubseteq$

Lemma 13 ($\sqsubseteq$ transitive). *If $s'' \sqsubseteq s'$ and $s' \sqsubseteq s$, then also $s'' \sqsubseteq s$.*

Proof. We prove the lemma by checking each of the conditions in Def. 3. Here, we only consider Def. 3(a)-(d) and refer to [3] for the (similar) proof of the remaining conditions. Let $\pi \in$ SPOS(s) and let h (h', h'') be the heap of s (s', s''). Note that $\pi \in$ SPOS(s) implies $\pi \in$ SPOS(s') and $\pi \in$ SPOS(s''), cf. [15, Lemma 4.1].

(a) If $h''(s''|_\pi) \in$ INTEGERS, then because of $s'' \sqsubseteq s'$ also $h'(s'|_\pi) \in$ INTEGERS and thus $h(s|_\pi) \in$ INTEGERS. We also have $h''(s''|_\pi) \subseteq h'(s'|_\pi) \subseteq h(s|_\pi)$.

(b) If $s''|_\pi = \texttt{null}$, then by $s'' \sqsubseteq s'$ we have either
$s'|_\pi = \texttt{null}$ and thus, $s|_\pi = \texttt{null}$ or $h(s|_\pi) \in$ UNKNOWN
or $h'(s'|_\pi) \in$ UNKNOWN and thus, $h(s|_\pi) \in$ UNKNOWN.

(c) If $h''(s''|_\pi) = (cl'', ?)$, then $h'(s'|_\pi) = (cl', ?)$ and thus also $h(s|_\pi) = (cl, ?)$. Here, cl'' is cl' or a subtype of cl', and cl' is cl or a subtype of cl.
Note that the subtype relation of JBC types is transitive by definition.

(d) If $h''(s''|_\pi) = (cl'', f'') \in$ INSTANCES, then either

$h'(s'|_\pi) = (cl', ?)$ and thus, also $h(s|_\pi) = (cl, ?)$
or $h'(s'|_\pi) = (cl'', f') \in$ INSTANCES and thus,
either $h(s|_\pi) = (cl, ?)$ or $h(s|_\pi) = (cl'', f) \in$ INSTANCES.
Again, cl'' is cl' or a subtype of cl', and cl' is cl or a subtype of cl. □

4.2 Validity of refinements

We say that a refinement $\rho : \text{STATES} \to 2^{\text{STATES}}$ is *valid* iff for all $s \in$ STATES and all concrete states c, $c \sqsubseteq s$ implies that there is an $s' \in \rho(s)$ such that $c \sqsubseteq s'$. We now prove the validity of our refinements from Def. 1, 4, and 5.

Lemma 14. *The integer refinement is valid.*

Proof. Let $\{s_1, \ldots, s_n\}$ be an *integer refinement* of s where $s_i = (s + \{o_i \mapsto V_i\})[o/o_i]$ and $h_s(o) = V = V_1 \cup \ldots \cup V_n \subseteq \mathbb{Z}$ for the heap h_s of s.

Let c be a concrete state with heap h_c and $c \sqsubseteq s$. Let $\Pi = \{\pi \in \text{SPOS}(s) \mid s|_\pi = o\}$. By Def. 3(e), there is a $z \in \mathbb{Z}$ such that $h_c(c|_\pi) = [z, z]$ for all $\pi \in \Pi$. Let $z \in V_i$ and let h_{s_i} be the heap of s_i. Then $h_{s_i}(s_i|_\pi) = V_i$ for all $\pi \in \Pi$.

To show $c \sqsubseteq s_i$, we only have to check condition Def. 3(a). Let $\tau \in \text{SPOS}(c) \cap \text{SPOS}(s_i)$ with $h_c(c|_\tau) = [z', z'] \in$ INTEGERS. If $\tau \notin \Pi$, then this position was not affected by the integer refinement and thus, $h_c(c|_\tau) \subseteq h_s(s|_\tau) = h_{s_i}(s_i|_\tau)$. If $\tau \in \Pi$, then we have $z' = z$ and thus $h_c(c|_\tau) \subseteq V_i = h_{s_i}(s_i|_\tau)$. □

Lemma 15. *The equality refinement is valid.*

Proof. Let $\{s_=, s_{\neq}\}$ be an equality refinement of s, using the annotation $o =^? o'$. Let c be a concrete state with $c \sqsubseteq s$. We want to prove that $c \sqsubseteq s_{\neq}$ or $c \sqsubseteq s_=$.

Let $\Pi = \{\tau \in \text{SPOS}(s) \mid s|_\tau = o\}$, $\Pi' = \{\tau' \in \text{SPOS}(s) \mid s|_{\tau'} = o'\}$. By Def. 3(e) there are o_c and o'_c with $c|_\tau = o_c$ for all $\tau \in \Pi$ and $c|_{\tau'} = o'_c$ for all $\tau' \in \Pi'$.

If $o_c \neq o'_c$, we trivially have $c \sqsubseteq s_{\neq}$, as $s_{\neq}$ differs from s only in the removed annotation "$o =^? o'$" which is not needed when regarding instances like c.

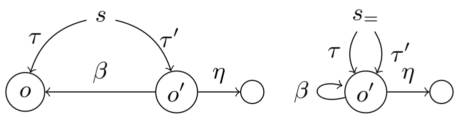


Fig. 8. Illustrating Lemma 15

If $o_c = o'_c$, we prove $c \sqsubseteq s_=$. The only change between s and $s_=$ was on or below positions in Π. Consider Fig. 8, where a state s with $s|_\tau = o$ and $s|_{\tau'} = o'$ is depicted on the left (i.e., $\tau \in \Pi$ and $\tau' \in \Pi'$). When we perform an equality refinement and replace o by o', we reach the state $s_=$ on the right. As illustrated there, we can decompose any position $\pi \in \text{SPOS}(s_=)$ with a prefix in Π into $\tau\beta\eta$, where τ is the shortest prefix in Π and $\tau\beta$ is the longest prefix with $s_=|_{\tau\beta} = s_=|_\tau$.

With this decomposition, we have $s_=|_\tau = s|_{\tau'}$ for $\tau' \in \Pi'$ and thus $s_=|_{\tau\beta\eta} = s_=|_{\tau\eta} = s_=|_{\tau'\eta} = s|_{\tau'\eta}$. For $c \sqsubseteq s_=$, we now only have to check the conditions of Def. 3 for any position of $s_=$ of the form $\tau\beta\eta$ as above. Then the claim follows directly, as the conditions of Def. 3 already hold for $\tau'\eta$, since $c \sqsubseteq s$. □

Lemma 16. *The instance refinement is valid.*

Proof. Let $S = \{s_{\texttt{null}}, s_1, \ldots, s_n\}$ be an instance refinement of s on reference o. Let c be concrete with heap h_c and $c \sqsubseteq s$. We prove that $c \sqsubseteq s'$ for some $s' \in S$.

By Def. 5, $h_s(o) = (cl, ?)$, where h_s is the heap of s. Let $\Pi = \{\pi \in \text{SPos}(s) \mid s|_\pi = o\}$. The instance refinement only changed values at positions in Π and below. It may have added annotations for references at other positions, but as annotations only allow *more* sharing effects, we do not have to consider these positions. By Def. 3(e), there is an o_c such that $c|_\pi = o_c$ for all $\pi \in \Pi$. If $o_c = \texttt{null}$, we set $s' = s_{\texttt{null}}$. If $h_c(o_c) = (cl_i, f)$, we set $s' = s_i$, where s_i is obtained by refining the type cl to cl_i. Now one can prove $c \sqsubseteq s'$ by checking all conditions of Def. 3, as in the proof of Lemma 13. For the full proof, see [3]. □

4.3 Correctness of symbolic evaluation

Finally, we prove that every evaluation of a concrete state is also represented by the evaluation of the corresponding abstract state. This is trivial for most instructions, since they only affect the values of local variables or the operand stack. The only instruction which changes data objects on the heap is `Putfield`.

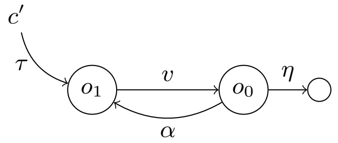


Fig. 9. Illustrating Lemma 17

Consider the evaluation of a concrete state c to another state c' by executing "`Putfield` v" which writes o_0 to the field v of the object at address o_1. Similar to the proof of Lemma 15, every position π of c' where the state was changed can be decomposed into $\pi = \tau\beta\eta$. Here, the first part τ leads to o_1 and it is the longest prefix that is not affected by the evaluation of `Putfield`. Similarly, the last part η is the longest suffix of π that was not changed by evaluating `Putfield`. So in particular, $c'|_{\tau\beta} = o_0$. The middle part β contains those parts that were actually changed in the evaluation step. So usually, β is just the field v. However, if $o_0 \rightarrow_c^* o_1$, then the object at o_1 in c' has become cyclic and then β can be more complex. Consider Fig. 9, where $c'|_\tau = o_1$ and regard the position $\pi = \tau v\, \alpha\, v\, \eta$. Here, the position π was influenced twice by the evaluation, as the middle part $\beta = v\, \alpha\, v$ contains a cycle using the field v. In the following, let $\pi_1 < \pi_2$ denote that π_1 is a proper prefix of π_2 and let $\leq$ be the reflexive closure of $<$.

Definition 17 (Change of concrete states by `Putfield`). *Let $c \in$ STATES be concrete with "`Putfield` v" as the next instruction to be evaluated and $c|_{\text{OS}_{0,1}} \neq$ `null`. Let $c \overset{SyEv}{\longrightarrow} c'$ (i.e., in c', the object at reference $c|_{\text{OS}_{0,0}}$ has been written to the field v of the object at reference $c|_{\text{OS}_{0,1}}$). Then δ denotes the function that maps positions in c', which has a shorter operand stack than c, to positions in c, i.e., $\delta(w\,\pi) = w\,\pi$ if $w \neq \text{OS}_{0,j}$ and $\delta(w\,\pi) = \text{OS}_{0,j+2}\ \pi$ if $w = \text{OS}_{0,j}$. For any $\pi \in \text{SPos}(c')$ with $c'|_\pi \neq c|_{\delta(\pi)}$, its* Putfield-decomposition *is $\pi = \tau\beta\eta$, where*

- *τ is the shortest prefix of π such that both $c'|_\tau = c|_{\text{OS}_{0,1}}$ and $\tau\, v \leq \pi$,*
- *β is the longest position of the form $\beta = v\, \alpha_1\, v\, \alpha_2\, v \ldots\, v\, \alpha_n\, v$ for some $n \geq 0$ where $\tau\beta \leq \pi$, $c'|_{\tau\, v\, \alpha_j} = c|_{\text{OS}_{0,1}}$, and $c'|_{\tau\, v\, \rho} \neq c|_{\text{OS}_{0,1}}$ for all $\rho < \alpha_j$ and all $1 \leq j \leq n$. Note that this implies $c'|_{\tau\,\beta} = c'|_{\tau\, v} = c|_{\text{OS}_{0,0}}$ and $c'|_\pi = c|_{\text{OS}_{0,0}\ \eta}$.*

We now show that Putfield-decompositions can be lifted to abstract states.

Lemma 18 (Change of abstract states by `Putfield`). *Let $s \in$ STATES with*

"`Putfield` *v" as next instruction and $s|_{\mathrm{OS}_{0,1}} \neq$* `null` *. Let $s \xrightarrow{SyEv} s'$ and let c be concrete with $c \sqsubseteq s$ and $c \xrightarrow{SyEv} c'$. For any $\pi \in \mathrm{SPos}(s') \cap \mathrm{SPos}(c')$, we have:*

- *If $c'|_\pi = c|_{\delta(\pi)}$, then $s'|_\pi = s|_{\delta(\pi)}$.*
- *If $c'|_\pi \neq c|_{\delta(\pi)}$, then for the corresponding Putfield-decomposition $\pi = \tau\beta\eta$, we have $s'|_\tau = s|_{\mathrm{OS}_{0,1}}$, $s'|_{\tau\beta} = s'|_{\tau\, v} = s|_{\mathrm{OS}_{0,0}}$, and $s'|_\pi = s|_{\mathrm{OS}_{0,0}\, \eta}$.*

Proof. Note that $s|_{\mathrm{OS}_{0,1}} \neq$ `null` also implies $c|_{\mathrm{OS}_{0,1}} \neq$ `null`, since $c \sqsubseteq s$. Hence, $c'|_\pi = c|_{\delta(\pi)}$ means that the position π is not influenced by the `Putfield` instruction. This implies that we also have $s'|_\pi = s|_{\delta(\pi)}$.

Now let $c'|_\pi \neq c|_{\delta(\pi)}$. Since τ is the shortest prefix with $c'|_\tau = c|_{\mathrm{OS}_{0,1}}$ and $\tau\, v \leq \pi$, this path is not affected by the evaluation, i.e., $c'|_\tau = c|_{\delta(\tau)}$ and $s'|_\tau = s|_{\delta(\tau)}$.

Assume that $s'|_\tau \neq s|_{\mathrm{OS}_{0,1}}$. As $s \xrightarrow{SyEv} s'$, we have $h_s(s|_{\mathrm{OS}_{0,1}}) \in \textsc{Instances}$, where h_s is the heap of the state s. But then, as $c|_{\delta(\tau)} = c'|_\tau = c|_{\mathrm{OS}_{0,1}}$ and $s|_{\delta(\tau)} = s'|_\tau \neq s|_{\mathrm{OS}_{0,1}}$, $c \sqsubseteq s$ implies $s|_{\delta(\tau)} =^? s|_{\mathrm{OS}_{0,1}}$. This contradicts the definition of $=^?$, which requires that at least one of $h_s(s|_{\delta(\tau)})$ and $h_s(s|_{\mathrm{OS}_{0,1}})$ must be in $\textsc{Unknown}$. But here, we do not only have $h_s(s|_{\mathrm{OS}_{0,1}}) \in \textsc{Instances}$, but also $h_s(s|_{\delta(\tau)}) \in \textsc{Instances}$. The reason for the latter is that since $\pi \in \mathrm{SPos}(s')$ and $\tau < \pi$, we have $h_{s'}(s'|_\tau) \in \textsc{Instances}$ and thus also $h_s(s|_{\delta(\tau)}) \in \textsc{Instances}$ (recall that $s'|_\tau = s|_{\delta(\tau)}$). Thus, we have shown that $s'|_\tau = s|_{\mathrm{OS}_{0,1}}$. The proof for $s'|_{\tau\beta} = s|_{\mathrm{OS}_{0,0}}$ works analogously (see [3] for details). As η was not affected by `Putfield`, $s'|_{\tau\beta} = s|_{\mathrm{OS}_{0,0}}$ implies $s'|_\pi = s|_{\mathrm{OS}_{0,0}\, \eta}$. □

Now we can finally prove the correctness of our evaluation on abstract states.

Lemma 19 (Correctness of evaluation on abstract states). *Let $c, s \in$ $\textsc{States}$ with c concrete and $c \sqsubseteq s$. If $c \xrightarrow{SyEv} c'$ and $s \xrightarrow{SyEv} s'$, then $c' \sqsubseteq s'$.*

Proof. For all instructions except `Putfield`, the claim is obvious. Therefore, we prove the lemma for "`Putfield` v", which writes the reference $c|_{\mathrm{OS}_{0,0}}$ to the field v of the instance referenced by $c|_{\mathrm{OS}_{0,1}}$. As the case for $c|_{\mathrm{OS}_{0,1}} =$ `null` (leading to an exception) is trivial, we will not consider it here and assume $c|_{\mathrm{OS}_{0,1}} \neq$ `null`. To prove that $c' \sqsubseteq s'$, we consider each of the conditions of Def. 3.

For Def. 3(a), (b), (d), (e), (f), for any $\pi \in \mathrm{SPos}(c')$, there are two possibilities. We either have $c'|_\pi = c|_{\delta(\pi)}$ and therefore also $s'|_\pi = s|_{\delta(\pi)}$ by Lemma 18. Then the condition on $c'|_\pi$ and $s'|_\pi$ that is needed for $c' \sqsubseteq s'$ follows from the respective condition on $c|_{\delta(\pi)}$ and $s|_{\delta(\pi)}$, since $c \sqsubseteq s$. Otherwise, $c'|_\pi \neq c|_{\delta(\pi)}$. By Lemma 18, there is a position η such that $c'|_\pi = c|_{\mathrm{OS}_{0,0}\, \eta}$ and $s'|_\pi = s|_{\mathrm{OS}_{0,0}\, \eta}$. Now the condition on $c'|_\pi$ and $s'|_\pi$ that is needed for $c' \sqsubseteq s'$ follows from the respective condition on $c|_{\mathrm{OS}_{0,0}\, \eta}$ and $s|_{\mathrm{OS}_{0,0}\, \eta}$, since we have $c \sqsubseteq s$.

Def. 3 (c), (g), (i), (j) are not applicable, since c is a concrete state. It remains to consider (h) and (k). We only give the proof for (h), since the proof for (k) is analogous. Let $h_c, h_{c'}, h_s, h_{s'}$ be the heaps of c, c', s, and s'. We have $c'|_\pi = c'|_{\pi'}$ and $\{\pi, \pi'\} \not\subseteq \mathrm{SPos}(s')$. We only handle the case where $\pi \notin \mathrm{SPos}(s')$ and $\pi' \in \mathrm{SPos}(s')$ and where both $\pi, \pi' \notin \mathrm{SPos}(s')$. The remaining case is analogous to the first. We now have four possibilities (1)-(4):

(1) $c'|_\pi = c|_{\delta(\pi)}$ and $c'|_{\pi'} = c|_{\delta(\pi')}$, i.e., neither π nor π' were affected by the

`Putfield` operation. Then $\{\delta(\pi), \delta(\pi')\} \not\subseteq \mathrm{SPos}(s)$ and thus due to $c \sqsubseteq s$, we have $s|_{\overline{\delta(\pi)}} \searrow\!\!\swarrow s|_{\overline{\delta(\pi')}}$ and thus also $s'|_{\overline{\pi}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$.

(2) $c'|_{\pi} \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} = c|_{\delta(\pi')}$, i.e., only π was affected by `Putfield`. Let $\pi = \tau\beta\eta$ be the Putfield-decomposition. We can then distinguish three subcases:

(2.1) $\tau\beta \in \mathrm{SPos}(s')$ and $\pi = \tau\beta\eta \notin \mathrm{SPos}(s')$. Then also $\mathrm{OS}_{0,0}\,\eta \notin \mathrm{SPos}(s)$, because otherwise the object at position $h_s(s|_{\mathrm{OS}_{0,0}\,\eta})$ would have been written to position $\pi = \tau\beta\eta$ in s'. We have $c|_{\delta(\pi')} = c'|_{\pi'} = c'|_{\pi} = c|_{\mathrm{OS}_{0,0}\,\eta}$ and as $c \sqsubseteq s$, we have $s|_{\overline{\delta(\pi')}} \searrow\!\!\swarrow s|_{\overline{\mathrm{OS}_{0,0}\,\eta}}$. Note that $\overline{\mathrm{OS}_{0,0}\,\eta} = \mathrm{OS}_{0,0}\,\widetilde{\eta}$ for some $\widetilde{\eta} \leq \eta$. Thus also $s'|_{\overline{\pi'}} \searrow\!\!\swarrow s'|_{\tau\beta\widetilde{\eta}}$ and hence, $s'|_{\overline{\pi'}} \searrow\!\!\swarrow s'|_{\overline{\pi}}$.

(2.2) $\tau \in \mathrm{SPos}(s')$ and $\tau\beta \notin \mathrm{SPos}(s')$. Then for $\beta = v\,\alpha_1\,v\,\alpha_2\,v\ldots\alpha_n\,v$, there is a minimal j with $\tau\,v\,\alpha_1\,\ldots\,v\,\alpha_j \notin \mathrm{SPos}(s')$. We then also have $\mathrm{OS}_{0,0}\;\alpha_j \notin \mathrm{SPos}(s)$. As $c|_{\mathrm{OS}_{0,1}} = c|_{\mathrm{OS}_{0,0}\,\alpha_j}$ by construction of the decomposition and as $c \sqsubseteq s$, we have $s|_{\mathrm{OS}_{0,1}} \searrow\!\!\swarrow s|_{\mathrm{OS}_{0,0}\,\widetilde{\alpha_j}}$ for some $\widetilde{\alpha_j} \leq \alpha_j$.
If $\{\delta(\pi'), \mathrm{OS}_{0,0}\,\eta\} \subseteq \mathrm{SPos}(s)$, by $c \sqsubseteq s$, we have $s|_{\delta(\pi')} = s|_{\mathrm{OS}_{0,0}\,\eta}$ or $s|_{\delta(\pi')} =^{?} s|_{\mathrm{OS}_{0,0}\,\eta}$. If $\{\delta(\pi'), \mathrm{OS}_{0,0}\,\eta\} \not\subseteq \mathrm{SPos}(s)$, by $c \sqsubseteq s$, we have $s|_{\overline{\delta(\pi')}} \searrow\!\!\swarrow s|_{\mathrm{OS}_{0,0}\,\widetilde{\eta}}$ for some $\widetilde{\eta} \leq \eta$. Both cases imply $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\overline{\delta(\pi')}}$.
As we have $s|_{\mathrm{OS}_{0,0}\,\widetilde{\alpha_j}} \sim s|_{\mathrm{OS}_{0,1}}$ and $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\overline{\delta(\pi')}}$, the first rule for the annotation additions (from Def. 6) requires $s'|_{\tau\,v\,\widetilde{\alpha_j}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$. Hence, $s'|_{\tau\,v\,\alpha_1\ldots v\,\widetilde{\alpha_j}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$ and thus, $s'|_{\overline{\pi}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$.

(2.3) $\tau \notin \mathrm{SPos}(s')$. Then also $\delta(\tau) \notin \mathrm{SPos}(s)$ and as $c|_{\delta(\tau)} = c|_{\mathrm{OS}_{0,1}}$ and $c \sqsubseteq s$, we have $s|_{\overline{\delta(\tau)}} \searrow\!\!\swarrow s|_{\mathrm{OS}_{0,1}}$. We also have $c|_{\mathrm{OS}_{0,0}\,\eta} = c|_{\delta(\pi')}$ and thus either $s|_{\mathrm{OS}_{0,0}\,\widetilde{\eta}} = s|_{\overline{\delta(\pi')}}$, $s|_{\mathrm{OS}_{0,0}\,\widetilde{\eta}} =^{?} s|_{\overline{\delta(\pi')}}$, or $s|_{\mathrm{OS}_{0,0}\,\widetilde{\eta}} \searrow\!\!\swarrow s|_{\overline{\delta(\pi')}}$ for $\widetilde{\eta} \leq \eta$. In all cases, $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\overline{\delta(\pi')}}$. Together with $s|_{\overline{\delta(\tau)}} \sim s|_{\mathrm{OS}_{0,1}}$, the first rule for the annotation additions requires $s'|_{\overline{\tau}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$ and hence, $s'|_{\overline{\pi}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$.

(3) $c'|_{\pi} = c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$, i.e., only π' was affected by the `Putfield` operation. This is analogous to Case (2).

(4) $c'|_{\pi} \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$, i.e., both π, π' were affected by `Putfield`. This is proved by a case analysis similar to Case (2) (see [3] for details). □

5 Conclusion

In this paper, we have shown that termination graphs correctly simulate the evaluation of JBC. To this end, we first gave a formal definition of termination graphs (Sect. 2). Then we showed that our notion of symbolic evaluation in these graphs corresponds to the operational semantics of JINJA Bytecode, as long as we are restricted to concrete states (Sect. 3). Afterwards, we proved that every evaluation of concrete states is simulated by a path on abstract states in the termination graph (Sect. 4). Together with the results of [15], this proves the soundness of our approach for automated termination analysis of JBC. Here, JBC is first transformed into termination graphs. Afterwards, one generates TRSs from these graphs and uses existing tools to prove their termination.

The result of the current paper (i.e., the proof that every JBC evaluation is represented by the termination graph) is also useful outside of termination analysis, since termination graphs could also be used for analysis of nullness, sharing, exceptions, etc. Compared to other static analysis techniques, termination graphs perform less abstraction and therefore, while the analysis may be more time-consuming, it can be more precise. Developing such other analyses that build upon termination graphs is the subject of future work.

Acknowledgement. J. Giesl wants to thank C. Walther for having introduced him to many of the research areas that are relevant for this paper (e.g., induction and symbolic evaluation [19], termination [20], and semantics [21]).

References

1. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, D. Zanardini. Termination analysis of Java Bytecode. In *Proc. FMOODS '08*, LNCS 5051, pages 2–18, 2008.
2. J. Berdine, B. Cook, D. Distefano, P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. *Proc. CAV '06*, LNCS 4144, p. 386-400, 2006.
3. M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl. Termination graphs for Java Bytecode. Technical Report AIB-2010-15, RWTH Aachen, 2010. `http://aib.informatik.rwth-aachen.de`.
4. M. Colón and H. Sipma. Practical methods for proving program termination. In *Proc. CAV '02*, LNCS 2404, pages 442–454, 2002.
5. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. PLDI '06*, pages 415–426. ACM Press, 2006.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL '77*, pages 238–252. ACM Press, 1977.
7. D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.
8. N. Dershowitz. Termination of rewriting. *J. Symb. Comp.*, 3(1-2):69–116, 1987.
9. C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, S. Falke. Proving termination of integer term rewriting. In *Proc. RTA '09*, LNCS 5595, pages 32–47, 2009.
10. J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM TOPLAS,* to appear.
11. J. Giesl, P. Schneider-Kamp, R. Thiemann. AProVE 1.2: Automatic termination proofs in the DP framework. *Proc. IJCAR '06*, LNAI 4130, pages 281–286, 2006.
12. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM TOPLAS*, 28(4):619–695, 2006.
13. T. Lindholm and F. Yellin. *Java Virtual Machine Specification.* Prentice Hall, 1999.
14. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* LNCS 2283. Springer, 2002.
15. C. Otto, M. Brockschmidt, C. von Essen, J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. *Proc. RTA '10*, LIPIcs 6, p. 259-276, 2010.
16. P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM TOCL*, 11(1), 2009.
17. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In *Proc. ILPS '95*, pages 465–479. MIT Press, 1995.
18. F. Spoto, F. Mesnard, and É. Payet. A termination analyser for Java Bytecode based on path-length. *ACM TOPLAS*, 32(3), 2010.
19. C. Walther. Mathematical induction. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 127–227. Oxford University Press, 1994.
20. C. Walther. On proving the termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157, 1994.
21. C. Walther. *Semantik und Programmverifikation.* Teubner-Wiley, 2001.