From Jinja Bytecode to Term Rewriting: A Complexity Reflecting Transformation

Georg Moser[a], Michael Schaper[a]

[a] *Institute of Computer Science, University of Innsbruck, Austria*

Abstract

In this paper we show how the runtime complexity of *imperative* programs can be analysed fully automatically by a transformation to *term rewrite systems*, the complexity of which can then be automatically verified by existing complexity tools. We restrict to well-formed *Jinja bytecode* programs that only make use of non-recursive methods. The analysis can handle programs with cyclic data only if the termination behaviour is independent thereof.

We exploit a *term-based* abstraction of programs within the abstract interpretation framework. The proposed transformation encompasses two stages. For the first stage we perform a combined control and data flow analysis by evaluating program states symbolically, which is shown to yield a finite representation of all execution paths of the given program through a graph, dubbed *computation graph*. In the second stage we encode the (finite) computation graph as a term rewrite system. This is done while carefully analysing complexity preservation and reflection of the employed transformations such that the complexity of the obtained term rewrite system reflects on the complexity of the initial program. Finally, we show how the approach can be automated and provide ample experimental evidence of the advantages of the proposed analysis.

Keywords: program transformation, term rewriting, termination and resource analysis, automation
2000 MSC: 68Q42,
2000 MSC: 03D15,
2000 MSC: 03B70

1. Introduction

In recent years research on complexity analysis of first-order term rewrite systems has matured and a number of noteworthy results could be established. First we give a quantitative assessment based on the annual competition of complexity analysers within TERMCOMP.[1] With respect to last year's run of TERMCOMP, we see a success rate of 42% (60%) in the category *Runtime Complexity – Innermost Rewriting*. This benchmark roughly represents runtime complexity analysis of functional programs evaluated in an eager fashion and success here means that a polynomial upper (lower) bound could be established fully automatically. It is known that a considerable amount of examples of the benchmark does not exhibit polynomial runtime complexity or even termination. With respect to a qualitative assessment we want to mention (i) efforts to suit existing termination techniques to complexity analysis [1–4], (ii) adaptions of the dependency pair method to complexity [5–7], (iii) the ongoing quest to incorporate compositionality [8, 9], and (iv) very recent work on worst case lower bounds [10]. ([11] provides a survey on methods of complexity analysis of term rewrite systems.)

Email addresses: georg.moser@uibk.ac.at (Georg Moser), michael.schaper@uibk.ac.at (Michael Schaper)

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

A natural idea is to exploit existing complexity analysers in the context of (fully automated) resource analysis of programs. Successful examples being for example the development of complexity analysis tools for logic or (higher-order) functional programs [12, 13]. In this paper we show how the runtime complexity of imperative programs can be analysed fully automatically by a transformation to term rewrite systems, the complexity of which can then be automatically verified by existing complexity tools. More precisely, we study complexity preservation and reflection of transformations from *Jinja bytecode (JBC) programs* to *constraint term rewrite systems*. Jinja is a Java-like language that exhibits the core features of Java [14]. Its semantics is clearly defined and machine checked in the theorem prover Isabelle/HOL [15]. JBC programs run on the Jinja virtual machine (JVM). In our analysis we focus on non-recursive programs. The analysis can handle programs with cyclic data only if the termination behaviour is independent thereof. We summarise the main contributions of this paper.

- We exploit a *term-based* abstraction of JBC programs within the abstract interpretation framework [16] (Lemma 11). The proposed transformation encompasses two stages.
- In the first stage we employ our idea of term-based abstraction to obtain a novel representation of abstractions of JVM states (Definition 11). We perform a combined control flow and data flow analysis by *symbolically evaluating* JVM states, thus obtaining a finite representation of all execution paths of a JBC program P through a graph, the *computation graph* (Theorem 2).
- In the second stage we encode the (finite) computation graph as *constraint term rewrite system*. These rewrite systems allow the formulation of conditions C over a theory T, such that a rule can only be used if the condition C is satisfied in T.
- We prove that the transformation of P to the constraint rewrite system $\mathcal{R}$ is *complexity reflecting*, that is, the runtime complexity function with respect to P is bounded by the runtime complexity function with respect to $\mathcal{R}$ (Corollary 3). Moreover, we obtain that the proposed transformation is also *non-termination preserving* (Corollary 4). We emphasise that our notion of complexity reflecting abstraction is carefully crafted such that we obtain a constant-factor size overhead in the simulation of the bytecode relation as a rewrite relation.
- Finally, we establish automatability of the transformation by providing a prototype implementation. Our prototype is already capable of yielding automated resource analysis of challenging examples from the literature.

We emphasise that our approach is *total* (Corollary 1). That is, the transformation can be applied to any well-formed, non-recursive JBC program and the computation graph is guaranteed to be finite. The constraints in the obtained rewrite systems are employed to express relations on program variables. In our actual use, we fix the underlying theory to Presburger arithmetic. However, the approach extends to any decidable theory that allows reasoning over the program states. With respect to automation, we show how to combine our proposed term-based abstraction with external shape analyses, as presented for example in [17–19].

Quite principally the established transformation allows for the direct use of rewriting based runtime complexity analysis for the resource analysis of JBC programs. However, currently existing tools for complexity analysis do not (yet) extend to *constraint* term rewrite systems. In order to test the effectivity of the approach, our prototype implements techniques for assessing the runtime complexity of constraint term rewrite systems. Furthermore we present (complexity reflecting) transformations from constraint rewrite systems to (standard) term rewrite systems and integer transitions systems. The systems obtained by these transformations thus can make use of well-known techniques from the literature and existing tools. The prototype is integrated into the general framework of T$_\mathsf{C}$T [20, 21].[2]

[2] `http://cl-informatik.uibk.ac.at/software/tct`

Structure. This paper is structured as follows. In Sections 2 and 3 we fix some basic notions to be used in the sequel as well as the definition of complexity reflecting abstractions. Furthermore, we give an overview over the Jinja programming language. Our notion of abstract states is presented in Section 4. We prove correctness of the abstraction in Section 5, while computation graphs are proposed in Section 6. Section 7 introduces constraint term rewrite systems, fixes the studied theory to Presburger arithmetic, and presents the transformation from computation graphs to rewrite systems. In Section 8 we show how to automate the transformation and give insights about the prototype implementation. Related work is presented in Section 9. Finally, in Section 10 we conclude.

2. Preliminaries

We assume basic familiarity with term rewriting and the Java programming language. In what follows we fix some basic notations as well as the definition of complexity reflecting abstractions.

Let f be a mapping from A to B, denoted $f : A \to B$, then $\mathsf{dom}(f) = \{x \mid f(x) \in B\}$ and $\mathsf{rg}(f) = \{f(x) \mid x \in A\}$. Let $a \in \mathsf{dom}(f)$. We define:

$$f\{a \mapsto v\}(x) := \begin{cases} v & \text{if } x = a \\ f(x) & \text{otherwise .} \end{cases}$$

We compare partial functions with *Kleene equality*: Two partial functions $f\colon \mathbb{N} \to \mathbb{N}$ and $g\colon \mathbb{N} \to \mathbb{N}$ are equal, denoted $f =_{\mathsf{k}} g$, if for all $n \in \mathbb{N}$ either $f(n)$ and $g(n)$ are defined and $f(n) = g(n)$ or $f(n)$ and $g(n)$ are not defined. We usually use square brackets to denote a list. Further, $(::)$ denotes the cons operator.

A *directed graph* $G = (V_G, Succ_G, L_G)$ over the set $\mathcal{L}$ of *labels* is a structure such that V_G is a finite set, the *nodes* or *vertices*, $Succ_G\colon V_G \to V_G^*$ is a mapping that associates a node u with an (ordered) sequence of nodes, called the *successors* of u. Note that the sequence of successors of u may be empty: $Succ_G(u) = []$. Finally $L_G\colon V_G \to \mathcal{L}$ is a mapping that associates each node u with its *label* $L_G(u)$. Let u, v be nodes in G such that $v \in Succ_G(u)$, then there is an *edge* from u to v in G; the edge from u to v is denoted as $u \to v$.

A structure $G = (V_G, Succ_G, L_G, E_G)$ is called *directed graph with edge labels* if $(V_G, Succ_G, L_G)$ is a directed graph over the set $\mathcal{L}$ and $E_G\colon V_G \times V_G \to \mathcal{L}$ is a mapping that associates each edge e with its *label* $E_G(e)$. Edges in G are denoted as $u \xrightarrow{\ell} v$, where $E_G(u \to v) = \ell$ and $u, v \in V_G$. We often write $u \to v$ if the label is either not important or is clear from the context.

If not mentioned otherwise, in the following a *graph* is a directed graph with edge labels. Usually nodes in a graph are denoted by $u, v, \ldots$ possibly followed by subscripts. We drop the reference to the graph G from V_G, $Succ_G$, L_G and E_G, ie. we write $G = (V, Succ, L, E)$ if no confusion can arise from this. Further, we also write $u \in G$ instead of $u \in V$.

Let $G = (V, Succ, L, E)$ be a graph and let $u \in G$. Consider $Succ(u) = [u_1, \ldots, u_k]$. We call u_i $(1 \leqslant i \leqslant k)$ the *i-th successor* of u (denoted as $u \rightharpoonup_G^i u_i$). If $u \rightharpoonup_G^i v$ for some i, then we simply write $u \rightharpoonup_G v$. A node v is called *reachable* from u if $u \rightharpoonup_G^* v$, where $\rightharpoonup_G^*$ denotes the reflexive and transitive closure of $\rightharpoonup_G$. We write $\rightharpoonup_G^+$ for $\rightharpoonup_G \circ \rightharpoonup_G^*$. A graph G is *acyclic* if $u \rightharpoonup_G^+ v$ implies $u \neq v$.

Let A be a set and $\to \subseteq A \times A$ be a binary relation on A. We write $a_1 \to a_2$ instead of $(a_1, a_2) \in \to$. The maximal *derivation height* is defined as usual: $\mathsf{dh}(a, \to) := \max\{n \mid \exists a'.a \to^n a'\}$. Let $|\cdot|\colon A \to \mathbb{N}$ denote a suitable *size measure* for A and $S \subseteq A$. We define the *complexity function* as the maximal derivation height with respect to the size of the starting elements S: $\mathsf{cp}(n, S, |\cdot|, \to) := \max\{\mathsf{dh}(s, \to) \mid s \in S \text{ and } |s| \leqslant n\}$.

Definition 1. Let A be a set and $\to \subseteq A \times A$ be a binary relation on A. Let $\to|_S \subseteq A \times A$ denote the restriction of $\to$ to elements reachable from $S \subseteq A$. Let B be another set, $\rightsquigarrow \subseteq B \times B$ and $T \subseteq B$. Moreover, let $\gg \subseteq B \times A$ be a relation. We say $\rightsquigarrow|_T$ *abstracts* $\to|_S$ (with respect to $\gg$), if $\gg \circ \to|_S \subseteq \rightsquigarrow|_T \circ \gg$ and for all $s \in S$ there exists $t \in T$ such that $t \gg s$. We call $\gg$ the *abstraction* of $\to|_S$ to $\rightsquigarrow|_T$. Furthermore, $\gg$ is a *complexity reflecting abstraction* if for all $t \gg s$ with $s \in S$ and $t \in T$ we have $\|t\| \in O(|s|)$. Here $|\cdot|$ and $\|\cdot\|$ are suitable size measures for S and T, respectively.

The intuition being that $\gg$ links elements of the considered relations and $\gg \circ \to|_S \subseteq \rightsquigarrow|_T \circ \gg$ represents a simulation of $\to|_S$ in $\rightsquigarrow|_T$. In the case of a complexity reflecting abstraction we additionally employ a size constraint on the initial elements T of the abstraction. Figure 1 visually represents the abstraction relation.

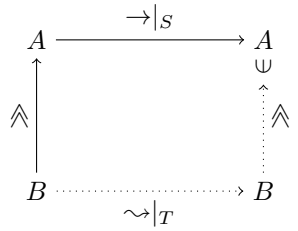


Figure 1: The commutative diagram for $\gg \circ \to|_S \subseteq \rightsquigarrow|_T \circ \gg$.

3. Jinja Bytecode

In this section, we give an overview over the Jinja programming language [15] and inspect the internal state of the Jinja virtual machine (JVM). Jinja has a clearly defined semantics that consists only of a small but expressive set of instructions and provides a formal notion of well-formedness. By design it exhibits the core features from the Java bytecode language and the Java virtual machine and incorporates most of the well-known features such as *class inheritance* and *dynamic dispatch*. On the other hand [15] also focuses on tractability and does not support some of the more advanced features such as *threading* and *reflection*, which are also out of scope of our work. Therefore, the Jinja bytecode language is an ideal choice for the formalisation of our abstraction and our prototype implementation.

Definition 2. A *Jinja value* can be a Boolean of type `bool`, an (unbounded) integer of type `int`, the dummy value `unit` of type `void`, the null reference `null` of type `nullable`, or a non-null reference (aka address).

The dummy value `unit` is used for the evaluation of assignments (see [15]) and also used in the JVM to allocate uninitialised local variables. The actual type of addresses is not important and we usually identify the type of an address with the type of the object bound to the address.

Example 1. Figure 2 depicts a program defining the `List` class with the `append` method. Deviating from the notation employed by Klein and Nipkow in [15], we present Jinja source code in a Java-like syntax.

```
class List{
  List next;
  int val;

  void append(List ys){
    List cur = this;
    while(cur.next != null){
      cur = cur.next
    }
    cur.next = ys;
  }
}
```

Figure 2: The `List` class with the `append` method.

In preparation for the sequent sections, we reflect the structure and properties of bytecode programs and the virtual machine.

Definition 3. A *JBC program* P consists of a set of *class declarations*. Each class is identified by a *class name* and further consists of the name of its direct *superclass*, *field declarations* and *method declarations*. The superclass declaration is non-empty, except for a dedicated class termed *Object*. Moreover, the subclass hierarchy of P is tree-shaped. We denote the strict preorder induced by the subclass hierarchy of P with $\prec$ and its reflexive closure is denoted $\preceq$. A field declaration is a pair of *field name* and *field type*. A method declaration consists of the *method name*, a list of *parameter types*, the *result type* and the *method body*. A method body is a triple of $(mxs \times mxl \times instructionlist)$, where mxs and mxl are natural numbers denoting the maximum size of the operand stack and the number of local variables, not including the `this` reference and the parameters of the method, while *instructionlist* gives a sequence of bytecode instructions. The `this` reference can be conceived as a hidden parameter and references the object that invokes the method.

The formalisation of Jinja in [15] covers for practical reasons only a minimal set of stack operations on Booleans and integers, namely addition and equality. We extend the original set of instructions by some standard operations on values, taking ideas from Jinja with Threads into account [22, 23]. The set of instructions is listed in Figure 3. We employ following conventions: Let n denote a natural number, i an integer, v a Jinja value, cn a class name, fn a field name, and mn a method name.

$$
\begin{aligned}
\texttt{Ins} := \quad & \texttt{Load}\ n \mid \texttt{Store}\ n \mid \texttt{Push}\ v \mid \texttt{Pop} \\
& \mid \texttt{IAdd} \mid \texttt{ISub} \mid \texttt{ICmpGt} \mid \texttt{CmpEq} \mid \texttt{CmpNeq} \mid \texttt{BAnd} \mid \texttt{BOr} \mid \texttt{BNot} \\
& \mid \texttt{Goto}\ i \mid \texttt{IfFalse}\ n \mid \\
& \mid \texttt{New}\ cn \mid \texttt{Checkcast}\ cn \mid \texttt{Getfield}\ cn\ fn \mid \texttt{Putfield}\ cn\ fn \mid \\
& \mid \texttt{Invoke}\ mn\ n \mid \texttt{Return}
\end{aligned}
$$

Figure 3: The Jinja bytecode instruction set.

Definition 4. A *(JVM) state* is a pair consisting of the *heap* and a list of *frames*. A *heap* is a mapping from *addresses* to *objects*, where an object is a pair $(cn, ftable)$ such that:

- cn denotes the *class name*, and
- $ftable$ denotes the fieldtable, ie. a mapping from (cn', fn) to values; here cn' denotes the class in which the field fn is declared and is a (not necessarily proper) superclass of cn.

A *frame* represents the environment of a method and is a quintuple (stk, loc, cn, mn, pc), such that:

- stk denotes the *operation stack*, ie. an array of values,
- loc denotes the *registers*, ie. an array of values,
- cn denotes the *class name*,
- mn denotes the *method name*, and
- pc is the *program counter*.

The *program location* of a state $(heap, frames)$ is a list of triples (cn, mn, pc) obtained by restricting the elements of $frames$ to class name, method name and program counter.

Let stk (loc) denote the operation stack (registers) of a given frame. Typically the structure of loc is as follows: the 0^{th} register holds the `this`-pointer, followed by the parameters and the local variables of the method. Uninitialised registers are preallocated with the dummy value `unit`. We denote the entries of stk (loc), by $stk(i)$ ($loc(i)$) for $i \in \mathbb{N}$ and write $\mathsf{dom}(stk)$ ($\mathsf{dom}(loc)$) for the set of indices of the array stk (loc). The collection of all stack (register) indices of a state is denoted Stk (Loc), that is, $(stk, i, j) \in Stk$ if $stk_i(j)$ denotes the j^{th} value in the operation stack of the i^{th} frame in $frames$. Similarly for $(loc, i', j') \in Loc$. Often there is no need to separate between the local variables of a Jinja program and the registers in a JBC

program. Hence we use registers and local variables interchangeably. The domain of the fieldtable for a given object of class cn contains all fields declared for cn together with all fields declared for superclasses of cn. Thus the domain of the fieldtable is equal for any instance of class cn.

We summarise the semantics of the bytecode instructions. For details see [15]. Most instructions affect only the current frame. `Load` n pushes the value of $loc(n)$ onto the stack, whereas `Store` n pops the top value of the operand stack and stores it at $loc(n)$.

$$\texttt{Load}\ n \quad \frac{(heap, (stk, loc, cn, mn, pc) :: frames)}{(heap, (loc(n) :: stk, loc, cn, mn, pc + 1) :: frames)}$$

$$\texttt{Store}\ n \quad \frac{(heap, (v :: stk, loc, cn, mn, pc) :: frames)}{(heap, (stk, loc\{n \mapsto v\}, cn, mn, pc + 1) :: frames)}$$

`Push` v pushes the value v onto the stack, whereas `Pop` removes the top element of the stack.

$$\texttt{Push}\ v \quad \frac{(heap, (stk, loc, cn, mn, pc) :: frames)}{(heap, (v :: stk, loc, cn, mn, pc + 1) :: frames)}$$

$$\texttt{Pop} \quad \frac{(heap, (v :: stk, loc, cn, mn, pc) :: frames)}{(heap, (stk, loc, cn, mn, pc + 1) :: frames)}$$

`IAdd`, `ISub`, `ICmpGt`, `BAnd`, `BOr`, and `BNot` denote the usual operations on integer and Boolean values. `CmpEq` and `CmpNeq` define equality and inequality on all values. Operands are taken from the top of the stack and the result is pushed onto the stack. We use `BinOp` together with $\otimes = \{+, -, \vee, \wedge, \geqslant, =, \neq\}$ to define the instructions `IAdd`, `ISub`, `BOr`, `BAnd`, `ICmpGt`, `CmpEq` and `CmpNeq`.

$$\texttt{BinOp} \quad \frac{(heap, (v_1 :: v_2 :: stk, loc, cn, mn, pc) :: frames)}{(heap, (v_2 \otimes v_1 :: stk, loc, cn, mn, pc + 1) :: frames)}$$

$$\texttt{BNot} \quad \frac{(heap, (b :: stk, loc, cn, mn, pc) :: frames)}{(heap, (\neg b :: stk, loc, cn, mn, pc + 1) :: frames)}$$

`Goto` i defines an unconditional (relative) jump. `IfFalse` n defines a conditional control flow instruction that depends on the Boolean value on top of the stack.

$$\texttt{Goto}\ i \quad \frac{(heap, (stk, loc, cn, mn, pc) :: frames)}{(heap, (stk, loc, cn, mn, pc + i) :: frames)}$$

$$\texttt{IfFalse}\ i \quad \frac{(heap, (\mathsf{false} :: stk, loc, cn, mn, pc) :: frames)}{(heap, (stk, loc, cn, mn, pc + i) :: frames)}$$

$$\texttt{IfFalse}\ i \quad \frac{(heap, (\mathsf{true} :: stk, loc, cn, mn, pc) :: frames)}{(heap, (stk, loc, cn, mn, pc + 1) :: frames)}$$

`New` dn allocates a new object $(dn, ftable)$ and pushes the corresponding (fresh) address a onto the stack. Each field of the freshly created instance is instantiated with the default value. That is `0` for integer typed fields, `false` for Boolean typed fields, and `null` otherwise. The instruction `Checkcast` dn checks the type of the object mapped by the address on top of the stack and it fails if the type is not a subclass of dn, ie. it fails if $dn' \preceq dn$ does not hold. Note that `Checkcast` has no effect on the state but is used for the static typing during compilation. Field access and field update of an object do not depend on the runtime type of the object. In particular, the information which field is accessed or updated, is already encoded in the bytecode program. `Getfield` $dn\ fn$ dereferences the address on top of the stack and replaces the address by the content of the field identified by (dn, fn) onto the stack. `Putfield` $dn\ fn$ dereferences the reference next to the top of the stack and updates the content of the field identified by (dn, fn) using the value on top

of the stack. Here dn' is the runtime type of the accessed object. By well-formedness we have $dn' \preceq dn$.

$$\texttt{New}\ dn \quad \frac{(heap\{a \mapsto \bot\}, (stk, loc, cn, mn, pc) :: frames)}{(heap\{a \mapsto (dn, ftable)\}, (a :: stk, loc, cn, mn, pc+1) :: frames)}$$

$$\texttt{Checkcast}\ dn \quad \frac{(heap\{a \mapsto (dn', ftable)\}, (a :: stk, loc, cn, mn, pc) :: frames)}{(heap\{a \mapsto (dn', ftable)\}, (a :: stk, loc, cn, mn, pc+1) :: frames)}$$

$$\texttt{Getfield}\ dn\ fn \quad \frac{(heap\{a \mapsto (dn', ftable)\}, (a :: stk, loc, cn, mn, pc) :: frames)}{(heap\{a \mapsto (dn', ftable)\}, (ftable(dn, fn) :: stk, loc, cn, mn, pc+1) :: frames)}$$

$$\texttt{Putfield}\ dn\ fn \quad \frac{(heap\{a \mapsto (dn', ftable)\}, (v :: a :: stk, loc, cn, mn, pc) :: frames)}{(heap\{a \mapsto (dn', ftable\{(dn, fn) \mapsto v\})\}, (stk, loc, cn, mn, pc+1) :: frames)}$$

The instructions `Invoke` and `Return` manipulate the frame stack. The instruction `Invoke` $mn\ n$ first copies the top n elements of the stack in reversed order. It then identifies the method to invoke by the runtime type of the corresponding reference and constructs a new frame including `this`, the parameters, and the local variables us initialised with `unit`. The program counter of the new frame is set to 0. Here dn is the class where the method mn' is defined, which corresponds either to the class of the accessed object or a superclass of it and depends wether the method mn' is overridden or inherited in dn'. The instruction `Return` pops the top frame and updates the stack of the next frame with the return value; if the frame stack consists only of a single frame the program terminates.

$$\texttt{Invoke}\ mn'\ n \quad \frac{(heap\{a \mapsto dn'\}, (p_{n-1} :: \cdots :: p_0 :: a :: stk, loc, cn, mn, pc) :: frames)}{(heap\{a \mapsto dn'\}, ([], a :: p_0 :: \cdots :: p_{n-1} :: us, dn, mn', 0) :: (stk, loc, cn, mn, pc) :: frames)}$$

$$\texttt{Return} \quad \frac{(heap, (v :: stk, loc, cn, mn, pc) :: (stk', loc', cn', mn', pc') :: frames)}{(heap, (v :: stk', loc', cn', mn', pc'+1) :: frames)}$$

$$\texttt{Return} \quad \frac{(heap, [frame])}{(heap, [])}$$

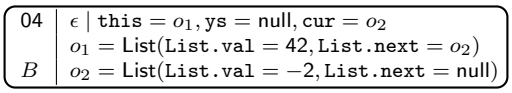
All instructions beside the control flow instructions and method invocation increment the program counter by one. The instructions `Getfield`, `Putfield`, `Checkcast` and `Invoke` dereference references and may fail if the reference is `null`, further the type check of `Checkcast` may fail. In this case we assume the computation aborts immediately.

Example 2. Consider the `List` class from Example 1. Figure 4 depicts the corresponding bytecode program, resulting from the compilation rules in [15]. In Jinja assignments evaluate to `unit`. That is why assignments are followed by `Push unit` and `Pop`. The compilation of the `while` statement uses a conditional jump instruction to inspect the result of the condition that is evaluated on the stack and an unconditional jump instruction to re-evaluate the condition after processing the body of the loop. The naming convention for the registers depicted in the figure is kept throughout the paper.

Example 3. We consider an evaluation of the `append` method and inspect the state after the assignment `cur = this`. We assume that `this` is initially an acyclic list of length two, the fields of the list store some randomly chosen integer values, and `ys` is `null`. We represent the state labelled with A as follows:

04	$\epsilon \mid \texttt{this} = o_1, \texttt{ys} = \mathsf{null}, \texttt{cur} = o_1$
	$o_1 = \mathsf{List}(\texttt{List.val} = 42, \texttt{List.next} = o_2)$
A	$o_2 = \mathsf{List}(\texttt{List.val} = -2, \texttt{List.next} = \mathsf{null})$

The program counter is `04`. The operation stack is empty, which is denoted by ϵ. The registers `this` and `cur` store the same address, namely o_1, and register `ys` stores `null`. The domain of the heap is $\{o_1, o_2\}$. Address o_1 is mapped to a `List` object. The domain of the fieldtable of the object is $\{(\texttt{List}, \texttt{val}), (\texttt{List}, \texttt{next})\}$ whose elements are mapped to the values `42` and o_2. State B represents the situation after one iteration of the loop. In particular `cur` now points to o_2.



In the following the registers 0,1, and 2 are named as `this`, `ys`, and `cur` respectively.

```
Class:
 Name: List                Bytecode:
 Classbody:                 00: Load 0
  Superclass: Object        01: Store 2             \\ cur = this;
  Fields:                   02: Push unit
   List next                03: Pop
   int val                  04: Load 2              \\ load condition
  Methods:                  05: Getfield List next
   Method: void append      06: Push null
    Parameters:             07: CmpNeq              \\ cur.ext != null;
     List ys                08: IfFalse 7           \\ begin:while
    Methodbody:             09: Load 2
     MaxStack:              10: Getfield List next
      2                     11: Store 2             \\ cur = cur.next;
     MaxVars:               12: Push unit
      1                     13: Pop
                            14: Goto -10            \\ end:while
                            15: Push unit
                            16: Pop
                            17: Load 2
                            18: Load 1
                            19: PutField List next \\ cur.next = ys
                            20: Push unit
                            21: Return
```

Figure 4: The bytecode for the `List` class.

Definition 5. We extend the subclass relation to a partial order on types, denoted $\leqslant_{\mathsf{type}}$. The types of P consists of $\{\texttt{bool}, \texttt{int}, \texttt{void}, \texttt{nullable}\}$ together with all classes cn defined in P. We use $\mathsf{type}(v)$ to denote the type of value v and $\mathsf{types}(P)$ to denote the collection of types in P. Moreover, we identify the type of an address with the type of the object bound to the address. Let t, t', cn, cn' be types in P. Then $t \leqslant_{\mathsf{type}} t'$ holds if $t = t'$ or

- $t = \texttt{void}$,
- $t = \texttt{nullable}$ and $t' = cn$,
- $t = cn$, $t' = cn'$ and $cn \preceq cn'$.

The *least common superclass* is the least upper bound for a set of classes $CN \subseteq \mathsf{types}(P)$ and is always defined in $\mathsf{types}(P)$ since the subclass hierarchy of a well-formed program is tree-shaped.

The bytecode verifier established in [15] ensures following properties: Bytecode instructions are provided with arguments of the expected type. No instruction tries to get a value from the empty stack, nor puts more elements on the stack or accesses more elements than specified. The program counter is always within the code array of the method. All registers, except the register storing `this`, must be written into before accessed. If two states have the same program location, then for all frames the domain $\mathsf{dom}(stk)$ ($\mathsf{dom}(loc)$) coincides for the two states. Moreover, corresponding entries of stk (loc) of the two states are well-typed in the sense that $v \leqslant_{\mathsf{type}} v'$ or $v' \leqslant_{\mathsf{type}} v$ holds for the two corresponding entries.

The compiler presented in [15] transforms a well-formed Jinja program into a well-formed JBC program. A JBC program that passes the bytecode verification is again called *well-formed*. While the set of instructions used here are a (slight) extension of the minimalistic set considered in [15], this notion of well-formedness is still applicable, as all considered extensions are present in Jinja with Threads [22, 23].

In the following we consider Jinja programs and JBC programs to be well-formed. To ease readability we do not consider exception handling, that is, an exception yields immediate termination of the program. This is not a restriction of our analysis, as it could be easily integrated, but complicates matters without gaining additional insight.

While Example 3 shows a succinct presentation of states, it is more natural to conceive the heap (and conclusively the state) as a graph. We omit the technical definition here (see [24] for further details) but provide the general idea: Let s be a state $(heap, frames)$. We define the *state graph* of s as $S = (V_S, Succ_S, L_S, E_S)$. For all non-address values of s we define a unique *implicit reference*. The idea is that sharing is only induced via references but not implicit references. The nodes of S consists of all stack (register) indices, the references in *heap* and the implicit references of s. The successors of a node indicate the values bound to stack (register) indices and the fields of instances in *heap*, and is an implicit reference if a non-address value is bound and a reference otherwise. The label of a node is either a stack (register) index, the type of an instance $heap(u)$ or a non-address value. The label of an edge indicates the fields (cn, id) for instances $heap(u)$, and is empty otherwise. In presenting state graphs, we indicate references, but do not depict implicit references. Furthermore, we use representative names for stack (register) indices.

Example 4. Recall the state representation A and B in Example 3. Figure 5 and Figure 6 depict the corresponding state graphs.

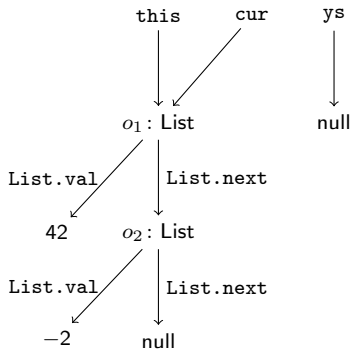


Figure 5: Representation of state A as state graph.

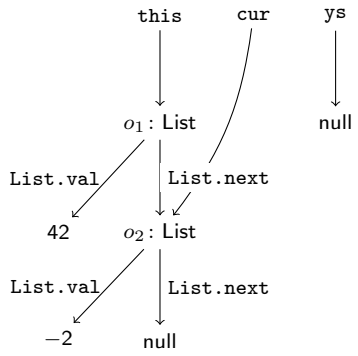


Figure 6: Representation of state B as state graph.

The *size* of a state is defined on a *per-reference* basis that unravels sharing. Thus, this size measure sums over all reachable locations on the heap. Similar notions are employed in the automated resource analysis of Java [25] or the automated amortised resource analysis (see e.g. [26]).

Definition 6. Let s be a state and let S be its state graph. Let u, v be nodes in S and $u \xrightarrow{+}_S v$ denote a simple path in S from u to v. Then the size of a stack or register index u, denoted as $|u|$, is defined as

$$|u| := \sum_{u \xrightarrow{+}_S v} |L_S(v)| \,,$$

where $|L_S(v)|$ is $\mathsf{abs}(L_S(v))$ if $L_S(v) \in \mathbb{Z}$, otherwise 1. Here, $\mathsf{abs}(z)$ denotes the absolute value of the integer z. The *size* of s is the sum of all sizes of stack or register indices in S. In the following we use $|s|$ to denote the size of a state s.

In order to provide a complexity reflecting abstraction we have to relate the size of the initial elements of the term-based abstraction to the size of the initial JVM states. This relation has to be linear, which

is a direct consequence of Definition 6. An execution of a program on the JVM starts from a dedicated entry point (usually the `main` function) with the heap of the initial state being empty. We generalise this behaviour to analyse programs like `append` where the initial heap is non-empty. Though we restrict the data on the heap to be *tree-shaped*, that is objects in the heap do not share. We emphasise that this restriction to tree-shaped initial heaps is embodied in the above notion of state size.

Example 5. We compute the size of state A as depicted in Figure 5. The size of A is obtained by collecting all labels reachable from each register (stack) index:

$$\begin{aligned}|A| =& |\texttt{this}| + |\texttt{cur}| + |\texttt{ys}| \\ =& (|\mathsf{List}| + |42| + |\mathsf{List}| + |{-2}| + |\mathsf{null}|) + (|\mathsf{List}| + |42| + |\mathsf{List}| + |{-2}| + |\mathsf{null}|) + |\mathsf{null}| \\ =& (1 + \mathsf{abs}(42) + 1 + \mathsf{abs}(-2) + 1) + (1 + \mathsf{abs}(42) + 1 + \mathsf{abs}(-2) + 1) + 1 = 95\ .\end{aligned}$$

Let P be a program and $\mathcal{JS}$ denote the set of JVM states of P. Moreover, let $s, t \in \mathcal{JS}$. We denote by $s \to_P t$ the one-step transition relation of the JVM. The reflexive and transitive closure is denoted $s \to_P^* t$. The complete lattice $\mathcal{P}(\mathcal{JS}) := (\mathcal{P}(\mathcal{JS}), \subseteq, \cup, \cap, \varnothing, \mathcal{JS})$ defines the *concrete computation domain*. Let $\mathcal{S} \subseteq \mathcal{JS}$. We define the *collecting semantics* on the domain $\mathcal{P}(\mathcal{JS})$ as the component-wise extension of the one-step transition relation to sets: $\{t \mid s \to_P t, s \in \mathcal{S}\}$.

Definition 7. Let $\mathcal{JS}$ denote the set of JVM states of P, and let $\mathcal{S} \subseteq \mathcal{JS}$ denote the set of initial states. We define the *runtime complexity* with respect to P as follows:

$$\mathsf{rcjvm}(n) =_{\mathsf{k}} \mathsf{cp}(n, \mathcal{S}, |\cdot|, \to_P)$$

Note that we adopt a (standard) unit cost model for system calls.

4. Abstract States

In this section, we introduce *abstract states* as generalisations of JVM states. The intuition being that abstract states represent sets of states in the JVM. The idea of abstracting JVM states in this way is due to Otto et al. [27]. However, our presentation technically differs from [27] (and also from follow-up work in the literature) as we employ an implicit representation of sharing that makes use of graph morphisms taking ideas from term graph rewriting [28, Chapter 13] into account, rather than the explicit sharing information proposed in [27, 29–31]. Furthermore, abstract states as defined below are a straightforward generalisation of JVM states as defined in [15]. This circumvents an additional transformation step as presented in [29]. In what follows we usually use superscript $\natural$ to denote abstract states.

Definition 8. We extend Jinja expressions by countable many abstract variables $X_1, X_2, X_3, \ldots$, denoted by x, y, z, ... An *abstract variable* may either abstract an object, an integer or a Boolean value.

In denoting abstract variables typically the name is of less importance than the type, that is we denote an abstract variable for an object of class cn, simply as cn, while abstract integer or Boolean variables are denoted as int, and $bool$, respectively. For brevity we sometimes refer to an abstract variable of integer or Boolean type, as *abstract integer* or *abstract Boolean*, respectively.

Definition 9. An *abstract value* is either a Jinja value (cf. Definition 2), or an abstract Boolean or integer. In turn a Jinja value is also called a *concrete value*.

Note that, as in the JVM, only (abstract) objects can be shared. In particular abstract variables for objects are only referenced via the heap. The next definition abstracts the heap of a JVM through the use of abstract variables and values.

Definition 10. An *abstract heap* is a mapping from *addresses* to *abstract objects*, where an abstract object is either a pair $(cn, ftable)$ or an abstract variable. *Abstract frames* are defined like frames of the JVM, but registers and operand stack of an abstract frame store abstract values.

We define (partial) projection functions cl and ft as follows:

$$\mathsf{cl}(obj) := \begin{cases} cn & \text{if } obj \text{ is an object and } obj = (cn, ftable) \\ cn & \text{if } obj \text{ is an abstract variable of type } cn \end{cases}$$

$$\mathsf{ft}(obj) := \begin{cases} ftable & \text{if } obj \text{ is an object and } obj = (cn, ftable) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Furthermore, we define *annotations* of addresses in an abstract state, denoted as iu. Formally, annotations represent pairs $p \neq q$ of addresses, where $p, q \in heap$ and p is not q. Annotations are used to perform refinements on abstract states and approximate aliasing. We clarify the precise use of annotations when providing the semantics of the abstract domain.

Definition 11. An *abstract state* $s^\sharp$ is a triple $(heap, frames, iu)$ consisting of an abstract heap $heap$, a list of abstract frames $frames$, and a set of annotations iu. We demand that all addresses in $heap$ are reachable from local variables or stack entries in the list of frames $frames$.

Due to the presence of abstract variables, abstract states can represent sets of states as the variables can be suitably instantiated. The annotation $p \neq q \in iu$ will be used to disallow aliasing of addresses in JVM states represented by the abstract state. Different JVM states can be abstracted to a single abstract state. To make this intuition precise, we will formally define the abstract domain, denoted as $\mathcal{AS}$, and extend it to a complete lattice $(\mathcal{AS}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ and show a Galois insertion between $\mathcal{P}(\mathcal{JS})$ and $\mathcal{AS}$.

When presenting abstract states we follow the presentation of concrete states, though registers and object fields may store integer and Boolean variables and addresses may point to abstract variables. Furthermore, we do not depict the set of annotations to ease readability. When necessary, we provide supplementary information.

Example 6. Reconsider Example 3, which illustrates state A from an example run of `append`. Now consider the abstract state $A^\sharp$.

04	$\epsilon \mid \texttt{this} = o_1, \texttt{ys} = o_2, \texttt{cur} = o_1$
	$o_1 = \mathsf{List}(\texttt{List.val} = int, \texttt{List.next} = o_3)$
$A^\sharp$	$o_2 = list, o_3 = list$

Informally, $A^\sharp$ is an abstraction of A, ie. A is an element of the set of states represented by $A^\sharp$. In particular $A^\sharp$ forms an abstraction of any JVM state obtained at instruction 04 in the `append` program (if `this` initially references a non-empty list) before any iteration of the `while`-loop. Furthermore, consider the following abstract state $B^\sharp$, which is an abstraction of B and represents any JVM state obtained if exactly one iteration of the loop has been performed.

04	$\epsilon \mid \texttt{this} = o_1, \texttt{ys} = o_2, \texttt{cur} = o_3$
	$o_1 = \mathsf{List}(\texttt{List.val} = int, \texttt{List.next} = o_3)$
	$o_2 = list, o_4 = list$
$B^\sharp$	$o_3 = \mathsf{List}(\texttt{List.val} = int, \texttt{List.next} = o_4)$

Definition 12. We define a preorder on abstract values, which are not references and abstract objects. We extend $\mathsf{type}(v)$ (cf. Definition 5) to abstract values the intended way, ie. $\mathsf{type}(int) = \texttt{int}$, $\mathsf{type}(bool) = \texttt{bool}$ and $\mathsf{type}(cn) = cn$ for an integer variable int, a Boolean variable $bool$, and class variable cn, respectively. The preorder $\trianglelefteq$ is defined as follows: We have $v \trianglelefteq w$, if either

1. $v = w$, or
2. $\mathsf{type}(v) \leqslant_{\mathsf{type}} \mathsf{type}(w)$ and w is an abstract variable.

We write $w \trianglerighteq v$, if $v \trianglelefteq w$.

Let $|stk|$, $|loc|$ denote the maximum size of the operand stack and the number of variables respectively. We make use of the following abbreviation: $w \trianglerighteq_m v$ if either $w \trianglerighteq v$ or v, w are references and we have $v = m(w)$, where m denotes a mapping on references.

Definition 13. Let $s^\sharp$ be an abstract state $(heap, frames, iu)$ and $t^\sharp$ be an abstract state $(heap', frames', iu')$. Furthermore, let $frames = [frame_1, \ldots, frame_k]$ and $frame_i = (stk_i, loc_i, cn_i, mn_i, pc_i)$, and let $frames' = [frame'_1, \ldots, frame'_k]$ and $frame'_i = (stk'_i, loc'_i, cn'_i, mn'_i, pc'_i)$. Then $s^\sharp$ is an *abstraction* of $t^\sharp$ (denoted as $s^\sharp \sqsupseteq t^\sharp$), if the following conditions hold:

1. for all $1 \leqslant i \leqslant k$: $pc_i = pc'_i$, $cn_i = cn'_i$, and $mn_i = mn'_i$,
2. for all $1 \leqslant i \leqslant k$: $\mathsf{dom}(stk_i) = \mathsf{dom}(stk'_i)$ and $\mathsf{dom}(loc_i) = \mathsf{dom}(loc'_i)$, and
3. there exists a mapping $m\colon \mathsf{dom}(heap) \to \mathsf{dom}(heap')$ such that
   - for all $1 \leqslant i \leqslant k$, $1 \leqslant j \leqslant |stk_i|$: $stk_i(j) \trianglerighteq_m stk'_i(j)$,
   - for all $1 \leqslant i \leqslant k$, $1 \leqslant j \leqslant |loc_i|$: $loc_i(j) \trianglerighteq_m loc'_i(j)$,
   - for all $a \in \mathsf{dom}(heap)$: $heap(a) \trianglerighteq heap'(m(a))$,
   - for all $a \in \mathsf{dom}(heap)$, such that $\mathsf{ft}(heap(a))$ is defined and for all $1 \leqslant i \leqslant \ell$: $f(cn_i, fn_i) \trianglerighteq_m f'(cn_i, fn_i)$, where $f := \mathsf{ft}(heap(a))$ and $f' := \mathsf{ft}(heap'(m(a)))$ with $\mathsf{dom}(f) = \mathsf{dom}(f') = \{(cn_1, fn_1), \ldots, (cn_\ell, fn_\ell)\}$.
4. finally, we have $m^*(iu) \subseteq iu'$.

Here, m^* denotes the lifting of the mapping m to sets: $m(\{iu_1, \ldots, iu_k\}) = \{m(iu_1), \ldots, m(iu_k)\}$.

Alternatively, we say that $t^\sharp$ is an instance of $s^\sharp$ (denoted as $t^\sharp \sqsubseteq s^\sharp$), whenever $s^\sharp \sqsupseteq t^\sharp$ holds. We write $s^\sharp \sqsupset t^\sharp$ if $s^\sharp \sqsupseteq t^\sharp$ but not $t^\sharp \sqsupseteq s^\sharp$. The first item states that the program locations of $s^\sharp$ and $t^\sharp$ are equivalent and the second item states that the domains of the registers are equivalent. For well-formed programs item one implies item two. Item three defines the abstraction of objects and values in the heap.

Example 7. Consider the states $A^\sharp$ and $B^\sharp$ described in Example 6. For the state $S^\sharp$ depicted below we obtain that $A^\sharp \sqsubseteq S^\sharp$ and $B^\sharp \sqsubseteq S^\sharp$, ie. $S^\sharp$ forms an abstraction of both states.

04	$\epsilon \mid \texttt{this} = o_1, \texttt{ys} = o_2, \texttt{cur} = o_4$
	$o_1 = \mathsf{List}(\texttt{List.val} = int, \texttt{List.next} = o_3)$
	$o_2 = list, o_3 = list, o_5 = list$
$S^\sharp$	$o_4 = \mathsf{List}(\texttt{List.val} = int, \texttt{List.next} = o_5)$

Akin to the graph representation of JVM states we can represent an abstract state $s^\sharp = (heap, frames, iu)$ as state graph $S^\sharp_1 = (V_{S^\sharp_1}, Succ_{S^\sharp_1}, L_{S^\sharp_1}, E_{S^\sharp_1}, iu_{S^\sharp_1})$.

Example 8. Consider the abstract states $A^\sharp$, $B^\sharp$, and $S^\sharp$ presented in Examples 6 and 7. The state graphs of $A^\sharp$ and $B^\sharp$ are given in Figure 7 and Figure 8, respectively. The state graph of the abstraction $S^\sharp$ is depicted in Figure 9.

We introduce *state homomorphisms* that allow an alternative but equivalent definition of the instance relation $\sqsubseteq$. We will use standard properties of morphisms to extend $\sqsubseteq$ to a partial order.

Definition 14. Let Stk (Loc) collect all stack (register) indices (see page 5). Furthermore, let $S^\sharp$ and $T^\sharp$ be the state graphs of abstract states $s^\sharp$ and $t^\sharp$. A *state homomorphism* from $S^\sharp$ to $T^\sharp$ (denoted $m\colon S^\sharp \to T^\sharp$) is a function $m\colon V_{S^\sharp} \to V_{T^\sharp}$ such that:

1. for all $u \in S^\sharp$ and $u \in Stk \cup Loc$, $L_{S^\sharp}(u) = L_{T^\sharp}(m(u))$,
2. for all $u \in S^\sharp \setminus (Stk \cup Loc)$, $L_{S^\sharp}(u) \trianglerighteq L_{T^\sharp}(m(u))$, and
3. for all $u \in S^\sharp$: if $u \xrightarrow{\ell} v \in S^\sharp$ then $m(u) \xrightarrow{\ell'} m(v) \in T^\sharp$ and $\ell = \ell'$.

Example 9. Figure 10 illustrates the state homomorphism $m\colon S^\sharp \to B^\sharp$.

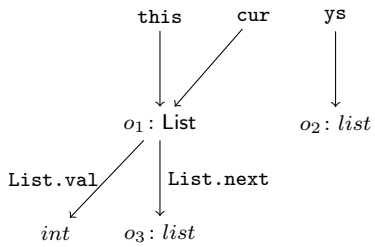


Figure 7: Abstract state $A^\natural$.

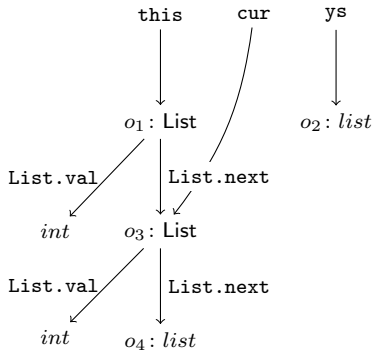


Figure 8: Abstract state $B^\natural$.

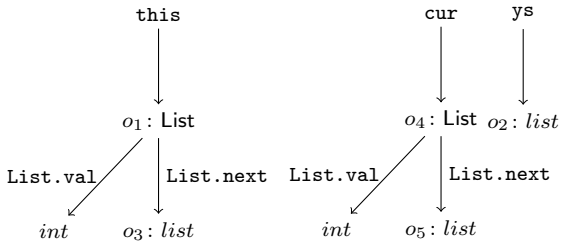


Figure 9: Abstract state $S^\natural$.

Lemma 1. *Let $s^\natural = (heap, frames, iu)$ and $t^\natural = (heap', frames', iu')$ be abstract states, and $S^\natural$ and $T^\natural$ denote their state graphs. The following notions are equivalent:*

1. *$s^\natural \sqsupseteq t^\natural$ according to Definition 13, and*
2. *$s^\natural$ and $t^\natural$ have the same program location, there exists a state homomorphism m from $S^\natural$ to $T^\natural$ and $m^*(iu) \subseteq iu'$.*

Proof. Straightforward. □

If no confusion can arise we refer to a state homomorphism simply as *morphism*. It is easy to see that the composition $m_1 \circ m_2$ of two morphisms m_1, m_2 is again a morphism. Due to the composability of morphism it follows that the instance relation $\sqsubseteq$ is transitive.

Two abstract states $s^\natural$ and $t^\natural$ are equivalent, denoted $s^\natural \sim t^\natural$, if and only if $s^\natural \sqsubseteq t^\natural$ and $t^\natural \sqsubseteq s^\natural$. This gives rise to an equivalence relation on abstract states. In what follows we identify equivalent states and leave the equivalence relation implicit. We denote the set of abstract states as $\{\top, \bot\} \subseteq \mathcal{AS}$, where $\bot$ denotes the *minimal abstract state* and $\top$ denotes the *maximal abstract state*, and extend $\sqsubseteq$ with the limit cases $\bot \sqsubseteq s^\natural$ and $s^\natural \sqsubseteq \top$ for all $s^\natural \in \mathcal{AS}$. Thus, $(\mathcal{AS}, \sqsubseteq)$ is a partial order. In order to extend the partial order $(\mathcal{AS}, \sqsubseteq)$ to a complete lattice $\mathcal{AS} := (\mathcal{AS}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ we provide a least upper bound definition of the *join* of abstract states.

Definition 15. Let $s_1^\natural$ and $s_2^\natural$ be states such that there exists an abstraction $t^\natural$ of $s_1^\natural$ and $s_2^\natural$. We call $t^\natural$ the *join* of $s_1^\natural$ and $s_2^\natural$, denoted as $s_1^\natural \sqcup s_2^\natural$, if $t^\natural$ is the least upper bound of $\{s^\natural, s_2^\natural\}$ with respect to the partial order $\sqsubseteq$.

The limit cases are handled as usual, that is, $\top$ is the absorbing element and $\bot$ is the identity element of the join operation. If the program locations of $s_1^\natural$ and $s_2^\natural$ differ, then $s_1^\natural \sqcup s_2^\natural = \top$. Otherwise, we can identify invariants to construct an upper bound $t^\natural \neq \top$ and prove well-definedness of $s_1^\natural \sqcup s_2^\natural$. Let $S_1^\natural = (V_{S_1^\natural}, Succ_{S_1^\natural}, L_{S_1^\natural}, E_{S_1^\natural}, iu_{S_1^\natural})$ and $S_2^\natural = (V_{S_2^\natural}, Succ_{S_2^\natural}, L_{S_2^\natural}, E_{S_2^\natural}, iu_{S_2^\natural})$ be the two state graphs of states $s_1^\natural$ and

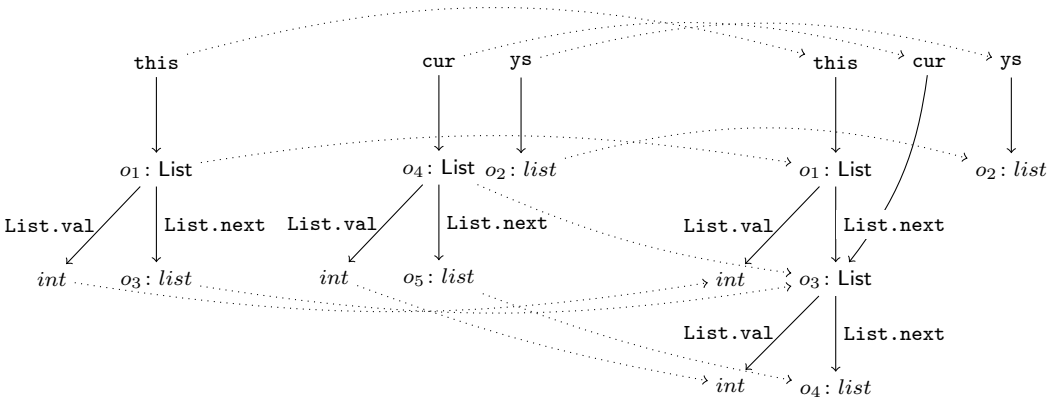


Figure 10: State homomorphism $m : S^\natural \to B^\natural$.

$s_2^\natural$, respectively. Furthermore, let $t^\natural$ be an abstraction of $s_1^\natural$ and $s_2^\natural$, and let $T^\natural = (V_{T^\natural}, Succ_{T^\natural}, L_{T^\natural}, E_{T^\natural}, iu_{T^\natural})$ be its state graph. By definition of abstraction, we have the following properties:

1. Let Stk (Loc) collect the stack (register) indices of state $s_1^\natural$. As $s_1^\natural \sqsubseteq t^\natural$, Stk (Loc) coincides with the set of stack (register) indices of $t^\natural$. Similarly for $s_2^\natural$ and thus $V_{T^\natural} \supseteq Stk \cup Loc$.
2. For any node $u \in T^\natural$ there exist uniquely defined nodes $v \in V_{S_1^\natural}$, $w \in V_{S_2^\natural}$ such that $L_{S_1^\natural}(v) \trianglelefteq L_{T^\natural}(u)$, $L_{S_2^\natural}(w) \trianglelefteq L_{T^\natural}(u)$. We say the nodes v and w *correspond* to u.
3. For any node $u \in T^\natural$ and any successor u' of u in $t^\natural$ there exists a successor v' (w') in $S_1^\natural$ ($S_2^\natural$) of the corresponding node v (w) in $S_1^\natural$ ($S_2^\natural$). Furthermore v' and w' correspond to u'.
4. For any edge $u \xrightarrow{\ell} u' \in T^\natural$ such that v (w) corresponds to u in $S_1^\natural$ ($S_2^\natural$) there is an edge $v \xrightarrow{\ell} v' \in S_1^\natural$ and an edge $w \xrightarrow{\ell} w' \in S_2^\natural$.
5. For any annotation $u \neq u' \in iu_{T^\natural}$ there exist $v \neq v'$ in $iu_{S_1^\natural}$ and $w \neq w'$ in $iu_{S_2^\natural}$, where v (v') and w (w') correspond to u (u').

In order to construct an abstraction $t^\natural$ of $s_1^\natural$ and $s_2^\natural$ we use the above mentioned properties as invariants and define its state graph $T^\natural$ by iterated extension. We define ${T^\natural}^0$ by setting $V_{T^\natural 0} := Stk \cup Loc$. Due to Property 1 these nodes exist in $S_1^\natural$ and $S_2^\natural$ as well. The labels of stack or register indices trivially coincide in $S_1^\natural$ and $S_2^\natural$, cf. Definition 14. Thus we set $L_{T^\natural 0}$ accordingly. Furthermore we set $Succ_{T^\natural 0} = E_{T^\natural 0} = iu_{T^\natural 0} := \varnothing$. Then ${T^\natural}^0$ satisfies Properties 1–5.

Suppose state graph ${T^\natural}^n$ has already been defined such that the Properties 1–5 are fulfilled. In order to update ${T^\natural}^n$, let $u \in V_{T^\natural n}$ such that v and w correspond to u. Suppose $v \xrightarrow{k} v' \in S_1^\natural$ and $w \xrightarrow{k} w' \in S_2^\natural$ such that there is no node u' in ${T^\natural}^n$ where v' and w' correspond to u'. Let u' denote a node fresh to ${T^\natural}^n$. We define $V_{T^\natural n+1} := V_{T^\natural n} \cup \{u'\}$ and establish Property 2 by setting $L_{T^\natural n+1}(u')$ such that $L_{S_1^\natural}(v') \trianglelefteq L_{T^\natural n+1}(u')$ and $L_{S_2^\natural} w' \trianglelefteq L_{T^\natural n+1}(u')$ where $L_{T^\natural n+1}(u')$ is as concrete as possible. If we succeed, we fix that v' and w' correspond to u'. It remains to update $iu_{T^\natural n+1}$ suitably such that Property 5 is fulfilled. If this also succeeds Properties 1–5 are fulfilled for ${T^\natural}^{n+1}$. On the other hand, if no further update is possible we set $T^\natural := {T^\natural}^n$. By construction $T^\natural$ is an abstraction of $S_1^\natural$ and $S_2^\natural$ and indeed represents $s_1^\natural \sqcup s_2^\natural$. Furthermore, the construction always terminates.

Example 10. Consider the states $A^\natural$, $B^\natural$, and $S^\natural$ described in Example 8. In Figure 9 an abstraction of $A^\natural$ and $B^\natural$ is given. In particular, abstraction $S^\natural$ results from the construction defined above, ie. $S^\natural = A^\natural \sqcup B^\natural$.

A sequence of abstract states $(s_i^\natural)_{i \geqslant 0}$ forms an ascending sequence, if $i < j$ implies $s_i^\natural \sqsubseteq s_j^\natural$. An ascending sequence $(s_i^\natural)_{i \geqslant 0}$ eventually stabilises, if there exists $i_0 \in \mathbb{N}$ such that for all $i \geqslant i_0$: $s_i^\natural = s_{i_0}^\natural$. The next lemma shows that any ascending sequence eventually stabilises.

Lemma 2. *The partial order $(\mathcal{AS}, \sqsubseteq)$ satisfies the ascending chain condition, that is, any ascending chain eventually stabilises.*

Proof. In order to derive a contradiction we assume the existence of an ascending sequence $(s_i^\natural)_{i\geqslant 0}$ that never stabilises. By definition for all $i \geqslant 0$: $|s_i^\natural| \geqslant |s_{i+1}^\natural|$. By assumption there exists $i \in \mathbb{N}$ such that for all $j > i$: $|s_i| = |s_j^\natural|$ and $s_i^\natural \sqsubset s_j^\natural$. The only possibility for two different states $s_i^\natural, s_j^\natural$ of equal size that $s_i^\natural \sqsubseteq s_j^\natural$ holds, is that addresses shared in $s_i^\natural$ become unshared in $s_j^\natural$. Clearly this is only possible for a finite amount of cases. Contradiction. □

Lemma 2 in conjunction with the fact that $(\mathcal{AS}, \sqsubseteq)$ has a least element $\bot$ and binary least upper bounds implies that $\mathcal{AS} := (\mathcal{AS}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ is a complete lattice. In particular any set of states $\mathcal{S}$ has a least upper bound, denoted as $\bigsqcup \mathcal{S}$. The meet operation $\bigsqcap$ can be expressed by $\bigsqcup$, yet in practice we do not need it.

5. Correctness

In the remainder of the paper we fix to a concrete JBC program P. Above, we already restricted our attention to well-formed JBC programs. For the proposed static analysis of these programs we additionally restrict to *non-recursive* methods. Note that the states in $\mathcal{AS}$ can in principle express recursive methods, but for recursive methods, we cannot use the below proposed construction to obtain *finite* computation graphs, as the graphs defined in Definition 21 cannot handle unbounded list of frames.

Let s be a state $(heap, frames)$. We define the *representation function* $\beta\colon \mathcal{JS} \to \mathcal{AS}$, that injects JVM states into $\mathcal{AS}$. For that let $\mathsf{dom}(heap) = \{p_1, \ldots, p_n\}$ and define iu such that all $p_i \neq p_j \in iu$ for different i, j.

Definition 16. Given the representation function β, we define the *abstraction function* $\alpha\colon \mathcal{P}(\mathcal{JS}) \to \mathcal{AS}$ and the *concretisation function* $\gamma\colon \mathcal{AS} \to \mathcal{P}(\mathcal{JS})$ as usual:

$$\alpha(\mathcal{S}) := \bigsqcup \{\beta(s) \mid s \in \mathcal{S}\} ,$$
$$\gamma(s^\natural) := \{s \in \mathcal{JS} \mid \beta(s) \sqsubseteq s^\natural\} .$$

We set $\alpha(s) := \alpha(\{s\})$.

Note that $\mathcal{AS}$ contains redundant states: Consider abstract states $s^\natural, t^\natural \in \mathcal{AS}$. Let $s^\natural = (heap, frames, iu)$, $p, q \in \mathsf{dom}(heap)$ and $p \neq q \in iu$. Let $t^\natural$ be defined like $s^\natural$ but $p \neq q \notin iu$. Now suppose that the types of p and q are not related with respect to the subclass order. Then $s^\natural \sqsubset t^\natural$ and $\gamma(s^\natural) = \gamma(t^\natural)$. To form a Galois insertion between $\mathcal{P}(\mathcal{JS})$ and $\mathcal{AS}$, we introduce a reduction operator that adds annotations for non-aliasing addresses.

Definition 17. Let $s^\natural$ be an abstract state $(heap, frames, iu)$. We define the *reduction operator* $\varsigma\colon \mathcal{AS} \to \mathcal{AS}$ as follows:

$$\varsigma(s^\natural) := (heap, frames, iu') ,$$

where $iu' := \{p \neq q \mid p, q \in \mathsf{dom}(heap)$ and there exists no $s \in \gamma(s^\natural)$ such that $m : s^\natural \to \beta(s)$ and $m(p) = m(q)\}$. Then $\varsigma(s^\natural) \sqsubseteq s^\natural$ and $\gamma(\varsigma(s^\natural)) = \gamma(s^\natural)$.

In practice, we compute the reduction by a unification argument of p and q in $s^\natural$: We try to construct a new state $t^\natural \sqsubseteq s^\natural$, where $r = m(p) = m(q)$. Let $T^\natural$ and $S^\natural$ be the state graphs of $t^\natural$ and $s^\natural$. Suppose u, v, w represent r, p, q in $T^\natural$ and $S^\natural$. We can use a similar reasoning we used for the join construction, but now require $L_{T^\natural}(u) \trianglelefteq L_{S^\natural}(v)$ and $L_{T^\natural}(u) \trianglelefteq L_{S^\natural}(w)$ if v and w correspond to u. If the construction succeeds, we can easily find a concrete state from $t^\natural$ such that $m(p) = m(q)$. The construction does not succeed if, for example, successors of corresponding nodes have different concrete values; then we add $p \neq q$.

Lemma 3. *The maps α and γ define a Galois insertion between the complete lattices $\mathcal{P}(\mathcal{JS})$ and $\varsigma^*(\mathcal{AS})$, where ς^* denotes the set extension of ς.*

Proof. It suffices to prove that γ is injective, ie. for all $s^\sharp, t^\sharp \in \varsigma^*(\mathcal{AS})$ if $s^\sharp \neq t^\sharp$ then $\gamma(s^\sharp) \neq \gamma(t^\sharp)$. Suppose $s^\sharp \neq t^\sharp$ but $\gamma(s^\sharp) = \gamma(t^\sharp)$. If the state graphs of $s^\sharp$ and $t^\sharp$ differ, it is a simple consequence of our morphism definition that $\gamma(s^\sharp) \neq \gamma(t^\sharp)$. Hence, $s^\sharp$ can only be different from $t^\sharp$ if the annotations of $s^\sharp$ and $t^\sharp$ differ. However, by construction they are equal. Contradiction. □

It follows that the reduction operator defined in Definition 17, indeed returns the greatest lower bound that represents the same element in the concrete domain as required. In the following we identify $\varsigma^*(\mathcal{AS})$ with $\mathcal{AS}$.

In order to prove that the abstract domain $\mathcal{AS}$ correctly approximates the concrete domain $\mathcal{P}(\mathcal{JS})$ we need to define a suitable notion of *abstract computation* on abstract states. Recall the set of JVM instructions introduced in Section 3. Based on these instructions, and actually mimicking them quite closely, we define how abstract states are evaluated symbolically.

In the following we assume that $s^\sharp$ denotes an abstract state $(heap, frames, iu)$. The instructions $\texttt{Push}^\sharp$, $\texttt{Pop}^\sharp$, $\texttt{Goto}^\sharp$ and $\texttt{New}^\sharp$ are defined like `Push`, `Pop`, `Goto` and `New`, respectively. The instructions $\texttt{Load}^\sharp$, $\texttt{Store}^\sharp$ and $\texttt{Return}^\sharp$ are defined like `Load`, `Store` and `Return`, but now consider abstract values. Next, consider instructions $\texttt{IAdd}^\sharp$, $\texttt{ISub}^\sharp$, $\texttt{ICmpGt}^\sharp$, $\texttt{BAnd}^\sharp$, $\texttt{BOr}^\sharp$ and $\texttt{BNot}^\sharp$. If the operands are concrete the instructions can be executed directly. Otherwise, we introduce a fresh variable together with a side-condition mimicking the instructions. As example for the latter case consider:

$$\texttt{IAdd}^\sharp \quad \frac{(heap, (i_2 :: i_1 :: stk, loc, cn, mn, pc) :: frames, iu)}{(heap, (i_3 :: stk, loc, cn, mn, pc + 1) :: frames, iu)} \qquad i_3 \equiv i_1 + i_2$$

In addition to symbolic evaluations, we define refinement steps on abstract states $s^\sharp$ if the information given in $s^\sharp$ is not concrete enough to execute a given instruction. Refinements are similar to the concept of materialisation as for example used in [32], and transforms a single abstract state in multiple abstract states. It will be a consequence of our definitions that for any abstract state $s_i^\sharp$ obtained from a refinement of $s^\sharp$, we have $s_i^\sharp \sqsubseteq s^\sharp$. We clarify the precise use of refinements and symbolic evaluations in the next section when generating the abstraction of P over abstract states in $\mathcal{AS}$. In short, we apply symbolic evaluation steps if possible and refinement steps when no symbolic evaluation step can be performed. Our construction computes a fixed point over the states obtained with refinement and evaluation steps and ensures that only a finite number of refinements are necessary before performing an symbolic evaluation step. Next, we present $\texttt{IfFalse}^\sharp$. If the top element of the stack is a concrete value, $\texttt{IfFalse}^\sharp$ is defined like `IfFalse`. Otherwise we perform a *Boolean refinement*, replacing the variable with values true and false.

Definition 18. Let $s^\sharp = (heap, (b :: stk, loc, cn, mn, pc) :: frames, iu)$, where b represents a boolean variable. The result of the *Boolean refinement* of $s^\sharp$ are two states in which b is replaced by true and false respectively.

$$\frac{(heap, (\mathsf{true} :: stk, loc, cn, mn, pc) :: frames, iu)}{(heap, (stk, loc, cn, mn, pc) :: frames, iu)} \qquad \frac{(heap, (\mathsf{false} :: stk, loc, cn, mn, pc) :: frames, iu)}{(heap, (stk, loc, cn, mn, pc) :: frames, iu)}$$

The instructions $\texttt{Getfield}^\sharp$, $\texttt{Putfield}^\sharp$, $\texttt{Checkcast}^\sharp$ and $\texttt{Invoke}^\sharp$ access the object on the heap. In abstract states, elements of the heap may be class variables. Recall that a class variable cn represents null as well as instances of cn and its subtypes. Therefore, if the top of the stack is an address p and $heap(s) = cn$ we perform an *instance refinement*.

Definition 19. Let $s^\sharp$ be an abstract state $(heap, frames, iu)$ and let p be an address such that $heap(p) = cn'$. Let cn be a subclass of cn'. Furthermore, suppose $(cn_1, fn_1), \ldots, (cn_n, fn_n)$ denote the fields of cn (together with the defining classes). We perform the following *instance refinement* where the second refinement takes care of the case when address p is replaced by null.

$$\frac{(heap\{p \mapsto cn'\}, frames, iu)}{(heap\{p \mapsto (cn, ftable_{cn})\}, frames, iu)} \qquad \frac{(heap\{p \mapsto cn'\}, frames, iu)}{(heap_{\mathsf{null}}, frames_{\mathsf{null}}, iu)}$$

Here $ftable_{cn}((cn_i, fn_i)) := v_i$ such that the type of the abstract variable v_i is defined in correspondence to the type of field (cn_i, fn_i), eg. a fresh *int* variable for integer fields. On the other hand we set $heap_{\mathsf{null}}$ $(frames_{\mathsf{null}})$ equal to *heap* $(frames)$, but $p \notin \mathsf{dom}(heap_{\mathsf{null}})$ and all occurrences of p are replaced by null.

If the accessed object on the heap is not abstract then $\texttt{Getfield}^\sharp$, $\texttt{Checkcast}^\sharp$ and $\texttt{Invoke}^\sharp$ behave like Getfield, Checkcast and Invoke. The next example shows that instance refinements properly handle dynamic dispatch.

Example 11. In Figure 11 we present an example detailing the need for the given definition of instance refinement. Here class B overrides method m inherited from class A. We only know the static type of the parameter when analysing method main(A a). Method main(A a) accepts null, any instance of class A or any instance of a subclass of A as parameter. In particular any instance of class B. Let's assume that the parameter is abstract when analysing main. (In fact our analysis always starts with a reasonable abstract state.) Thus, when invoking method m we perform the necessary refinement steps, considering the case that a is null the object A or the object B. Afterwards we can perform an symbolic evaluation step to analyse the class specific method.

```
class A{
  void m(){return;}
}
class B extends A{
  void m(){while(true);}
}
```

```
class Dispatch{
  void main(A a){
    a.m();
  }
}
```

Figure 11: Handling dynamic dispatch using instance refinement.

When manipulating the heap via $\texttt{Putfield}^\sharp$ we may perform additional refinement steps as addresses may-alias in the abstract representation of the heap. Consider a $\texttt{Putfield}^\sharp$ instruction on address p. Due to abstraction there may exist addresses $q \in \mathsf{dom}(heap)$ different from p that may-alias with p. Hence they are affected by the field update. We introduce *unsharing refinements* for all q, where $p \neq q \notin iu$.

Definition 20. Let $s^\sharp$ denote an abstract state $(heap, frames, iu)$ and let p and q be different addresses in *heap* such that $p \neq q \notin iu$, that is, p and q may-alias. We perform the following *unsharing* step: The first case forces these addresses to be distinct. The second case substitutes all occurrences of q with p.

$$\frac{(heap, frames, iu)}{(heap, frames, iu \cup \{p \neq q\})} \qquad \frac{(heap, frames, iu)}{(heap_p, frames_p, iu)}$$

Here $heap_p$ $(frames_p)$ is equal to *heap* $(frames)$ with all occurrences of q replaced by p.

The unsharing step demonstrates the main motivation for the annotations, namely the strict refinement of the abstraction with respect to $\sqsubseteq$ by performing necessary case distinctions on the abstract heap. If the accessed object is not abstract and no other address may-alias with the referenced address in the abstract heap, ie. after performing the necessary refinement steps, $\texttt{Putfield}^\sharp$ is defined like Putfield. Finally, we show the abstract computation steps for $\texttt{CmpEq}^\sharp$. The instruction $\texttt{CmpNeq}^\sharp$ is handled analogously. The $\texttt{CmpEq}^\sharp$ instruction splits into different cases depending on the compared values. We adapt the instruction to abstract values as follows:

1. Let val_1 and val_2 be addresses. If the addresses of val_1 and val_2 are the same then the test evaluates to true. Otherwise, we check wether $val_1 \neq val_2 \in iu$. If this is the case then the test returns false. Otherwise an unsharing refinement is performed (cf. Definition 20) before performing the test again.
2. Wlog. let val_1 be an address and val_2 be null. If $heap(val_1) = obj$ and $\mathsf{cl}(obj) = cn$, we perform an instance refinement (cf. Definition 19) on val_1 and re-consider the condition.

3. If val_1 and val_2 are concrete non-address Jinja values, then the test $(val_1 = val_2)$ can be directly executed and the symbolic evaluation equals the instruction on the JVM.
4. Wlog. val_1 is an abstract Boolean or integer variable. Then we introduce a new Boolean variable and the side condition $b \equiv (val_1 = val_2)$.

For the last case consider:

$$\texttt{CmpEq}^\sharp \quad \frac{(heap, (val_2 :: val_1 :: stk, loc, cn, mn, pc) :: frames, iu)}{(heap, (b :: stk, loc, cn, mn, pc + 1) :: frames, iu)} \qquad b \equiv (val_1 = val_2)$$

For the case where $s^\sharp$ is the minimal abstract state or maximal abstract state we consider following evaluation steps: Any instruction applied to $\bot$ yields $\top$ and any instruction to $\top$ yields $\top$.

Let $s^\sharp, s^{\sharp\prime}$ and $t^\sharp$ be abstract states such that $s^{\sharp\prime}$ is obtained by zero or multiple refinement steps from $s^\sharp$. Furthermore, suppose $t^\sharp$ is obtained from $s^{\sharp\prime}$ due to a symbolic evaluation. Then we say $t^\sharp$ is obtained from $s^\sharp$ by an *abstract computation*.

To prove correctness of a symbolic evaluation step it is enough to show that for all $s \in \gamma(s^\sharp)$ and $s \to_P t$ it follows that $t \in \gamma(t^\sharp)$, where $t^\sharp$ is obtained from $s^\sharp$ by applying a symbolic evaluation step. Similarly, to prove correctness of the refinement steps it is enough to show that for all $s \in \gamma(s^\sharp)$ there exists a state $s_i^\sharp$ obtained from $s^\sharp$ by a state refinement such that $s \in \gamma(s_i^\sharp)$. Correctness of an abstract computation step follows from the correctness of refinement and symbolic evaluation steps.

Lemma 4. *Let $s^\sharp \in \mathcal{AS}$. Suppose $s_1^\sharp, \ldots, s_n^\sharp$ are obtained by a state refinement from $s^\sharp$. Then $s^\sharp \sqsupseteq s_i^\sharp$ for all $s_i^\sharp$. Furthermore, $s \in \gamma(s^\sharp)$ implies that there exists an abstract state $s_i^\sharp$ such that $s \in \gamma(s_i^\sharp)$.*

Proof. The claim follows easily by the definition of Boolean and class variables, and the fact that two addresses in the heap of $s^\sharp$ either alias or not. □

Lemma 5. *Let $s^\sharp, t^\sharp \in \mathcal{AS}$ such that $t^\sharp$ is obtained by a symbolic evaluation from $s^\sharp$. Suppose $s \in \gamma(s^\sharp)$ and $s \to_P t$. Then $t \in \gamma(t^\sharp)$.*

Proof. The proof is straightforward in most cases; we only treat some informative ones. Let $s^\sharp = (heap^\sharp, frame^\sharp :: frames^\sharp, iu)$ and $s = (heap, frame :: frames)$. By assumption the domains of $frame^\sharp :: frames^\sharp$ and $frame :: frames$ coincide.

- Consider $\texttt{Load}^\sharp\ n$. By assumption $loc^\sharp(n) \trianglerighteq_m loc(n)$. In the abstract computation step $loc^\sharp(n)$ is loaded on to the top of the stack. Obviously $stk_i^\sharp(n) \trianglerighteq_m stk_i(n)$, where stk_i represents the top of the stack. Then $t \in \gamma(t^\sharp)$.

- Consider $\texttt{IAdd}^\sharp$. Let i_2, i_1 denote the first two stack elements of $s^\sharp$. Wlog. suppose that i_1 is abstract. By definition of the symbolic evaluation of $\texttt{IAdd}^\sharp$ we perform the step by introducing a new abstract integer i_3 and adding the constraint $i_3 \equiv i_1 + i_2$. Then $t \in \gamma(t^\sharp)$, since $i_3 \trianglerighteq z$ for all numbers z.

- Consider $\texttt{IfFalse}^\sharp\ i$. Wlog. let false be the top element of the stack of $s^\sharp$. Executing the symbolic step yields a state $t^\sharp$, which is an abstraction of t by assumption on s and $s^\sharp$. Then $t \in \gamma(t^\sharp)$.

- Consider $\texttt{Putfield}^\sharp\ cn\ fn$ on address p. By assumption the instruction can be symbolically evaluated and p does not alias with some address $q \in \mathsf{dom}(heap^\sharp)$ different from p. The only interesting case to consider is when $heap^\sharp(q)$ is a class variable and there exists $s \in \gamma(s^\sharp)$ such that $m(q) \rightharpoonup_S r \rightharpoonup_S^* m(p)$, where $r \in \mathsf{dom}(heap)$. Here S denotes the state graph of s. Then $m(q)$ reaches $m(p)$ via r and is affected by the update instruction. This does not matter, since $heap^\sharp(q)$ is also a class variable in $t^\sharp$, thus also representing the affected instance. Then $t \in \gamma(t^\sharp)$.

- Consider $\texttt{CmpEq}^\sharp$. By assumption the instruction can be symbolically executed. That is, the necessary refinement steps are already performed. Then $t \in \gamma(t^\sharp)$ follows directly. □

As α and γ define a Galois insertion between the set of Jinja states and the abstract states, the next theorem is an immediate result of the previous two lemmas.

Theorem 1. *Let s and t be JVM states, such that $s \to_P^* t$. Suppose $s \in \gamma(s^\natural)$ for some abstract state $s^\natural$. Then there exists an abstract computation of $t^\natural$ from $s^\natural$ such that $t \in \gamma(t^\natural)$.*

Theorem 1 formally proves the correctness of the proposed abstract domain with respect to the semantics for JBC, as established by Klein and Nipkow [15]. In order to exploit this abstract domain we require a finite representation of the abstract domain $\mathcal{AS}$ induced by P. For that we propose in the next section computation graphs as finite representations of all relevant states in $\mathcal{AS}$, abstracting JVM states in P.

6. Computation Graphs

In this section, we define *computation graphs* as *finite* representations of the abstract domain $\mathcal{AS}$ with respect to P. We first provide an intuition of the computation graph method and relate it to the casual approach of data flow analysis as presented for example in [33]. The following steps are typically involved in data flow analysis: (i) computation of the control flow graph of P; (ii) mapping of functions f to labels; (iii) computation of a fixed point from an initial state of the abstract domain. Computation graphs unify the above mentioned items: Starting from an initial abstract state, states are dynamically expanded through abstract computation. Thus a control flow graph is dynamically generated. Its node are elements of the abstract domain. We obtain a finite representation of loops, if we suitably exploit the fact that any subset of $\mathcal{AS}$ has a least upper bound.

Definition 21. A *computation graph* $G = (V_G, E_G)$ is a directed graph with edge labels, where $V_G \subset \mathcal{AS}$ and $s^\natural \xrightarrow{\ell} t^\natural \in E_G$ if either $t^\natural$ is obtained from $s^\natural$ by an abstract computation or $s^\natural$ is an instance of $t^\natural$. Furthermore, if there exists a constraint C in the symbolic evaluation, then $\ell := C$. For all other cases $\ell := \varnothing$. We say that G is the computation graph of program P if there exists an (initial) abstract state $i^\natural \in G$ such that for all initial states i of P we have that $i \in \gamma(i^\natural)$.

Example 12. Consider the `List` class from Example 1 together with the well-formed JBC program depicted in Figure 4. Figure 12 illustrates the computation graph of `append`. For the sake of readability we omit the `List.val` field of the list, the unsharing annotations and some intermediate nodes.

Consider the initial node $I^\natural$, which is an abstraction of all concrete initial states, when `this` is not `null`. We assume that `this` is acyclic and initially does not share with `ys`. Nodes $A^\natural$, $B^\natural$ and $S^\natural$ correspond to the situation described in Example 6 and Example 7. That is, node $A^\natural$ is obtained after assigning `cur` to `this` before any iteration of the loop, node $B^\natural$ is obtained after exactly one iteration of the loop and node $S^\natural = \bigsqcup\{A^\natural, B^\natural\}$. Intermediate iterations are usually removed in the resulting graph. For clarity we illustrate the states of the intermediate iteration with a dashed border.

After pushing the reference of `cur.next` and `null` onto the operand stack, we reach node $C^\natural$. At `pc` $= 7$ we want to compare the reference of `cur.next` with `null`. But, `cur.next` is not concrete. Therefore, a class instance refinement is performed, yielding nodes $C_1^\natural$ and $C_2^\natural$.

First, we consider that `cur.next` is not `null`, but references an arbitrary instance as illustrated in node $C_1^\natural$. The step from $C_1^\natural$ to $D^\natural$ is straightforward. Let id denote the identity function and $m = id(V_{S^\natural})$. Then $m\{o_4 \mapsto o_5, o_5 \mapsto o_6\}$ is a morphism from $S^\natural$ to $D^\natural$. Therefore, $D^\natural$ is an instance of $S^\natural$. Second, we consider the case when `cur.next` is `null`, as depicted in node $C_2^\natural$. Node $E^\natural$ is obtained from $C_2^\natural$ after loading registers `cur` and `ys` onto the stack. At program counter 19 a `Putfield`$^\natural$ instruction is performed. Therefore we perform a refinement according to Definition 20. We consider the subcase where `this` and `cur` point to the same reference. Thus the corresponding addresses are set equal and we obtain node $E_1^\natural$. Alternatively, we assume $o_1 \neq o_3$, which is exemplarily added to the unsharing annotation. Based on this subcase, we apply another unsharing refinement with respect to the addresses o_3 and o_4. In $E_2^\natural$ we depict the case where $o_3 = o_4$, while in $E_3^\natural$ assume that $o_3 \neq o_4$ holds, which is again added to the unsharing annotations.

Nodes F_1, F_2 and F_3 are obtained after performing the `Putfield`$^\natural$ instruction.

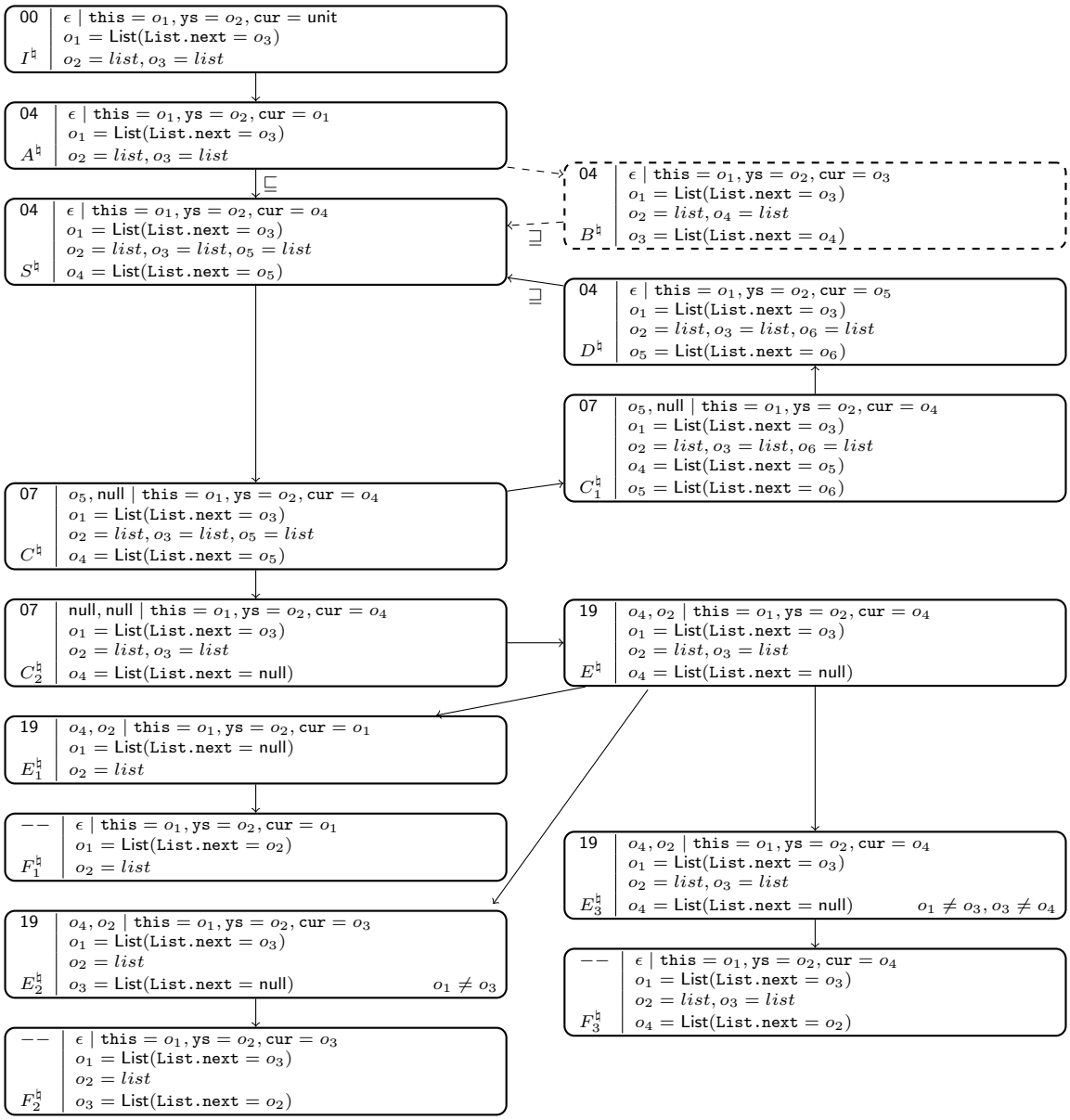


Figure 12: The (incomplete) computation graph of `append`.

To concretise the employed strategy, note that whenever we are about to finish a loop, we attempt to use an instance refinement to the state starting this loop. If this fails, for example in an attempted step from $B^\natural$ to $A^\natural$ in Example 12, we widen the corresponding state. Here we collect all states that need to be abstracted and join them to obtain an abstraction. Complementing the proposed strategy, we restrict the applications of refinements, such that refinement steps are only performed if no other steps are applicable. We say that this strategy is an *eager strategy*. The next lemma shows that if an eager strategy is followed we are guaranteed to obtain a *finite* computation graph.

Lemma 6. *Let G denote the computation graph of P obtained from some initial abstract state $i^\natural$ by applying an eager strategy. Then G is finite.*

Proof. We argue indirectly. Suppose the computation graph G of P is infinite. This can only happen if there exists a loop in P. Since we are applying an eager strategy it follows that the widening operation for this loop gives rise to an infinite sequence of states $(s^\sharp_j)_{j\geqslant 0}$ such that $s^\sharp_j \sqsubset s^\sharp_{j+1}$ for all j. However, this is impossible as any ascending chain of abstract states eventually stabilises, cf. Lemma 2. □

Let G be a computation graph. In what follows, we assume that G is always obtained by an eager strategy. Hence G is always finite. We write $s^\sharp \rightharpoonup_G t^\sharp$ to indicate that state $t^\sharp$ is directly reachable in G from $s^\sharp$. Sometimes we want to distinguish whether $t^\sharp$ is obtained by a refinement (denoted as $s^\sharp \rightharpoonup_{\text{ref}} t^\sharp$) or by a symbolic evaluation (denoted as $s^\sharp \rightharpoonup_{\text{eva}} t^\sharp$), or whether $s^\sharp$ is an instance of $t^\sharp$ (denoted as $s^\sharp \rightharpoonup_{\text{ins}} t^\sharp$). If $t^\sharp$ is reachable from $s^\sharp$ in G we write $s^\sharp \rightharpoonup^*_G t^\sharp$. If $s^\sharp \neq t^\sharp$ this is denoted by $s^\sharp \rightharpoonup^+_G t^\sharp$.

Lemma 7. *Let $s, t \in \mathcal{JS}$ such that $s \to_P t$. Let G denote the computation graph of P obtained from some initial state $i^\sharp$, such that $s^\sharp \in G$. Suppose $s \in \gamma(s^\sharp)$, then there exists $t^\sharp \in G$ such that $t \in \gamma(t^\sharp)$ and $s^\sharp \rightharpoonup^*_{\text{ins}} \circ \rightharpoonup^*_{\text{ref}} \circ \rightharpoonup_{\text{eva}} t^\sharp$. Furthermore $s^\sharp \rightharpoonup^*_{\text{ins}} \circ \rightharpoonup^*_{\text{ref}} \circ \rightharpoonup_{\text{eva}} t^\sharp$ has finitely many instance and refinement steps, depending only on G*

Proof. By construction of G we have to consider two cases: Suppose $t^\sharp$ is obtained by an abstract computation from $s^\sharp$. We employ Lemma 5 to conclude that $t \in \gamma(t^\sharp)$. Then $s^\sharp \rightharpoonup^*_{\text{ref}} \circ \rightharpoonup_{\text{eva}} t^\sharp$. Next, suppose $t^\sharp$ is obtained by an abstract computation from $s^{\sharp\prime}$, where $s^\sharp \sqsubseteq s^{\sharp\prime}$. Hence, we also have $s \in \gamma(s^{\sharp\prime})$. We employ Lemma 5 to conclude that $t \in \gamma(t^\sharp)$. Then $s^\sharp \rightharpoonup^*_{\text{ins}} \circ \rightharpoonup^*_{\text{ref}} \circ \rightharpoonup_{\text{eva}} t^\sharp$. The second claim follows by construction as G is finite. □

We arrive at the main result of this section.

Theorem 2. *Let $i, t \in \mathcal{JS}$ and suppose $i \to^*_P t$, where the derivation height of the execution is m. Let G denote the computation graph of P obtained from some initial state $i^\sharp$ such that $i \in \gamma(i^\sharp)$. Then there exists an abstraction $t^\sharp \in G$ and a path $i^\sharp \rightharpoonup^*_G t^\sharp$ of length m' such that $m \leqslant m' \leqslant K \cdot m$. Here constant $K \in \mathbb{N}$ only depends on G.*

Proof. By induction on m (employing Lemma 7), we conclude the existence of an abstract state $t^\sharp$ such that $i \rightharpoonup^*_G t^\sharp$. Hence, the first part of the theorem follows. Furthermore by Lemma 7 there exists m' such that $m \leqslant m' \leqslant K \cdot m$. □

By construction together with Lemma 6, we obtain following corollary.

Corollary 1. *The computation graph method is* total. *That is, the computation graph method can be applied to any well-formed and non-recursive JBC program and the computation graph obtained is guaranteed to be finite.*

Recall Definition 1. From Theorem 2 and standard properties of Galois connections we obtain following corollary.

Corollary 2. *Let P be a program and $\mathcal{S} \subseteq \mathcal{JS}$ denote a set of initial states. Suppose computation graph G is obtained from the initial abstract state $\alpha(\mathcal{S})$. We set $s^\sharp \gg s$ iff $s \in \gamma(s^\sharp)$. We have $\gg \circ \to_{P|_\mathcal{S}} \subseteq \rightharpoonup^*_{\text{ins}} \circ \rightharpoonup^*_{\text{ref}} \circ \rightharpoonup_{\text{eva}} \circ \gg$ and $\alpha(\mathcal{S}) \gg s$ for all $s \in \mathcal{S}$. Hence $\gg$ is the abstraction of $\to_{P|_\mathcal{S}}$ to $\rightharpoonup^*_{\text{ins}} \circ \rightharpoonup^*_{\text{ref}} \circ \rightharpoonup_{\text{eva}}$.*

Note that Corollary 2 only establishes an abstraction rather than a complexity reflecting abstraction. Although it is possible to extend the result by defining a suitable size measure, we do not gain much from it as the computation graph G obtained from a program P represents a finite relation over $\mathcal{AS}$. Without further reasoning the derivation height for any path reaching a cycle in G is not defined. In the next section we show how to make use of a term-based representation of G and establish a complexity reflecting abstraction from JBC programs to rewrite systems.

In practice, when construction the computation graph we usually start with a suitable abstract state. In Example 12 we assume that `this` is not `null` and the objects bound to `this` and `ys` do not share. By construction, the abstraction is sound for all states that are represented by the initial state.

7. Constraint Rewrite Systems

Let G be the computation graph for program P with initial state $i^\natural$; G is kept fixed for the remainder of the section. In the following we describe the translation from G into a *constraint term rewrite system* (*cTRS* for short). Our definition is a variation of cTRSs as for example defined by Falke and Kapur [34, 35] or Sakata et al. [36]. Kop and Nishida introduced a very general formalism of term rewrite systems with constraints, termed *logical constrained term rewrite systems (LCTRSs)* [37]. The proposed notion of cTRSs is not directly interchangeable with LCTRSs, yet the rewrite system resulting from the transformation could also be formalised as LCTRS. The here proposed transformation is inspired by [27]. Otto et al. transform *termination graphs* into *integer term rewrite systems* (*ITRSs* for short) [38].

Let $\mathcal{C}$ be a (not necessarily finite) sorted signature, let $\mathcal{V}'$ denote a countably infinite set of sorted variables. Furthermore let T denote a theory over $\mathcal{C}$. Quantifier-free formulas over $\mathcal{C}$ are called *constraints*. Suppose $\mathcal{F}$ is a sorted signature that extends $\mathcal{C}$ and let $\mathcal{V} \supseteq \mathcal{V}'$ denote an extension of the variables in $\mathcal{V}'$. Let $\mathcal{T}(\mathcal{F}, \mathcal{V})$ denote the set of *(sorted) terms* over the signature $\mathcal{F}$ and $\mathcal{V}$. Note that the sorted signature is necessary to distinguish between *theory* variables that are to be interpreted over the theory T and *term* variables whose interpretation is free. A *constraint rewrite rule*, denoted as $l \to r \,[\![C]\!]$, is a triple consisting of terms l and r, together with a constraint C. We assert that $l \notin \mathcal{V}$, but do *not* require that $\mathsf{Var}(l) \supseteq \mathsf{Var}(r) \cup \mathsf{Var}(C)$, where $\mathsf{Var}(t)$ ($\mathsf{Var}(C)$) denotes the variables occurring in the term t (constraint C). A *constraint term rewrite system* (*cTRS*) is a finite set of constraint rewrite rules.

Let $\mathcal{R}$ denote a cTRS. A context D is a term with exactly one occurrence of a *hole* $\Box$, and $D[t]$ denotes the term obtained by replacing the hole $\Box$ in D by the term t. A substitution σ is a function that maps variables to terms, and $t\sigma$ denotes the homomorphic extension of this function to terms. We define the rewrite relation $\to_{\mathcal{R}}$ as follows. For terms s and t, $s \to_{\mathcal{R}} t$ holds, if there exists a context D, a substitution σ and a constraint rule $l \to r \,[\![C]\!] \in \mathcal{R}$ such that $s =_T D[l\sigma]$ and $t = D[r\sigma]$ with $T \vdash C\sigma$. Here $=_T$ denotes unification modulo T. For extra variables x, possibly occurring in t, we demand that $\sigma(x)$ is in normal-form.

We often drop the reference to the cTRS $\mathcal{R}$, if no confusion can arise from this. A function symbol in $\mathcal{F}$ is called *defined* if f occurs as the root symbol of l, where $l \to r \,[\![C]\!] \in \mathcal{R}$. Function symbols in $\mathcal{F} \setminus \mathcal{C}$ that are not defined are called *constructor* symbols, and the symbols in $\mathcal{C}$ are called *theory* symbols. A term $t = f(t_1, \ldots, t_k)$ is a *basic term* if f is a defined symbol and the terms t_i are only built over constructor, theory symbols, and variables.

A cTRS $\mathcal{R}$ is called *terminating* if the relation $\to_{\mathcal{R}}$ is well-founded. For a terminating cTRS $\mathcal{R}$, we define its *runtime complexity*, denoted as rctrs. We adapt the runtime complexity with respect to a standard TRS suitable for cTRS $\mathcal{R}$. (See [5] for the standard definition.) Note that $\to_{\mathcal{R}}$ is not necessarily finitely branching for finite cTRSs, as fresh variables on the right-hand side of a rule can occur. That is why we make use of Kleene equality.

Definition 22. We define the *runtime complexity* (with respect to $\mathcal{R}$) as follows:

$$\mathsf{rctrs}(n) =_{\mathsf{k}} \mathsf{cp}(n, \mathcal{T}_{\mathsf{b}}, \|\cdot\|, \to_{\mathcal{R}}) \,,$$

where $\mathcal{T}_{\mathsf{b}}$ denotes the set of *basic* terms. We fix the size measure $\|\cdot\|$ that depends on theory T below.

In the following we are only interested in cTRSs over a specific theory T, namely Presburger arithmetic, that is, we have $T \vdash C$, if all ground instances of the constraint C are valid in Presburger arithmetic. Recall that Presburger arithmetic is decidable. If $T \vdash C$ then C is *valid*. On the other hand, if there exists a substitution σ, such that $T \vdash C\sigma$, then C is *satisfiable*.

To represent the basic operations in the Jinja bytecode instruction set (cf. Section 3) we collect the following connectives and truth constants in $\mathcal{C}$: $\wedge$, $\vee$, $\neg$, true, and false, together with the following relations and operations: $=$, $\neq$, $\geqslant$, $+$, $-$. Furthermore, we add infinitely many integer constants. We often write $l \to r$ instead of $l \to r \,[\![\mathsf{true}]\!]$. As expected $\mathcal{C}$ makes use of two sorts: `bool` and `int`. We suppose that all abstract variables $X_1, X_2, \ldots$ are present in the set of variables $\mathcal{V}$, where abstract integer (Boolean) variables are assigned sort `int` (`bool`) and all other variables are assigned sort `univ`. The remaining elements of the signature $\mathcal{F}$ will be defined in the course of this section. As the signature of these function symbols is easily

read off from the translation given below, in the following the sort information is left implicit, to simplify the presentation. We fix the size of a term t, denoted as $\|t\|$, in the theory of Presburger arithmetic as follows:

$$\|t\| := \begin{cases} 1 & \text{if } t \text{ is a variable} \\ \mathsf{abs}(t) & \text{if } t \text{ is an integer} \\ 1 + \sum_{i=1}^{n} \|t_i\| & \text{if } t = f(t_1, \ldots, t_n) \text{ and } f \text{ is not an integer} , \end{cases}$$

where $\mathsf{abs}(i)$ denotes the absolute value of an integer i.

In what follows we make use of the *maybe-cyclic* and *may-reach* approximations as presented for example in [17–19]. Given an abstract state $(heap, frames, iu)$ and $p, q \in heap$, maybe-cyclic provides a safe approximation wether p could be cyclic in any execution of P and may-reach provides a safe approximation wether an address p reaches q in any execution of P. In principle we are able to approximate these properties in $\mathcal{AS}$, however we make use of the modularity of the abstract interpretation framework and incorporate existing analyses to improve effectivity of the overall approach. In the next section we clarify how the analyses presented in [17–19] are incorporated in our prototype.

Next, we show how an abstract state becomes representable as a term over $\mathcal{F}$.

Definition 23. Let $s^\natural$ be an abstract state $(heap, frames, iu)$. Suppose v is a value. Then v is translated as follows:

$$\mathsf{tval}(v) := \begin{cases} \mathsf{null} & \text{if } v \in \{\mathsf{unit}, \mathsf{null}\} \\ v & \text{if } v \text{ is a non-address value, except } \mathsf{unit} \text{ or } \mathsf{null} \\ \mathsf{taddr}(v) & \text{if } v \text{ is an address} . \end{cases}$$

Let a be an address. Then a is translated as follows:

$$\mathsf{taddr}(a) := \begin{cases} x & \text{if } a \text{ is maybe-cyclic and } x \text{ is a fresh variable} \\ x & \text{if } heap(a) \text{ denotes an abstract variable } x \\ cn(\mathsf{tval}(v_1), \ldots, \mathsf{tval}(v_n)) & \text{if } heap(a) = (cn, ftable) . \end{cases}$$

Here we suppose in the last case that $\mathsf{dom}(ftable) = \{(cn_1, id_1), \ldots, (cn_n, id_n)\}$ and for all $1 \leqslant i \leqslant n$: $ftable((cn_i, id_i)) = v_i$. Finally, to translate the abstract state $s^\natural$ into a term, it suffices to translate the values of the registers and the operand stacks of all frames in the list $frames$. Thus we set

$$\mathsf{ts}(s^\natural) := \mathsf{f}_{s^\natural}(\mathsf{tval}(stk_1(1)), \ldots, \mathsf{tval}(stk_k(|stk_k|)), \mathsf{tval}(loc_1(1)), \ldots, \mathsf{tval}(loc_k(|loc_k|))) ,$$

Example 13. Consider the presentation of state $C^\natural$ in Figure 12. Then $\mathsf{ts}(C^\natural)$ yields the following term:

$$\mathsf{ts}(C^\natural) = \mathsf{f}_{C^\natural}(list_5, \mathsf{null}, \mathsf{List}(list_3), list_2, \mathsf{List}(\mathsf{List}(list_5))) .$$

Observe that our term representation can only fully represent acyclic data. In this sense, the term representation of a state is less general, than its graph-based representation. However, we still obtain the following lemma.

Lemma 8. *Let $s^\natural$ and $t^\natural$ be abstract states. If $t^\natural \sqsubseteq s^\natural$, then there exists a substitution σ such that $\mathsf{ts}(t^\natural) = \mathsf{ts}(s^\natural)\sigma$.*

Proof. Let $S^\natural$ and $T^\natural$ be the state graphs of $s^\natural$ and $t^\natural$, respectively. By assumption there exists a morphism $m\colon S^\natural \to T^\natural$. The lemma is a direct consequence of the following observations:

- Consider the terms $\mathsf{ts}(s^\natural)$ and $\mathsf{ts}(t^\natural)$. By construction these terms encode the standard term representations of the graphs $S^\natural$ and $T^\natural$.

- Let u and v be nodes in $S^\sharp$ and $T^\sharp$ such that $m(u) = v$. The label of u (in $S^\sharp$) can only be distinct from the label of v (in $T^\sharp$), if $L_{S^\sharp}(u)$ is an abstract variable or null. In the former case $\mathsf{tval}(L_{S^\sharp}(u))$ is again a variable and the latter case implies that $L_{T^\sharp}(v) = \mathsf{unit}$. Thus in both cases, $\mathsf{tval}(L_{S^\sharp}(u))$ matches $\mathsf{tval}(L_{T^\sharp}(v))$.
- By assumption, we have $m(u)$ is maybe-cyclic, if v is maybe-cyclic. In this case $\mathsf{tval}(L_{S^\sharp}(u))$ and $\mathsf{tval}(L_{T^\sharp}(v))$ are fresh variables. Hence, $\mathsf{tval}(L_{S^\sharp}(u))$ matches $\mathsf{tval}(L_{T^\sharp}(v))$. □

Recall Definition 6. The next lemma relates the size of a state to its term representation.

Lemma 9. *Let s be a state $(heap, frames)$. Then $\|\mathsf{ts}(\beta(s))\| \in O(|s|)$.*

Proof. As a consequence of Definition 6 and the above proposed variant of the term size we see that $\|\mathsf{ts}(\beta(s))\| \leqslant |s| + 1$ for all states s. Note that $\mathsf{ts}(\beta(s))$ uses an extra compound symbol to collect the term representation of the register entries of $\beta(s)$ and cyclic data in $\beta(s)$ are replaced by fresh variables. □

Definition 24. Let G be a finite computation graph and $s^\sharp = (heap, frames, iu)$ and $t^\sharp$ be abstract states in G. We define the constraint rule *corresponding* to the edge $s^\sharp \to t^\sharp \in G$ denoted by $\mathsf{rule}(s^\sharp, t^\sharp)$, as follows:

$$\mathsf{rule}(s^\sharp, t^\sharp) = \begin{cases} \mathsf{f}_{s^\sharp}(\mathsf{ts}(s^\sharp)) \to \mathsf{f}_{t^\sharp}(\mathsf{ts}(s^\sharp)) & \text{if } s^\sharp \sqsubseteq t^\sharp \\ \mathsf{f}_{s^\sharp}(\mathsf{ts}(t^\sharp)) \to \mathsf{f}_{t^\sharp}(\mathsf{ts}(t^\sharp)) & \text{if } t^\sharp \text{ is a state refinement of } s^\sharp \\ \mathsf{f}_{s^\sharp}(\mathsf{ts}(s^\sharp)) \to \mathsf{f}_{t^\sharp}(\mathsf{ts}(t^\sharp))\,[\![\mathsf{tval}(C)]\!] & \text{the edge is labelled by } C \\ \mathsf{f}_{s^\sharp}(\mathsf{ts}(s^\sharp)) \to \mathsf{f}_{t^\sharp}(\mathsf{ts}^*(t^\sharp)) & s^\sharp \text{ corresponds to a } \texttt{Putfield}^\sharp \text{ on address } p, \\ & heap(q) \text{ is variable } cn, \text{ and } q \text{ may-reach } p \\ \mathsf{f}_{s^\sharp}(\mathsf{ts}(s^\sharp)) \to \mathsf{f}_{t^\sharp}(\mathsf{ts}(t^\sharp)) & \text{otherwise .} \end{cases}$$

Here $\mathsf{tval}(C)$ denotes the standard extension of the mapping tval to labels of edges and ts^* is defined as ts but employs fresh variables for any reference q that may-reach the object that is updated. The cTRS obtained from G consists of rules $\mathsf{rule}(s^\sharp, t^\sharp)$ for all edges $s^\sharp \to t^\sharp \in G$.

Example 14. Figure 13 illustrates the cTRS obtained from the computation graph of Example 12. To ease readability we use l to represent *list* variables. In the last rule l_4 is fresh on the right-hand side. This is because we update `cur` and have a side-effect on `this` that is not directly observable in the abstraction.

In the following we show that the rewrite relation of the obtained cTRS safely approximates the concrete semantics of the concrete domain. We first argue informally:

- By Lemma 7 there exists a path $s^\sharp \to_{\text{ins}}^* \circ \to_{\text{ref}}^* \circ \rightharpoonup_{\text{eva}} t^\sharp$ in G for $s \to_P t$ such that $s \in \gamma(s^\sharp)$ and $t \in \gamma(t^\sharp)$.
- Together with Lemma 8 we have to show that $\mathsf{f}_{s^\sharp}(\mathsf{ts}(\beta(s))) \to_{\mathcal{R}}^+ \mathsf{f}_{t^\sharp}(\mathsf{ts}(\beta(t)))$.
- We do this by inspecting the rules obtained from the transformation. We will see that instance steps and refinement steps do not modify the term instance. In case of evaluation steps the effect is either directly observable in the abstract state, as it happens for example for the $\texttt{Push}^\sharp$ instruction, or indirectly by requiring that the substitution is conform with the constraint. In the case of the $\texttt{Putfield}^\sharp$ instruction we have to find a suitable substitution for fresh variables to accommodate possible side-effects.

Lemma 10. *Let $s^\sharp$ and $t^\sharp$ be states in G connected by an edge $s^\sharp \xrightarrow{\ell} t^\sharp$ from $s^\sharp$ to $t^\sharp$. Suppose $s \in \mathcal{JS}$ and $s \in \gamma(s^\sharp)$. Suppose further that if the constraint ℓ labelling the edge is non-empty, then s satisfies ℓ. Moreover, if $s^\sharp \xrightarrow{\ell} t^\sharp$ follows due to a refinement step, then s is consistent with the chosen refinement. Then there exists $t \in \gamma(t^\sharp)$ such that $\mathsf{f}_{s^\sharp}(\mathsf{ts}(s')) \to_{\mathsf{rule}(s^\sharp, t^\sharp)} \mathsf{f}_{t^\sharp}(\mathsf{ts}(t'))$ with $s' = \beta(s)$, $t' = \beta(t)$.*

Proof. Recall Definition 16. By construction we have $s \in \gamma(s^\sharp)$ iff $\beta(s) \sqsubseteq s^\sharp$ for any concrete state s and abstract state $s^\sharp$. In particular we also have $s' \sqsubseteq s^\sharp$. The proof proceeds by case analysis on the edge $s^\sharp \xrightarrow{\ell} t^\sharp$ in G,

$$
\begin{aligned}
\mathsf{f}_{I^\sharp}(\mathsf{List}(l_3), l_2, \mathsf{null}) &\to \mathsf{f}_{A^\sharp}(\mathsf{List}(l_3), l_2, \mathsf{List}(l_3))\\
\mathsf{f}_{A^\sharp}(\mathsf{List}(l_3), l_2, \mathsf{List}(l_3)) &\to \mathsf{f}_{S^\sharp}(\mathsf{List}(l_3), l_2, \mathsf{List}(l_3))\\
\mathsf{f}_{S^\sharp}(\mathsf{List}(l_3), l_2, \mathsf{List}(l_5)) &\to \mathsf{f}_{C^\sharp}(l_5, \mathsf{null}, \mathsf{List}(l_3), l_2, \mathsf{List}(l_5))\\
\mathsf{f}_{C^\sharp}(\mathsf{List}(l_6), \mathsf{null}, \mathsf{List}(l_3), l_2, \mathsf{List}(\mathsf{List}(l_6))) &\to \mathsf{f}_{C_1^\sharp}(\mathsf{List}(l_6), \mathsf{null}, \mathsf{List}(l_3), l_2, \mathsf{List}(\mathsf{List}(l_6)))\\
\mathsf{f}_{C^\sharp}(\mathsf{null}, \mathsf{null}, \mathsf{List}(l_3), l_2, \mathsf{List}(\mathsf{null})) &\to \mathsf{f}_{C_2^\sharp}(\mathsf{null}, \mathsf{null}, \mathsf{List}(l_3), l_2, \mathsf{List}(\mathsf{null}))\\
\mathsf{f}_{C_1^\sharp}(\mathsf{List}(l_6), \mathsf{null}, \mathsf{List}(l_3), l_2, \mathsf{List}(\mathsf{List}(l_6))) &\to \mathsf{f}_{D^\sharp}(\mathsf{List}(l_3), l_2, \mathsf{List}(l_6))\\
\mathsf{f}_{D^\sharp}(\mathsf{List}(l_3), l_2, \mathsf{List}(l_6)) &\to \mathsf{f}_{S^\sharp}(\mathsf{List}(l_3), l_2, \mathsf{List}(l_6))\\
\mathsf{f}_{C_2^\sharp}(\mathsf{null}, \mathsf{null}, \mathsf{List}(l_3), l_2, \mathsf{List}(\mathsf{null})) &\to \mathsf{f}_{E^\sharp}(\mathsf{List}(\mathsf{null}), l_2, \mathsf{List}(l_3), l_2, \mathsf{List}(\mathsf{null}))\\
\mathsf{f}_{E^\sharp}(\mathsf{List}(\mathsf{null}), l_2, \mathsf{List}(\mathsf{null}), l_2, \mathsf{List}(\mathsf{null})) &\to \mathsf{f}_{E_1^\sharp}(\mathsf{List}(\mathsf{null}), l_2, \mathsf{List}(\mathsf{null}), l_2, \mathsf{List}(\mathsf{null}))\\
\mathsf{f}_{E_1^\sharp}(\mathsf{List}(\mathsf{null}), l_2, \mathsf{List}(\mathsf{null}), l_2, \mathsf{List}(\mathsf{null})) &\to \mathsf{f}_{F_1^\sharp}(\mathsf{List}(l_2), l_2, \mathsf{List}(l_2))\\
\mathsf{f}_{E^\sharp}(\mathsf{List}(\mathsf{null}), l_2, \mathsf{List}(\mathsf{List}(\mathsf{null})), l_2, \mathsf{List}(\mathsf{null})) &\to \mathsf{f}_{E_2^\sharp}(\mathsf{List}(\mathsf{null}), l_2, \mathsf{List}(\mathsf{List}(\mathsf{null})), l_2, \mathsf{List}(\mathsf{null}))\\
\mathsf{f}_{E_2^\sharp}(\mathsf{List}(\mathsf{null}), l_2, \mathsf{List}(\mathsf{List}(\mathsf{null})), l_2, \mathsf{List}(\mathsf{null})) &\to \mathsf{f}_{F_2^\sharp}(\mathsf{List}(\mathsf{List}(l_2)), l_2, \mathsf{List}(l_2))\\
\mathsf{f}_{E^\sharp}(\mathsf{List}(\mathsf{null}), l_2, \mathsf{List}(l_3), l_2, \mathsf{List}(\mathsf{null})) &\to \mathsf{f}_{E_3^\sharp}(\mathsf{List}(\mathsf{null}), l_2, \mathsf{List}(l_3), l_2, \mathsf{List}(\mathsf{null}))\\
\mathsf{f}_{E_3^\sharp}(\mathsf{List}(\mathsf{null}), l_2, \mathsf{List}(l_3), l_2, \mathsf{List}(\mathsf{null})) &\to \mathsf{f}_{F_3^\sharp}(\mathsf{List}(l_4), l_2, \mathsf{List}(l_2))
\end{aligned}
$$

Figure 13: The cTRS of `append`.

- *Case* $s^\sharp \xrightarrow{\ell} t^\sharp$, as $s^\sharp \sqsubseteq t^\sharp$; $\ell = \varnothing$. By transitivity of the instance relation we have $s' \sqsubseteq t^\sharp$ and thus $s \in \gamma(t^\sharp)$. By Lemma 8 there exists a substitution σ such that $\mathsf{ts}(s') = \mathsf{ts}(s^\sharp)\sigma$. In sum, we obtain:

$$\mathsf{f}_{s^\sharp}(\mathsf{ts}(s')) = \mathsf{f}_{s^\sharp}(\mathsf{ts}(s^\sharp))\sigma \to_{\mathsf{rule}(s^\sharp, t^\sharp)} \mathsf{f}_{t^\sharp}(\mathsf{ts}(s^\sharp))\sigma = \mathsf{f}_{t^\sharp}(\mathsf{ts}(t')) \ ,$$

  where we set $t' := s'$.

- *Case* $s^\sharp \xrightarrow{\ell} t^\sharp$, as $t^\sharp$ is a refinement of $s^\sharp$; $\ell = \varnothing$. By assumption s is consistent with the chosen refinement step, hence $s' \sqsubseteq t^\sharp$. Again by Lemma 8 there exists a substitution σ, such that $\mathsf{ts}(s') = \mathsf{ts}(t^\sharp)\sigma$. In sum, we obtain:

$$\mathsf{f}_{s^\sharp}(\mathsf{ts}(s')) = \mathsf{f}_{s^\sharp}(\mathsf{ts}(t^\sharp))\sigma \to_{\mathsf{rule}(s^\sharp, t^\sharp)} \mathsf{f}_{t^\sharp}(\mathsf{ts}(t^\sharp))\sigma = \mathsf{f}_{t^\sharp}(\mathsf{ts}(t')) \ ,$$

  where we again set $t' := s'$.

- *Case* $s^\sharp \xrightarrow{\ell} t^\sharp$, as $t^\sharp$ is the result of the symbolic evaluation of $s^\sharp$ and $\ell = C \neq \varnothing$. By assumption s satisfies the constraint C. More precisely, there exists a substitution σ such that $\mathsf{ts}(s') = \mathsf{ts}(s^\sharp)\sigma$ and $T \vdash \mathsf{tval}(C)\sigma$. We obtain:

$$\mathsf{f}_{s^\sharp}(\mathsf{ts}(s')) = \mathsf{f}_{s^\sharp}(\mathsf{ts}(s^\sharp))\sigma \to_{\mathsf{rule}(s^\sharp, t^\sharp)} \mathsf{f}_{t^\sharp}(\mathsf{ts}(t^\sharp))\sigma \ .$$

  Let t be defined such that $s \to_P t$. By Lemma 5 we obtain $t' \sqsubseteq t^\sharp$ and by inspection of the proof of Lemma 5 we observe that $\mathsf{ts}(t') = \mathsf{ts}(t^\sharp)\sigma$. In sum, $\mathsf{f}_{s^\sharp}(\mathsf{ts}(s')) \to_{\mathsf{rule}(s^\sharp, t^\sharp)} \mathsf{f}_{t^\sharp}(\mathsf{ts}(t'))$.

- *Case* $s^\sharp \xrightarrow{\ell} t^\sharp$, as $t^\sharp$ is the result of a $\texttt{Putfield}^\sharp$ instruction on p and there exists an address q in $s^\sharp$ that may-reaches p. By Lemma 8 we have $\mathsf{ts}(s') = \mathsf{ts}(s^\sharp)\sigma$ for some substitution σ. Let t be defined such that $s \to_P t$. Due to Lemma 5, we have $t' \sqsubseteq t^\sharp$ and thus there exists a substitution τ such that $\mathsf{ts}(t') = \mathsf{ts}^*(t^\sharp)\tau$.

  Consider the rule $\mathsf{f}_{s^\sharp}(\mathsf{ts}(s^\sharp)) \to \mathsf{f}_{t^\sharp}(\mathsf{ts}^*(t^\sharp))$. By definition address q points in $s^\sharp$ to an abstract variable x such that x occurs in $\mathsf{ts}(s^\sharp)$ and $\mathsf{ts}(t^\sharp)$. Furthermore, x is replaced by an extra variable x' in $\mathsf{ts}^*(t^\sharp)$.

Wlog., we assume that x' is the only extra variable in $\mathsf{ts}^*(t^\sharp)$. Let m be a morphism such that $m\colon s^\sharp \to s'$ and $m(q) \xrightarrow{+} m(p)$. By definition of $\mathtt{Putfield}^\sharp$, $m(p)$ and $m(q)$ exist in t' and only the part of the heap reachable from these addresses can differ in s' and t'.

In order to show the admissibility of the rewrite step $\mathsf{f}_{s^\sharp}(\mathsf{ts}(s')) \to \mathsf{f}_{t^\sharp}(\mathsf{ts}(t'))$ we define a substitution ρ such that $\mathsf{ts}(s^\sharp)\rho = \mathsf{ts}(s')$ and $\mathsf{ts}^*(t^\sharp)\rho = \mathsf{ts}(t')$. We set:

$$\rho(y) := \begin{cases} \tau(x) & \text{if } y = x' \\ \sigma(y) & \text{otherwise .} \end{cases}$$

Then $\mathsf{ts}(s^\sharp)\rho = \mathsf{ts}(s')$ by definition as $x' \notin \mathsf{Var}(s^\sharp)$. On the other hand $\mathsf{ts}^*(t^\sharp)\rho = \mathsf{ts}(t')$ follows as the definition of ρ forces the correct instantiation of x' and Lemma 5 in conjunction with Lemma 8 implies that σ and τ coincide on the portion of the heap that is not changed by the field update.

- *Case* $s^\sharp \xrightarrow{\ell} t^\sharp$, as $t^\sharp$ is the result of the symbolic evaluation of $s^\sharp$ and $\ell = \varnothing$. By convention this is equivalent to $s^\sharp \xrightarrow{\ell} t^\sharp$ and $\ell = \mathsf{true}$. Hence the third case applies. □

The next lemma emphasises that any execution step is represented by finitely many but at least one rewrite steps in $\mathcal{R}$.

Lemma 11. *Let $s^\sharp \in G$ and $s \in \mathcal{JS}$ such that $s \in \gamma(s^\sharp)$. Then $s \to_P t$ implies that there exists a state $t^\sharp \in G$ such that $t \in \gamma(t^\sharp)$ and $\mathsf{f}_{s^\sharp}(\mathsf{ts}(\beta(s))) \xrightarrow{\leqslant K} \mathsf{f}_{t^\sharp}(\mathsf{ts}(\beta(t)))$. Here K depends only on G and $\xrightarrow{\leqslant K}$ denotes at least one and at most K many rewrite steps in $\mathcal{R}$.*

Proof. The lemma follows from Lemma 7 and Lemma 10. □

We arrive at the main result of the paper.

Theorem 3. *Let $s, t \in \mathcal{JS}$. Suppose $s \to_P^* t$, where s is reachable in P from some initial state $i \in \mathcal{JS}$. We set $s' = \beta(s)$ and $t' = \beta(t)$. Let G denote the computation graph of P obtained from some initial abstract state $i^\sharp$, such that $i \in \gamma(i^\sharp)$. Then there exists $s^\sharp, t^\sharp \in G$ and a derivation $\mathsf{f}_{s^\sharp}(\mathsf{ts}(s')) \to_{\mathcal{R}}^* \mathsf{f}_{t^\sharp}(\mathsf{ts}(t'))$ such that $s \in \gamma(s^\sharp)$ and $t \in \gamma(t^\sharp)$. Furthermore, for all n: $\mathsf{rcjvm}(n) \in O(\mathsf{rctrs}(n))$.*

Proof. The existence of $s^\sharp$ follows from the correctness of abstract computation together with the construction of the computation graph. Let m denote the derivation height of the execution $s \to_P^* t$. Then by induction on m in conjunction with Lemma 11 we obtain the existence of a state $t^\sharp$ such that $t' \sqsubseteq t^\sharp$ and a derivation:

$$\mathsf{f}_{s^\sharp}(\mathsf{ts}(s')) \xrightarrow{\leqslant K \cdot m} \mathsf{f}_{t^\sharp}(\mathsf{ts}(t')) \; . \tag{1}$$

Here the constant K depends only on G. In particular we have $\mathsf{f}_s(\mathsf{ts}(s')) \to_{\mathcal{R}}^+ \mathsf{f}_t(\mathsf{ts}(t'))$ from which we conclude the first part of the theorem.

To conclude the second part, let n be arbitrary and suppose m denotes the derivation height of the execution $i \to_P^* t$, where $|i| \leqslant n$. From the proof of Lemma 9 it follows that $\|\mathsf{ts}(\beta(i))\| \leqslant |i| + 1$. We set $i' = \beta(i)$. By assumption we have $i \in \gamma(i^\sharp)$ and therefore $i' \sqsubseteq i^\sharp$. Specialising (1) to $i^\sharp$ and i' yields $\mathsf{f}_{i^\sharp}(\mathsf{ts}(i')) \xrightarrow{\leqslant K \cdot m} \mathsf{f}_{t^\sharp}(\mathsf{ts}(t'))$. Thus we obtain

$$\mathsf{rcjvm}(|i|) = m \leqslant K \cdot m \leqslant \mathsf{rctrs}(\|\mathsf{ts}(\beta(i))\|) \leqslant \mathsf{rctrs}(|i| + 1) \; . \qquad \square$$

The corollary follows directly from the previous theorem.

Corollary 3. *Let P be a program and $\mathcal{S} \subseteq \mathcal{JS}$. Suppose computation graph G is obtained from initial state $\alpha(\mathcal{S})$. Suppose cTRS $\mathcal{R}$ is obtained from G. We set $t \gg s$ iff $\mathsf{ts}(\beta(s)) = t$. Then $\gg\circ \to_{P|_\mathcal{S}} \subseteq \to_{\mathcal{R}}^+ \circ \gg$, and for all $s \in \mathcal{S}$ and $t = \mathsf{ts}(\beta(s))$ we have $t \gg s$. Furthermore, for all $s \in \mathcal{S}$ and $t \gg s$ we have $\|t\| \in O(|s|)$. Hence, $\gg$ is a complexity reflecting abstraction.*

```
class List{ List next; }

class Inits{
  void main(List ys){
    while(ys.next != null){
      List cur = ys;
        while(cur.next.next != null){
          cur = cur.next;
        }
      cur.next = null;
      }
  }
}
```

Figure 14: The `Inits` program.

It is tempting to think that the precise bound on the number of rewrite steps presented in Lemma 11 should translate to a linear simulation between JVM executions and rewrite derivations. Unfortunately this is not the case as the transformation is not termination preserving. For this consider the program illustrated in Figure 14. Here the outer loop cuts away the last cell until the initial list consists only of one cell whereas the inner loop is used to iterate through the list. It is easy to see that the main function terminates if the argument is an acyclic list. Since variables `ys` and `cur` share during iteration, the proposed transformation introduces a fresh variable for the `next` field of the initial argument `ys` when performing the `Putfield` instruction. Termination of the resulting rewrite system can not be shown any more. However *non-termination preservation* follows as an easy corollary of Theorem 3.

Corollary 4. *The computation graph method, that is the transformation from a given JBC program P to a cTRS $\mathcal{R}$ is non-termination preserving.*

Proof. Suppose there exists an infinite run in P, but $\mathcal{R}$ is terminating. Let i be some initial state i of P. By Theorem 3 there exists a state t such that $i \to_P^* t$ and $\mathsf{f}_{i^\natural}(\mathsf{ts}(i')) \to_{\mathcal{R}}^+ \mathsf{f}_{t^\natural}(\mathsf{ts}(t'))$, where $i \in \gamma(i^\natural)$, $i' = \beta(i)$, $t \in \gamma(t^\natural)$, and $t' = \beta(t)$. Furthermore, as $\mathcal{R}$ is terminating we can assume $\mathsf{f}_{t^\natural}(\mathsf{ts}(t'))$ is in normal form. However, as t' is non-terminating, there exists a successor, thus Lemma 11 implies that $\mathsf{f}_{t^\natural}(\mathsf{ts}(t'))$ cannot be in normal form. Contradiction. □

8. Automation

We have implemented and integrated a prototype of our proposed transformation in the T$_\mathsf{C}$T^3 framework [20, 21]. Moreover we have incorporated techniques to analyse the complexity of cTRSs, thus providing a fully automated complexity analysis of JBC programs. In the following we discuss some implementation details.

Modularity of our transformation allows us to make use of additional heap shape domains. In particular we make use of may-alias, may-reaches and acyclicity analysis. We clarify how these properties can be safely approximated and how they are integrated into our transformation.

Type Analysis. The type analysis abstracts values of the program environment to types. In particular it provides an upper bound with respect to $(\mathsf{types}(P), \leqslant_{\mathsf{type}})$ (cf. Definition 5) on stack and local variables. The analysis is based on the well-typed analysis of JBC [15] and is used in the analyses mentioned below to restrict the abstract domain based on the typing information. For example, assume that the analysis infers that the types of variables `x` and `y` is cn and cn', respectively. If neither $cn \preceq cn'$ nor $cn \preceq cn'$ holds then the elements bound to `x` and `y` can not alias.

[3] `http://cl-informatik.uibk.ac.at/software/tct`

Aliasing Analysis. Aliasing information is used when evaluating `Putfield`$^\sharp$ or `CmpEq`$^\sharp$ to check whether two addresses in the abstract heap may alias. We use the information given in the abstract state together with the sharing analysis introduced in [17] to approximate the desired property. Given two variables `x` and `y`, the abstract domain in [17] approximates whether the elements bound to `x` and `y` share a common part in the heap. We can use this information to determine whether two addresses p and q do not alias by considering the addresses reachable from `x` and `y` in the state graph. More precisely, if there exists no pair `x`, `y` in the sharing domain such that p is reachable from `x` and q is reachable from `y` in the state graph, then p and q do not share and therefore they do not alias.

Reachability Analysis. Reachability information is used when transforming computation graphs into cTRSs to accommodate for possible side-effects of `Putfield`. We have incorporated the reachability analysis introduced in [19]. Given two variables `x` and `y` the abstract domain in [19] approximates whether there exists a path (of at least length 1) in the graph representation of the heap from the object bound to `x` to the object bound to `y`. Akin to the aliasing analysis we can lift the approximation to addresses of our abstract state by considering the addresses reachable from two variables in the state graph.

Acyclicity Analysis. Acyclicity information is used in the term abstraction of bytecode states. Only acyclic objects can be represented as finite terms. Possible cyclic objects are approximated by introducing fresh variables. The domain provided in [19] approximates whether there exists a cyclic path in the heap graph reachable from the element bound to variable `x`. We can determine that an address p is definitely acyclic if there exists a variable `x` in the approximation and a path from `x` to p in the state graph.

Complexity Analysis of cTRS. Finally to complete the analysis, we need to run a complexity tool on cTRSs. In [39] support for verifying properties such as confluence, quasi-reductivity and termination as well as equational reasoning for logical constrained term rewrite systems is presented. The applied techniques however are not suitable for complexity analysis. Our prototype covers simplification techniques, polynomial interpretations for cTRSs and complexity reflecting transformations to (standard) TRSs and *integer transition systems* [40] (*ITSs* for short). We apply *inlining* and *instantiation* [13] to simplify the original problem. Inlining is used to group the effect of several bytecode instructions together and is performed by combining rules using the unifier of right-hand sides and left-hand sides. Instantiation is used to refine the problem by performing case analyses on terms. In particular we substitute variables of sort `bool` with true and false, respectively.

Example 15. Figure 15 illustrates a program with boxed integers that makes use of sharing together with the simplified cTRS problem that is obtained after instantiating variables of sort `bool` and repetitive inlining. Our tool is able to synthesize a (linear) polynomial interpretation which implies a linear runtime bound on the program.

```
class I { int val; }

class Boxed {
  void main(I x, I z){
    I y = x;
    while (x.val < z.val){
      x.val = x.val - 1;
      y.val = y.val + 2;
    }
  }
}
```

$$\mathsf{f}_m(i_1, i_2) \to \mathsf{f}_w(i_1, i_2, i_1)$$
$$\mathsf{f}_w(\mathsf{I}(i_1), \mathsf{I}(i_2), \mathsf{I}(i_1)) \to \mathsf{f}_w(\mathsf{I}(i_1+1), \mathsf{I}(i_2), \mathsf{I}(i_1+1)) \quad [\![i_1 < i_2]\!]$$
$$\mathsf{f}_w(\mathsf{I}(i_1), \mathsf{I}(i_2), \mathsf{I}(i_1)) \to \mathsf{f}_w(\mathsf{I}(i_1), \mathsf{I}(i_2), \mathsf{I}(i_1)) \quad [\![i_1 \geqslant i_2]\!]$$

Figure 15: The `Boxed` program.

Transformation to TRS and ITS. Our prototype provides only some basic techniques to analyse cTRS directly. To make use of existing techniques and tools we apply complexity reflecting transformations from cTRSs to TRSs and ITSs. At first observe that defined symbols on the right-hand side of the cTRSs obtained via the transformation only occur at root positions, therefore we can apply *argument filtering*, ie. we can restrict arguments of terms to specified positions. The application of an argument filter on the cTRSs obtained by our transformation is sound, ie. complexity reflecting, but not complete. Moreover recall that we associate sorts to symbols and variables, thus providing a simple criterion for argument filtering. To transform a cTRS to a TRS we remove all constraints and apply an argument filter to restrict to argument positions that are not theory symbols or variables of sort `int` and `bool`. Additionally, we apply an argument filter to restrict to argument positions that contain no fresh variables on the right-hand sides of the rules. The resulting system is a (standard) TRS and can then be analysed using existing techniques for term rewriting.

Example 16. Figure 16 depicts an imperative version of John McCarthy's Flatten specialised to a list of trees storing integers. We assume that the argument of the `main` function is tree-shaped and `this` is initially not `null`. The complexity tool T$_C$T is able to show that the system resulting from the TRS transformation has linear runtime complexity.

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

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

class TreeList{
  TreeList next;
  Tree value;
}
```

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

Figure 16: The `Flatten` program.

Akin to the transformation to TRSs, we obtain ITSs by applying an argument filter on theory symbols and variables of sort `bool` and `int`.

Example 17. Figure 17 illustrates a program for printing Floyd's triangle and the ITS obtained by the transformation. The complexity tool T$_C$T provides an (optimal) quadratic bound on the runtime complexity.

Our prototype implements only some basic techniques that work on cTRSs directly. Both `Flatten` and `FloydsTriangle` can not be analysed without further transformations to TRSs and ITSs. On the other hand, the `Boxed` program can not be analysed via our proposed transformations to TRSs and ITSs. It is subject to future work to investigate more techniques for cTRSs and how to effectively combine techniques of TRSs and ITSs.

```
class FloydsTriangle{
  void main(int n){
    int c,d,num = 0;
    for(c = 1; c <= n; c = c + 1){
      for(d = 1; d <= c; d = d + 1){
        //print(num+" ");
        num = num + 1;
      } //println();
    }
  }
}
```

$$
\begin{aligned}
&\mathsf{f}_i(n, num, c, d) \to \mathsf{f}_{w1}(n, 0, 1, num) && \\
&\mathsf{f}_{w1}(n, num, c, d) \to \mathsf{f}_{w2}(n, num, c, 1) && [\![c \leqslant n]\!] \\
&\mathsf{f}_{w2}(n, num, c, d) \to \mathsf{f}_{w2}(n, num + 1, c, d + 1) && [\![d \leqslant c]\!] \\
&\mathsf{f}_{w2}(n, num, c, d) \to \mathsf{f}_{w1}(n, num, c + 1, d) && [\![d > c]\!] \\
&\mathsf{f}_{w1}(n, num, c, d) \to \mathsf{f}_{e}(n, num, c, d) && [\![c > n]\!]
\end{aligned}
$$

Figure 17: The `FloydsTriangle` program.

Experimental Evaluation. We compare our prototype implementation with the COSTA[4] tool. We have selected a representative set of examples that fits in the scope of our work from the TPDB[5] benchmark and the literature [40–42]. Amongst others, it contains programs operating on standard data structures and integers as well as programs using inheritance and dynamic dispatch. Programs are written in Java source. The Jinja bytecode programs have been generated by following the compiler implementation in [15] closely, and the Java bytecode programs for COSTA have been obtained by the standard Java compiler of the OpenJDK8 platform. For the analysis we assume that the input is tree-shaped. Table 1 illustrates the results of our experiment. We provide detailed information on the evaluation online.[6] With respect to the selected example set our prototype is better on programs manipulating tree data structures. However, a more rigorous assessment is necessary to draw more precise conclusions.

	T$_C$T	COSTA
Boxed	$O(n)$	?
Convert	?	?
DivMinus	$O(n)$	$O(n)$
FloydsTriangle	$O(n^2)$	$O(n^2)$
Increment	$O(n)$	$O(n)$
IntIncrease	$O(n^2)$	?
ListAppendAppend	?	$O(n)$
ListAppend	$O(n)$	$O(n)$
ListDuplicate	$O(n)$	?
ListIncrease	?	$O(n^2)$
ListIntersperse	$O(n)$	?
ListReverse1	?	?
ListReverse2	$O(n)$	?
MatrixAddition	$O(n)$	$O(n^2)$
NestedListIterate	$O(n^2)$	$O(n^2)$
SortedListInsert	$O(n)$	$O(n)$
TreeCopy	$O(n)$	?
TreeFlatten	$O(n)$	?
TreeMirror	?	?
TreeTraverse	$O(n^2)$	?
TreeTraverseRecursive	?	$O(2^n)$

Table 1: Summary of experimental evaluation.

9. Related Work

Resource Analysis of Transition Systems. Zuleger et al. [43] employ *size-change abstraction* to analyse the runtime complexity of C programs automatically. In connection with pathwise analysis and *contextualisation* size-change abstraction yields a powerful analysis. The paper [44] introduces a simple abstraction on programs together with loop path extraction to represent different control flows in the original program. Lexicographic ranking functions are used to provide an amortised bound analysis. The approach has been implemented in the tool LOOPUS[7].

[4] https://costa.ls.fi.upm.es
[5] http://www.termination-portal.org/wiki/TPDB
[6] http://cl-informatik.uibk.ac.at/software/tct/experiments/16dice
[7] http://forsyte.at/software/loopus

The KoAT[8] tool combines well-known results and techniques from the literature to provide an efficient modular analysis [40]. Global timebounds are established by composing local timebounds with global sizebounds, and global sizebounds are established by composing global timebounds with local sizebounds. Timebounds and sizebounds are incrementally approximated and improved, thus providing an efficient modular analysis.

In [45] cost equations are used to provide a uniform representation of loop constructs and recursion. The cost equations are iteratively refined using invariant analysis together with an approximation of feasible phases of strongly connected components. Bounds are computed bottom-up taking dependencies of different phases into account. The approach has been implemented in the tool CoFloCo[9].

Our approach extends the use of transition systems by cTRSs, which theoretically form a strict extension. Furthermore, as our methods are rooted in rewriting we are not limited to the power of invariant generation tools.

Resource Analysis of Object-Oriented Programming Languages. Termination behaviour and complexity of JBC programs is studied by Albert et al. in [42]. The approach employs program transformations to *constraint logic programs* and has been successfully implemented in the COSTA[4] tool; it often allows precise bounds on the resource usage and is not restricted to runtime complexity. A theoretical limitation of the work is the focus on a path-length abstraction of the heap, which does not provide the same detail as the term-based abstraction presented here.

In [41] Atkey presents an approach for amortised resource analysis of imperative languages by embedding resource information within separation logic. The techniques developed have been implemented for the resource analysis of Java bytecode [46]. The approach relies on explicitly defined inductive predicates that represents the shape of the objects. Furthermore, no techniques to handle loops depending on numeric quantities are provided. The motivating Frying Pan example in [41] is beyond the scope of our analysis due to the abstraction of cyclic data. Note however that in general we can deal with cyclic data as long as it is independent from the termination behaviour of the program.

Term Rewriting. Our work was inspired by Panitz and Schmidt-Schauß original observation that term-based abstraction can provide powerful termination analysis [47]. Furthermore, we got inspiration from the ongoing quest to establish non-termination preserving transformations from JBC programs to integer term rewrite system [27, 29–31]. The approach has been implemented in AProVE[10] and has shown significant power in comparison to dedicated termination tools for JBC programs [42, 48]. Comparing our work with earlier results reported for the termination graph method [27, 29] we see that a similar transformation from graphs to rewrite systems is employed. On the other hand in Otto et al. [27] (and follow-up work) sharing is dealt with explicitly, while in our context sharing is always allowed if not stated otherwise. Furthermore, we do not rely on heuristics to obtain a finite graph representation but prove finiteness and totality of our transformation. We remark that our results also show that the transformation proposed by Otto et al. is complexity reflecting.

In [49] reachability rules together with a language-independent proof system to verify reachability properties are provided. Reachability rules are rewrite rules with matching logic patterns [50] and provide a formal and uniform way to express the operational semantics of a programming language and reachability properties of a program. Reachability rules can incorporate and reason about term-based abstractions of state components [51]. But the verification of properties depends on domain knowledge and invariants which make it difficult to integrate it in an automated analysis.

Program Analysis. At the beginning of Section 6 we provided a short intuition how computation graphs relate to more traditional data flow analyses. We want to elaborate this relation in more detail. The computation graph method is conceptually simple and excels in analysing unstructured code such as the Jinja bytecode.

[8] `https://github.com/s-falke/kittel-koat`

[9] `https://www.se.tu-darmstadt.de/se/group-members/antonio-flores-montoya/cofloco`

[10] `http://aprove.informatik.rwth-aachen.de`

For example, exceptional control flow could be easily integrated. The abstraction however has to incorporate enough information to determine the control flow. In our case we used class names, method names and program counters to determine the program location. Since the computation graph expands dynamically depending on the abstraction rather than operating on a fixed control flow, precision can be improved using refinements and different data flows can be represented via branches in the computation graph. To ensure termination and providing a scalable analysis, strategies for joining and widening of abstractions have to be incorporated. In the present work abstractions resulting from conditional branches and method invocations are not joined and the abstraction is refined for the `Putfield`$^\sharp$ instructions. This is in contrast to many other analyses, as presented for example in [17–19], where one operates on a fixed control flow obtained from the program. Though there is a tight connection. Informally, the computation graph method lifts a reachable states abstraction as present for example in [17–19] to a trace partitioning abstract domain [52] at the cost of a more expensive analysis.

10. Conclusion and Future Work

In this paper we show how the runtime complexity of *Jinja bytecode* (*JBC*) programs can be analysed fully automatically by a transformation to *(constraint) term rewrite systems* and *integer transition systems*, the complexity of which can then be automatically verified by existing complexity tools.

We exploit a *term-based* abstraction of programs within the *abstract interpretation* framework [16] thus linking traditional results in program analysis to results independently obtained in rewriting. The proposed transformation encompasses two stages. For the first stage we perform a combined control and data flow analysis by evaluating program states symbolically, which is shown to yield a finite representation of all execution paths of the given program through a graph, dubbed *computation graph*. Note that *widening* of abstract states is carefully controlled so that the representation of JBC executions is provably finite.

In the second stage we encode the (finite) computation graph as a *constraint term rewrite system*. This is done while carefully analysing complexity preservation and reflection of the employed transformations such that the complexity of the obtained term rewrite system reflects on the complexity of the initial program.

In order to test the applicability of the proposed method we have implemented the transformation from JBC programs to constraint rewrite systems and techniques to assess the complexity of constraint rewrite systems automaticallly. Furthermore, in order to make use of other existing tools we established further transformations from constraint rewrite systems to (standard) term rewrite systems and integer transitions systems. The full transformation pipeline has been implemented within T$_\subset$T [20, 21]. Our prototype implementation is able to fully automatically verify *linear* runtime complexity of a standard encoding of tree flattening, a motivating example of early work on non-termination preserving transformations from JBC to term rewrite systems, cf. [27]. A simple corollary of our results is that the transformation proposed in the literature is complexity reflecting.

As we have based our transformation quite principally on the abstract interpretation framework, it allows for an easy incorporation of the existing wealth of results on shape analysis present in the literature and thus improves upon the modularity of the transformational approach. Furthermore we emphasise that this approach to resource analysis is not limited to *polynomial* bounds on the runtime complexity. For example dedicated techniques exist that establish *exponential* upper bounds on the runtime complexity of term rewrite systems [53, 54]. Thus our result directly also yields resource analysis for JBC programs beyond polynomial upper bounds.

Future work will be dedicated towards new methods for complexity analysis of constraint term rewrite systems and combining techniques for term rewriting systems and integer transitions systems.

Acknowledgement

We are grateful to the anonymous reviewers whose comments greatly helped us to improve the presentation of the paper. This work was partially supported by FWF (Austrian Science Fund) project number P 25781-N15 and Draper Labs, project number 15-B13.

References

[1] A. Middeldorp, G. Moser, F. Neurauter, J. Waldmann, H. Zankl, Joint Spectral Radius Theory for Automated Complexity Analysis of Rewrite Systems, in: Proc. 4th CAI, vol. 6742 of *LNCS*, 1–20, 2011.

[2] M. Avanzini, G. Moser, Polynomial Path Orders, LMCS 9 (4).

[3] J. Waldmann, Matrix Interpretations on Polyhedral Domains, in: Proc. of 26th RTA, 318–333, 2015.

[4] M. Avanzini, N. Eguchi, G. Moser, New Order-theoretic Characterisation of the Polytime Computable Functions, TCS 585 (2015) 3–24.

[5] N. Hirokawa, G. Moser, Automated Complexity Analysis Based on the Dependency Pair Method, in: Proc. 4th IJCAR, vol. 5195 of *LNCS*, 364–380, 2008.

[6] N. Hirokawa, G. Moser, Complexity, Graphs, and the Dependency Pair Method, in: Proc. of 15th LPAR, 652–666, 2008.

[7] L. Noschinski, F. Emmes, J. Giesl, Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs, JAR 51 (1) (2013) 27–56.

[8] H. Zankl, M. Korp, Modular Complexity Analysis via Relative Complexity, in: Proc. 21th RTA, vol. 6 of *LIPIcs*, 385–400, 2010.

[9] M. Avanzini, G. Moser, A Combination Framework for Complexity, IC 248 (2016) 22–55.

[10] F. Frohn, J. Giesl, J. Hensel, C. Aschermann, T. Ströder, Inferring Lower Bounds for Runtime Complexity, in: Proc. of 26th RTA, LIPIcs, 334–349, 2015.

[11] G. Moser, Proof Theory at Work: Complexity Analysis of Term Rewrite Systems, CoRR abs/0907.5527, Habilitation Thesis.

[12] J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, C. Fuhs, Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs, in: Proc. of 14th PPDP, ACM, 1–12, 2012.

[13] M. Avanzini, U. D. Lago, G. Moser, Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order, in: Proc. 20th ICFP, ACM, 152–164, 2015.

[14] R. Stärk, J. Schmid, E. Börger, Java and the Java Virtual Machine: Definition, Verification, Validation, Springer, 2001.

[15] G. Klein, T-Nipkow, A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler, TOPLAS 28 (4) (2006) 619–695.

[16] P. Cousot, R. Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: Proc. 4th POPL, 238–252, 1977.

[17] S. Secci, F. Spoto, Pair-Sharing Analysis of Object-Oriented Programs, in: Proc. 12th SAS, vol. 3672 of *LNCS*, 320–335, 2005.

[18] S. Rossignoli, F. Spoto, Detecting Non-cyclicity by Abstract Compilation into Boolean Functions, in: Proc. 7th VMCAI, vol. 3855 of *LNCS*, 95–110, 2006.

[19] S. Genaim, D. Zanardini, Reachability-based Acyclicity Analysis by Abstract Interpretation, TCS 474 (2013) 60–79.

[20] M. Avanzini, G. Moser, Tyrolean Complexity Tool: Features and Usage, in: Proc. of 24th RTA, vol. 21 of *LIPIcs*, 71–80, 2013.

[21] M. Avanzini, G. Moser, M. Schaper, TcT: Tyrolean Complexity Tool, in: Proc. 22th TACAS, LNCS, 407–423, 2016.

[22] A. Lochbihler, Jinja with Threads, Archive of Formal Proofs 2007.

[23] A. Lochbihler, Verifying a Compiler for Java Threads, in: Proc. 19th ESOP, vol. 6012 of *LNCS*, 427–447, 2010.

[24] M. Schaper, A Complexity Preserving Transformation from Jinja Bytecode to Rewrite Systems, Master's thesis, University of Innsbruck, Austria, `http://cl-informatik.uibk.ac.at/users/c7031025/publications/masterthesis.pdf`, 2014.

[25] F. Frohn, M. Brockschmidt, J. Giesl, Automated Inference of Upper Complexity Bounds for Java Programs, in: Proceedings of the 15th International Workshop on Termination, 2016.

[26] J. Hoffmann, K. Aehlig, M. Hofmann, Multivariate amortized resource analysis, TOPLAS 34 (3) (2012) 14.

[27] C. Otto, M. Brockschmidt, C. v. Essen, J. Giesl, Automated Termination Analysis of Java Bytecode by Term Rewriting, in: Proc. 21th RTA, 259–276, 2010.

[28] TeReSe, Term Rewriting Systems, vol. 55 of *Cambridge Tracks in Theoretical Computer Science*, Cambridge University Press, 2003.

[29] M. Brockschmidt, C. Otto, C. von Essen, J. Giesl, Termination Graphs for Java Bytecode, in: Verification, Induction, Termination Analysis, vol. 6463 of *LNCS*, 17–37, 2010.

[30] M. Brockschmidt, C. Otto, J. Giesl, Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting, in: Proc. 22nd RTA, LIPIcs, 155–170, 2011.

[31] M. Brockschmidt, R. Musiol, C. Otto, J. Giesl, Automated Termination Proofs for Java Bytecode with Cyclic Data, in: Proc. 24th CAV, vol. 7358 of *LNCS*, 105–122, 2012.

[32] M. Sagiv, T. Reps, R. Wilhelm, Parametric Shape Analysis via 3-valued Logic, in: Proc. 26th POPL, ACM, 105–118, 1999.

[33] F. Nielson, H. Nielson, C. Hankin, Principles of Program Analysis, Springer, 2005.

[34] S. Falke, D. Kapur, A Term Rewriting Approach to the Automated Termination Analysis of Imperative Programs, in: Proc. 22nd CADE, vol. 5663 of *LNCS*, 277–293, 2009.

[35] S. Falke, D. Kapur, C. Sinz, Termination Analysis of C Programs Using Compiler Intermediate Languages, in: Proc. 22nd RTA, vol. 10 of *LIPIcs*, 41–50, 2011.

[36] T. Sakata, N. Nishida, T. Sakabe, On Proving Termination of Constrained Term Rewrite Systems by Eliminating Edges from Dependency Graphs, in: Proc. of 20th WFLP, vol. 6816 of *LNCS*, 138–155, 2011.

[37] C. Kop, N. Nishida, Term Rewriting with Logical Constraints, in: Proc. 9th FroCos, vol. 8152 of *LNCS*, Springer, 343–358, 2013.

[38] C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, S. Falke, Proving Termination of Integer Term Rewriting, in: Proc. 20th RTA, vol. 5595 of *LNCS*, 32–47, 2009.
[39] C. Kop, N. Nishida, Constrained Term Rewriting tooL, in: Proc. 20th LPAR, LNCS, 549–557, 2015.
[40] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, J. Giesl, Alternating Runtime and Size Complexity Analysis of Integer Programs, in: Proc. 20th TACAS, 140–155, 2014.
[41] R. Atkey, Amortised Resource Analysis with Separation Logic, in: Proc. 19th ESOP, vol. 6012 of *LNCS*, Springer, 85–103, 2010.
[42] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost Analysis of Object-Oriented Bytecode Programs, TCS 413 (1) (2012) 142–159.
[43] F. Zuleger, S. Gulwani, M. Sinn, H. Veith, Bound Analysis of Imperative Programs with the Size-Change Abstraction, in: Proc. 18th SAS, vol. 6887 of *LNCS*, 280–297, 2011.
[44] M. Sinn, F. Zuleger, H. Veith, A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis, in: Proc. 26th CAV, 745–761, 2014.
[45] A. Flores-Montoya, R. Hähnle, Resource Analysis of Complex Programs with Cost Equations, in: Proc. 12th APLAS, 275–295, 2014.
[46] D. Fenacci, K. MacKenzie, Static Resource Analysis for Java Bytecode Using Amortisation and Separation Logic, ENTCS 279 (2011) 19 – 32, ISSN 1571-0661, proc. 6th BYTECODE.
[47] S. E. Panitz, M. Schmidt-Schauß, TEA: Automatically Proving Termination of Programs in a Non-Strict Higher-Order Functional Language, in: Proc. 4th SAS, 345–360, 1997.
[48] F. Spoto, F. Mesnard, É. Payet, A Termination Analyzer for Java Bytecode based on Path-length, TOPLAS 32 (3).
[49] G. Roşu, A.Stefănescu, Checking Reachability using Matching Logic, in: Proc. 27th OOPSLA, ACM, 555–574, 2012.
[50] G. Roşu, C. Ellison, W. Schulte, Matching Logic: An Alternative to Hoare/Floyd Logic, in: Proc. 13th AMAST, vol. 6486 of *LNCS*, 142–162, 2010.
[51] A.Stefănescu, MatchC: A Matching Logic Reachability Verifier Using the K Framework, ENTCS 304 (2014) 183–198.
[52] X. Rival, L. Mauborgne, The Trace Partitioning Abstract Domain, TOPLAS 29 (5).
[53] J. Endrullis, J. Waldmann, H. Zantema, Matrix Interpretations for Proving Termination of Term Rewriting, JAR 40 (2-3) (2008) 195–220.
[54] M. Avanzini, N. Eguchi, G. Moser, A Path Order for Rewrite Systems that Compute Exponential Time Functions, in: Proc. of 22nd RTA, vol. 10 of *LIPIcs*, Dagstuhl, 123–138, 2011.