A Complexity Preserving Transformation from Jinja Bytecode to Rewrite Systems*

Georg Moser and Michael Schaper

Institute of Computer Science
University of Innsbruck, Austria
{georg.moser,michael.schaper}@uibk.ac.at

Abstract

We revisit known transformations from object-oriented bytecode programs to rewrite systems from the viewpoint of runtime complexity. Suitably generalising the constructions proposed in the literature, we define an alternative representation of Jinja bytecode (JBC) executions as *computation graphs* from which we obtain a representation of JBC executions as *constrained rewrite systems*. We show that the transformation is *complexity preserving*. We restrict to non-recursive methods and make use of heap shape pre-analyses.

1998 ACM Subject Classification F.3.2, F.4.1, F.4.2

Keywords and phrases Jinja Bytecode, Program Analysis, Runtime Complexity, Term Rewriting

Digital Object Identifier 10.4230/OASIcs.xxx.yyy.p

1 Introduction

In this work we study the automatic runtime complexity analysis of Jinja bytecode, an object-oriented bytecode language, by means of a transformation to *constrained term rewrite systems (cTRSs)*. Here, cTRSs are defined as an extension of term rewrite systems (TRSs) that incorporate the theory of Presburger arithmetic. TRSs have been used before to prove termination of programs in functional [9], imperative [4] and object-oriented [8, 2] languages. TRSs are a reasonable cost model [1] and several methods have been developed to infer upper bounds of TRSs automatically [6]. This motivates to extend existing approaches to complexity analysis. Based on [8, 2], we propose an alternative representation of abstract states. We relate our approach to standard techniques from static program analysis, in particular *abstract interpretation* [3], and show that the transformation to cTRS is *complexity preserving*. This extended abstract is an excerpt from a report currently in progress [7].

2 Concrete Bytecode Domain

We analyse Jinja bytecode (JBC) programs. Jinja is a Java like language that exhibits its core features and is formally specified and verified [5]. We expect the reader to be familiar with Java. A *Jinja value* is either a Boolean, an integer, the dummy value unit, the reference null, or an address. Instances of user-defined data types are termed *objects*. Figure 1 illustrates a bytecode program that appends a list to an existing list; `L` is a list class with field `n` of type `L`. Bytecode is executed on the Jinja virtual machine (JVM). A *JVM state* is a pair consisting of a *heap* and a list of *frames*. A heap is a mapping from *addresses* to *objects* and a frame consists of a *register* and an *operand stack*. A state can be naturally represented as

* This work is partially supported by FWF (Austrian Science Fund) project P25781.

Conference/workshop/symposium title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–4

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

a directed graph, termed *state graph*, where node labels (denoted $L(v)$) are stack (register) indices, non-address values or class identifiers and edges are empty or field identifiers. In Jinja sharing is only possible via addresses. We define the *size of a value* v as the absolute value of v, if v is an integer, and 1 otherwise. The *size of a state* s (denoted $|s|$) is defined on a *per-reference* basis that unravels sharing. Hence, a value bound to a register (stack) index or an object field is accounted for each simple access path in the state graph of s, starting from the register (stack) indices. Let $\mathcal{JS}$ denote the set of states of program P, and let $s, t \in \mathcal{JS}$. The one-step relation of P is denoted $s \to_P t$, and an evaluation of s to t 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*. We define the *collecting semantics* (denoted f^*) on $\mathcal{P}(\mathcal{JS})$ as the set extension of the one-step relation to sets.

▶ **Definition 1.** The *runtime* of $s \to_P^* t$ is the number of single-step executions of the evaluation from s to t. Let $\mathcal{S} \subseteq \mathcal{JS}$. The *runtime complexity* of P is $\mathsf{rcjvm}(n) := \mathsf{max}\{m \mid i \to_P^* t$ such that the runtime is m, $i \in \mathcal{S}$ and $|i| \leqslant n\}$.

```
00: Load 0
01: Store 2
02: Load 2
03: Getfield n L
04: Push null
05: CmpNeq
06: IfFalse 5
07: Load 2
08: Getfield n L
09: Store 2
10: Goto -8
11: Load 2
12: Load 1
13: Putfield n L
14: Push unit
15: Return
```

Figure 1 List append.

3 Abstract Bytecode Domain

We introduce *abstract states* as generalisations of JVM states. Abstract states represent sets of JVM states and are similar defined, but heap and frames may contain (sorted) variables: *bool* (*int*) represents an undefined Boolean (integer) value, and *cn* represents either null or an instance of class cn', where cn' is a (not necessarily proper) subclass of *cn*. We employ an implicit representation of sharing in the abstract heap, and incorporate annotations $p \neq q \in iu$ to disallow aliasing of addresses p and q in the represented states. The set of abstract states is denoted $\mathcal{AS} \supseteq \{\top, \bot\}$. Elements of $\mathcal{AS}$ are usually indicated with $\natural$. We define a preorder $\preccurlyeq$ on (abstract) non-address values, class identifiers and class variables. We have $v \preccurlyeq w$, if either (1) $v = w$; (2) $v =$ unit; (3) $v =$ null and w is a class variable *cn*; (4) v is a Boolean (integer) and $w =$ *bool* (*int*); or (5) $v = cn'$, w a class variable *cn* and cn' is a subclass of *cn*. We exploit $\preccurlyeq$ and the graph representation to define a partial order $\sqsubseteq$ on abstract states. We only provide an informal description; for details please refer to [7]. Let $S^\natural$ and $T^\natural$ be state graphs of states $s^\natural$ and $t^\natural$. We have $s^\natural \sqsubseteq t^\natural$, if there exists a (graph-)morphism m from $T^\natural$ to $S^\natural$ such that (1) all stack (register) indices of $T^\natural$ are mapped to the same stack (register) indices in $S^\natural$; (2) we have $m(u) \in S^\natural$ and $L_{S^\natural}(m(u)) \preccurlyeq L_{T^\natural}(u)$, for all $u \in T^\natural$; and (3) m respects edge labels (ie., the field identifiers), and annotations iu. The limit cases are handled as usual. For a suitable join operation $\mathcal{AS} := (\mathcal{AS}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ is a complete lattice.

▶ **Definition 2.** Let $s = (heap, frms) \in \mathcal{JS}$. We define $\beta\colon \mathcal{JS} \to \mathcal{AS}$. Suppose $\mathsf{dom}(heap) = \{p_1, \ldots, p_n\}$. Define iu such that $p_i \neq p_j \in iu$ for all different i, j. Then $\beta(s) = (heap, frms, iu)$. Let $\alpha\colon \mathcal{P}(\mathcal{JS}) \to \mathcal{AS}$ and $\gamma\colon \mathcal{AS} \to \mathcal{P}(\mathcal{JS})$ be: $\alpha(\mathcal{S}) := \bigsqcup\{\beta(s) \mid s \in \mathcal{S}\}$ and $\gamma(s^\natural) := \{s \in \mathcal{S} \mid \beta(s) \sqsubseteq s^\natural\}$. Then $(\mathcal{P}(\mathcal{JS}), \alpha, \gamma, \mathcal{AS})$ is a Galois connection [3].

We propose *computation graphs* as finite representations of $\mathcal{AS}$, abstracting $\mathcal{JS}$. A computation graph is a finite control flow graph, in which nodes are abstract states, and is obtained by expanding nodes via the *abstract semantics* and merging nodes with equal program location. An abstract computation consists of finitely many *refinement* steps and an *abstract evaluation* step. An evaluation step mimics the semantics of the JVM instructions closely. In case of an (abstract) integer and Boolean operation we label the edge with a constraint

representing the operation. Refinement steps are performed if no evaluation step can be executed. This happens, if the instruction is either (1) a conditional jump and the top of the stack is a Boolean variable; (2) a field access, field update, or a method invocation and the address points to a class variable; or (3) a field update, and the address may-alias with another address in the heap. For (1), we consider states where the variable is substituted with Boolean values. For (2), we consider states where the variable is substituted with null and (most general) instances of all subclasses. For (3), we consider states where we set the addresses equal and unequal. Figure 2 illustrates the computation graph of `append`, obtained under the assumption that all variables are acyclic, and do not share at the beginning. We refine states by sharing and acyclicity facts, as defined for example in [10]. Here ϵ denotes the empty stack; S is obtained from a join operation; C depicts a refinement; annotations are left out. Note that due to (3) all side-effects in the visible part of the heap are accounted to. Correctness requires that the abstract semantics safely approximates the concrete semantics, ie., $f^*(\gamma(s^\natural)) \subseteq \gamma(f^\natural(s^\natural))$. We obtain following result:

▶ **Theorem 3.** *Let $i, t \in \mathcal{JS}$. Suppose $i \to_P^* t$ with runtime m. Let G be the computation graph of P obtained from initial state $i^\natural$ such that $i \in \gamma(i^\natural)$. Then there exists an abstraction $t^\natural$ of t and m' such that $i^\natural \xrightarrow{m'}_G t^\natural$ holds, for $m \leqslant m' \leqslant K \cdot m$. Here $K \in \mathbb{N}$ only depends on G.*

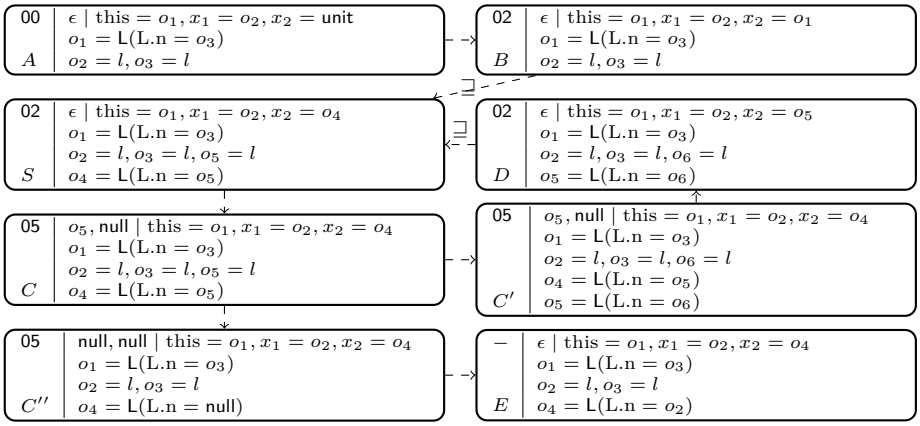

Figure 2 The (incomplete) computation graph of append.

4 Abstract Term Domain

We present the transformation from computation graphs to *constrained term rewrite systems*. Our definition is a variation of cTRSs as for example in [4]. We are interested in cTRS over the theory T of Presburger arithmetic (PA). We have $T \vdash C$, if all ground instances of constraint C are valid in PA. If there exists a substitution σ, such that $T \vdash C\sigma$, then C is *satisfiable*. Let C be a formula over theory symbols and (sorted) variables. 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 constrained 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 $=_T$ is unification modulo T. A cTRS $\mathcal{R}$ is called *terminating*, if the relation $\to_\mathcal{R}$ is well-founded. We adapt the runtime complexity wrt. a standard TRS suitable for cTRS $\mathcal{R}$. The size of a term t (denoted $\|t\|$) is (1) 1, if t is a variable; (2) the absolute value of t, if t is an integer; and (3) $1 + \sum_{i=1}^n \|t_i\|$, if $t = f(t_1, \ldots, t_n)$. The *derivation height* of a term t (denoted $\mathsf{dh}(t)$) wrt. $\mathcal{R}$ is the maximal length of a derivation starting in t.

▶ **Definition 4.** We define the *runtime complexity* wrt. $\mathcal{R}$ as follows: $\mathsf{rctrs}(n) := \mathsf{max}\{\mathsf{dh}(t) \mid t \text{ is } \textit{basic} \text{ and } \|t\| \leqslant n\}$, where $t = f(t_1, \ldots, t_k)$ is *basic* if f is defined, and terms t_i are only built over constructor, theory symbols, and variables.

To represent program states as terms over $\mathcal{F}$ we proceed as follows: We collect the values bound to stack and register indices in a list (denoted $\mathsf{ts}(s)$). A value v is (1) v, if v is a non-address value; (2) cn', if the value bound to v is possible cyclic, and cn' is a fresh class variable; (3) cn, if v is a class variable cn; (4) $cn(fields)$, ie., the term representation of an object cn, if v is bound to an acyclic instance. Let G be a computation graph. For any state $s^\natural$ in G we introduce a new function symbol $\mathsf{f}_{s^\natural}$, and for each edge $s^\natural \xrightarrow{\ell} t^\natural \in G$ we construct a rule (1) $\mathsf{f}_{s^\natural}(\mathsf{ts}(s^\natural)) \to \mathsf{f}_{t^\natural}(\mathsf{ts}(s^\natural))$, if $s^\natural \sqsubseteq t^\natural$; (2) $\mathsf{f}_{s^\natural}(\mathsf{ts}(t^\natural)) \to \mathsf{f}_{t^\natural}(\mathsf{ts}(t^\natural))$, if $t^\natural$ is a state refinement of $s^\natural$; (3) $\mathsf{f}_{s^\natural}(\mathsf{ts}(s^\natural)) \to \mathsf{f}_{t^\natural}(\mathsf{ts}(t^\natural))\,[\![\mathsf{tval}(C)]\!]$, if the edge is labelled by C; (4) $\mathsf{f}_{s^\natural}(\mathsf{ts}(s^\natural)) \to \mathsf{f}_{t^\natural}(\mathsf{ts}^*(t^\natural))$; $s^\natural$ corresponds to a field update on address p, $heap(q)$ is variable cn, and q may-reach p; and (5) $\mathsf{f}_{s^\natural}(\mathsf{ts}(s)) \to \mathsf{f}_t(\mathsf{ts}(t))$, otherwise. Here $\mathsf{ts}^*(t^\natural)$ is $\mathsf{ts}(t^\natural)$ but q is a fresh class variable to account for the side-effects. Figure 3 illustrates the cTRS obtained from the computation graph of `append`. The fresh-variable $l7$ in f_E, is due to a (non-observed) side-effect of the field update.

$$
\begin{aligned}
\mathsf{f}_A(\mathsf{L}(l3), l2, \mathsf{null}) &\to \mathsf{f}_B(\mathsf{L}(l3), l2, \mathsf{L}(l3))\\
\mathsf{f}_B(\mathsf{L}(l3), l2, \mathsf{L}(l3)) &\to \mathsf{f}_S(\mathsf{L}(l3), l2, \mathsf{L}(l3))\\
\mathsf{f}_S(\mathsf{L}(l3), l2, \mathsf{L}(l5)) &\to \mathsf{f}_C(l5, \mathsf{null}, \mathsf{L}(l3), l2, \mathsf{L}(l5))\\
\mathsf{f}_C(\mathsf{L}(l6), \mathsf{null}, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{L}(l6))) &\to \mathsf{f}_{C'}(\mathsf{L}(l6), \mathsf{null}, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{L}(l6)))\\
\mathsf{f}_C(\mathsf{null}, \mathsf{null}, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{null})) &\to \mathsf{f}_{C''}(\mathsf{null}, \mathsf{null}, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{null}))\\
\mathsf{f}_{C'}(\mathsf{L}(l6), \mathsf{null}, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{L}(l6))) &\to \mathsf{f}_D(\mathsf{L}(l3), l2, \mathsf{L}(l6))\\
\mathsf{f}_D(\mathsf{L}(l3), l2, \mathsf{L}(l6)) &\to \mathsf{f}_S(\mathsf{L}(l3), l2, \mathsf{L}(l6))\\
\mathsf{f}_{C''}(\mathsf{null}, \mathsf{null}, \mathsf{L}(l3), l2, \mathsf{L}(l5)) &\to \mathsf{f}_E(\mathsf{L}(l7), l2, \mathsf{L}(l2))
\end{aligned}
$$

Figure 3 The cTRS of `append`.

▶ **Theorem 5.** *Let $i, t \in \mathcal{JS}$. We have $\|\mathsf{ts}(\beta(i))\| \in O(|i|)$. Suppose $i \to_P^* t$ and G is obtained from P and $i^\natural$ such that $i \in \gamma(i^\natural)$. Then there exists $t^\natural \in \mathcal{AS}$ and a derivation $\mathsf{f}_{i^\natural}(\mathsf{ts}(\beta(i))) \to_{\mathcal{R}}^* \mathsf{f}_{t^\natural}(\mathsf{ts}(\beta(t)))$ such that $i \in \gamma(i^\natural)$ and $t \in \gamma(t^\natural)$. Moreover, $\mathsf{rcjvm} \in O(\mathsf{rctrs})$.*

References

1 M. Avanzini and G. Moser. Closing the Gap Between Runtime Complexity and Polytime Computability. In *Proc. 21th RTA*, LIPIcs, 2010.
2 M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl. Termination Graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, LNCS, 2010.
3 P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of 4th POPL*.
4 S. Falke and D. Kapur. A Term Rewriting Approach to the Automated Termination Analysis of Imperative Programs. In *Proc. 22nd CADE*, LNCS, 2009.
5 G. Klein and T-Nipkow. A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler. *ACM TOPLAS*, 28, 2006.
6 G. Moser. Proof Theory at Work: Complexity Analysis of Term Rewrite Systems. *CoRR*, abs/0907.5527, 2009. Habilitation Thesis.
7 Georg Moser and Michael Schaper. A Complexity Preserving Transformation from Jinja Bytecode to Rewrite Systems. *CoRR*, abs/1204.1568.
8 C. Otto, M. Brockschmidt, C. v. Essen, and J. Giesl. Automated Termination Analysis of Java Bytecode by Term Rewriting. In *Proc. 21th RTA*, LIPIcs, 2010.
9 S. E. Panitz and M. Schmidt-Schauß. Tea: Automatically proving termination of programs in a non-strict higher-order functional language. In *Proc. 4th SAS*, LNCS, 1997.
10 S. Rossignoli and F. Spoto. Detecting Non-cyclicity by Abstract Compilation into Boolean Functions. In *Proc. 7th VMCAI*, LNCS, 2006.