A Complexity Preserving Transformation from Jinja Bytecode to Rewrite Systems

Michael Schaper

Institute of Computer Science, University of Innsbruck, Austria,
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 a heap shape pre-analysis.

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 that incorporates the theory of Presburger arithmetic to express integer and Boolean operations naturally. TRSs (and its derivatives) have been successfully applied before for proving termination of computer programs: In [1] a transformation from C like programs with integer valued variables is proposed. This approach was extended in [2, 3] to prove termination of Java programs including user-defined data structures. A finite relation on abstract states is obtained by *symbolically evaluating* the bytecode instructions on abstract states, and suitably merging them. This relation is then transformed into rewrite rules, such that multiple rewrite steps mimic a program step. In [4] it has been shown that TRSs are a reasonable cost model for polytime computable functions and several methods have been developed in recent years to compute upper bounds of TRSs automatically [5–7]. This motivates to extend existing approaches to complexity analysis. Based on [2, 3] we propose an alternative representation of abstract states. We relate our approach to standard techniques from static program analysis, in particular *abstract interpretation* [8], and show that the transformation to cTRS is *complexity preserving*. This extended abstract is an excerpt from a report currently in progress [9].

Concrete Bytecode Domain. We analyse Jinja bytecode (JBC) programs. Jinja is a Java like language that exhibits its core features, but is formally specified and verified in Isabelle [10]. We expect the reader to be familiar with Java or a similar object-oriented language. A *Jinja value* is either a Boolean, an (unbounded) integer, the dummy value unit, the null reference null, or an address. Beside values JBC has *objects*, which are instances of user-defined data types. Figure 1 illustrates a bytecode program that appends a list to an existing list.

Bytecode is executed on the Jinja virtual machine (JVM). A *(JVM) state* is a pair consisting of the *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 heap and thus a state can be naturally represented as a graph, termed *state graph*, where labels (denoted $L(v)$) are stack (register) indices, non-address values or class identifiers and edges are empty or field identifiers. Furthermore a state graph has a root. The *size of a state* is defined on a *per-reference* basis, which unravels sharing. Let s be a state and S be its state graph. Let u, v be nodes and $u \rightharpoonup^*_S v$ be a simple path in S. The size of a stack (register) index u is $|u| := \sum_{u \rightharpoonup^*_S v} |L_S(v)|$, where $|l|$ is $\mathsf{abs}(l)$ if $l \in \mathbb{Z}$, otherwise 1. Then, the *size* of s is the sum of all sizes of stack and register indices in S, plus 1 for the root.

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

Fig. 1. List append.

Let P be a program. Let $\mathcal{JS}$ denote the set of states of 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* on $\mathcal{P}(\mathcal{JS})$ as the set extension of the one-step transition 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 $|i| \leqslant n\}$.*

Abstract Bytecode Domain. We introduce *abstract states* as generalisations of JVM states. Abstract states are similar to concrete states 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*. An abstract state represents a set of JVM states. Furthermore we employ an implicit representation of aliasing and 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 $\mathcal{AS} \ni \{\top, \bot\}$. Elements of $\mathcal{AS}$ are usually indicated with $\natural$.

Definition 2. *We define a preorder $\trianglelefteq$ on (abstract) non-address values, class identifiers and class variables. We have $v \trianglelefteq w$, if either (1) $v = w$; (2) $v =$ unit; (3) $v =$ null and w is class variable cn; (4) v is a Boolean (integer) and $w =$ bool (int); (5) $v = cn'$, w a class variable cn and v is a subclass of w.*

Let $S^\natural$ and $T^\natural$ be state graphs of states $s^\natural$ and $t^\natural$. We exploit $\trianglelefteq$ and the graph representation to define a partial order $\sqsubseteq$ on abstract states. The relation $s^\natural \sqsubseteq t^\natural$ holds, if there exists a morphism $m\colon V_{S^\natural} \to V_{T^\natural}$, such that (1) $root(S^\natural) =$

$root(T^\natural)$, (2) for all stack (register) indices $u \in S^\natural$, $L_{S^\natural}(u) = L_{T^\natural}(m(u))$, (3) for all other $u \in S^\natural$, $L_{S^\natural}(u) \trianglerighteq L_{T^\natural}(m(u))$, (4) for all $u \in S^\natural$: if $u \overset{i}{\rightharpoonup}_{S^\natural} v$, then $m(u) \overset{i}{\rightharpoonup}_{T^\natural} m(v)$, and (5) for all $u \xrightarrow{\ell} v \in S^\natural$ and $m(u) \xrightarrow{\ell'} m(v) \in T^\natural$, $\ell = \ell'$. Furthermore, for all $p \neq q \in s^\natural$, $m(p) \neq m(q) \in t^\natural$. Note that stack and register indices of $S^\natural$ and $T^\natural$ coincide for the same program location. For a suitable join operation $\mathcal{AS} := (\mathcal{AS}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ is a complete lattice.

Definition 3. *Let $s = (heap, frms) \in \mathcal{JS}$. We define $\beta\colon \mathcal{JS} \to \mathcal{AS}$. Suppose $\mathsf{dom}(heap) = \{p_1, \dots, p_n\}$. 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 [8, 11].*

In order to exploit the abstract domain, we propose *computation graphs* as finite representations of all relevant states in $\mathcal{AS}$, abstracting $\mathcal{JS}$. A computation graph is a finite control flow graph, in which nodes are abstract states, obtained by dynamically expanding nodes via *abstract computation* and suitably merging nodes representing equal program locations. 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 that represents the effect of the operation. Refinement steps are performed when no evaluation step can be performed. This is the case, if the instruction is either (1) a conditional jump and the top value of the stack is a Boolean variable; (2) a field access, field update, or a method invocation and the address is bound to a class variable; (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 instances of all subclasses. For (3), we consider states, where we set the addresses equal and unequal. Figure 2 illustrates the (incomplete) computation graph of append, obtained under the assumption that all variables are acyclic, do not alias and do not share at the beginning. We refine states by sharing and acyclicity facts [12, 13]. 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. For correctness, we require that the abstract semantics safely approximates the concrete semantics, ie., $\gamma(f^\natural(s^\natural)) \supseteq f^*(\gamma(s^\natural))$. We obtain following result:

Theorem 4. *Let $i, t \in \mathcal{JS}$. Suppose $i \to_P^* t$, where the runtime is m. Let G denote the computation graph of P obtained from some initial state $i^\natural$ such that $i \in \gamma(i^\natural)$. Then there exists an abstraction $t^\natural$ of t and m' such that $i^\natural \overset{m'}{\rightharpoonup}_G t^\natural$ holds, for $m \leqslant m' \leqslant K \cdot m$. Here constant $K \in \mathbb{N}$ only depends on G.*

Abstract Term Domain. We present the transformation from computation graphs to *constrained term rewrite systems* (*cTRS*). Our definition of cTRSs is a special case of the logical term rewrite systems introduced in [14]. We are only interested

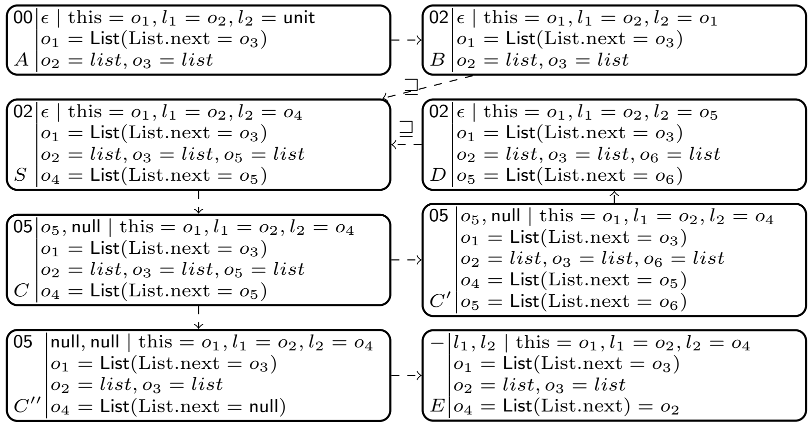

Fig. 2. The (incomplete) computation graph of append.

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. On the other hand, if there exists a substitution σ, such that $T \vdash C\sigma$, then C is *satisfiable*. Let C denote 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 $=_T$ denotes unification modulo T. 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 $\mathcal{R}$. The size of a term t, denoted as $\|t\|$ is defined as follows: (1) 1, if t is a variable; (2) $\mathsf{abs}(t)$, if t is an integer; (3) $1 + \sum_{i=1}^{n} \|t_i\|$ if $t = f(t_1, \ldots, t_n)$ and f is not an integer. The *derivation height* of a term t (denoted $\mathsf{dh}(t)$) with respect to $\mathcal{R}$ is defined as the maximal length of a derivation starting in t.

Definition 5. *We define the* runtime complexity *(wrt. $\mathcal{R}$) as follows:* $\mathsf{rctrs}(n) := \max\{\mathsf{dh}(t) \mid t \textit{ is basic and } \|t\| \leqslant n\}$*, 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 *finite* computation graph. For any state $s^\natural$ in G we introduce a new function symbol $\mathsf{f}_{s^\natural}$. Let $s^\natural, t^\natural$ be states in G: 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; (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. We write L (l) for a list symbol (variable). Note the fresh-variable $l7$ in f_E, due to (non-observed) side-effect of the field update.

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

Fig. 3. The cTRS of append.

Theorem 6. *Let $s, t \in \mathcal{JS}$. Then $\|\mathsf{ts}(\beta(s))\| \in O(|s|)$. Suppose $s \to_P^* t$, where s is reachable in P from some initial state i. Set $s' = \beta(s)$, $t' = \beta(t)$. Then there exists $s^\natural, t^\natural \in \mathcal{AS}$ and a derivation $\mathsf{f}_{s^\natural}(\mathsf{ts}(s')) \to_{\mathcal{R}}^+ \mathsf{f}_{t^\natural}(\mathsf{ts}(t'))$ such that $s \in \gamma(s^\natural)$ and $t \in \gamma(t^\natural)$. Furthermore for all n: $\mathsf{rcjvm} \in O(\mathsf{rctrs}(n))$.*

References

1. Falke, S., Kapur, D.: A Term Rewriting Approach to the Automated Termination Analysis of Imperative Programs. In: Proc. 22nd CADE. Volume 5663 of LNCS.
2. Otto, C., Brockschmidt, M., v. Essen, C., Giesl, J.: Automated Termination Analysis of Java Bytecode by Term Rewriting. In: Proc. 21th RTA. (2010) 259–276
3. Brockschmidt, M., Otto, C., von Essen, C., Giesl, J.: Termination Graphs for Java Bytecode. In: Verification, Induction, Termination Analysis. LNCS (2010)
4. Avanzini, M., Moser, G.: Closing the Gap Between Runtime Complexity and Polytime Computability. In: Proc. 21th RTA. LIPICS (2010)
5. Noschinski, L., Emmes, F., Giesl, J.: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems. In: Proc. 23rd CADE. LNCS (2011)
6. Hirokawa, N., Moser, G.: Automated Complexity Analysis Based on the Dependency Pair Method. In: Proc. 4th IJCAR. Volume 5195 of LNCS. (2008) 364–380
7. Avanzini, M., Moser, G.: A Combination Framework for Complexity. In: Proc. 24th RTA. LIPICS (2013)
8. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of 4th POPL. (1977) 238–252
9. Moser, G., Schaper, M.: A complexity preserving transformation from jinja bytecode to rewrite systems. CoRR **abs/1204.1568**
10. Klein, G., T-Nipkow: A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler. ACM Trans. Program. Lang. Syst. **28**(4) (2006) 619–695
11. Nielson, F., Nielson, H., Hankin, C.: Principles of Program Analysis. Springer Verlag (2005)
12. Secci, S., Spoto, F.: Pair-sharing analysis of object-oriented programs. In: Proc. 12th SAS. Volume 3672 of LNCS. (2005) 320–335
13. Rossignoli, S., Spoto, F.: Detecting Non-cyclicity by Abstract Compilation into Boolean Functions. In: Proc. 7th VMCAI. Volume 3855 of LNCS. (2006) 95–110
14. Kop, C., Nishida, N.: Term Rewriting with Logical Constraints. In: Proc. 9th FroCos. (2013) 343–358