Seminar Report

Deciding the Complexity of Programs Automatically
Termination Graphs Revisited

Michael Schaper
michael.schaper@uibk.ac.at

29 February 2012

Supervisor: Assoc. Prof. Dr. Georg Moser

Abstract

We revisit termination graphs from the viewpoint of runtime complexity. Suitably generalising the construction proposed in the literature, we define an alternative representation of Jinja bytecode (Jinja BC) executions as *computation graphs*. We show that the transformation from Jinja BC programs to computation graphs is *sound*, i.e., an infinite execution gives rise to an infinite path in the computation graph. Moreover, we establish that the transformation is *complexity preserving*.

Contents

1 Introduction

In this report we recall and survey a method for analyzing termination of bytecode programs. In particular, we are interested in Jinja, a programming language very similar to Java. There are various attempts for automated analysis of Java bytecode, for example [1, 8]. Further, in [14, 5, 4] termination of Java bytecode (Java BC) programs is studied. To this extent the execution of a Java BC program P is represented in a finite graph, a so-called *termination graph*. Based on this graph, integer term rewrite systems $\mathcal{R}$ (cf. [6]) are defined, such that termination of $\mathcal{R}$ yields termination of P. That is, the proposed transformation from Java BC to rewrite systems is *non-termination preserving*. A similar idea was already used before for the TEA termination tool to detect termination of functions written in a non-strict high-level functional programming language [15]. Although, they directly employed ordering constraints on the graph, rather than translating it into a term rewrite system.

We revisit termination graphs from the viewpoint of runtime complexity. Suitably generalising the earlier construction, we propose an alternative representation of Jinja BC executions in graph form as *computation graphs* G. The nodes of the computation graph are abstract states, representing sets of states of the Jinja Virtual Machine (Jinja VM). The edges represent symbolic evaluations together with refinements and abstraction steps.

We show that the transformation from Jinja BC programs to computation graphs is non-termination preserving, that is, any infinite evaluation of P gives rise to the existence of infinite paths in G. Moreover, we establish that the transformation is *complexity preserving*. For this, we measure the runtime complexity of P as a function that relates the maximal length of evaluations to the size of the initial state. (Note that this measure overestimates the size of the input to P only by a constant factor.) Moreover, the computation complexity maps the maximal length of a path in G to the size of the initial abstract state. We show that the runtime complexity of P is asymptotically bounded in the computation complexity of G.

The report is structured as follows: First, we give the preliminaries in Section 2. Second, we present an overview over Jinja in Section 3. Then, we will state similarities and differences between Jinja and Java in Section 4. In Section 5 and Section 6, we introduce abstract states and computation graphs. We conclude in Section 7.

2 Preliminaries

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:

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

We usually use square brackets to denote a list. Further, (::) denotes the cons operator, and (@) is used to denote the concatenation of two lists.

Definition 2.1. 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*, $Succ_{\cdot}\, V_G \to V_G^*$ 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 nodes in G, then *edges* in G are denoted as $u \to v$.

Definition 2.2. 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 $E_G \colon E_G \to \mathcal{L}$ 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 $E_G(u \to v)$ and $u, v \in V_G$. We often write $u \to v$ if the label is either not important or is clear from 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, $Succ_G$, and L_G, i.e., we write $G = (V, Succ, L)$ if no confusion can arise from this. Further, we also write $u \in G$ instead of $u \in V$.

Let $G = (V, Succ, L)$ 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 $u \neq v$. The *size* of G, i.e., the number of nodes, is denoted as $|G|$. The *depth* of G, i.e., the length of the longest path in G, is denoted as $\mathsf{dp}(G)$. We write $G \restriction u$ for the subgraph of G reachable from u.

3 Jinja

In this section we are going to give an overview over the Jinja programming language and the Jinja virtual machine, as presented in [10] by Klein and Nipkow. Jinja is a Java-like language and an acronym for "Jinja is not Java". It is formally specified and machine-checked in the theorem prover Isabelle/HOL and designed to exhibit the core features of Java. The whole work [10] includes the description of the Jinja language, the Jinja bytecode (Jinja BC), and the Jinja virtual machine (Jinja VM). Further, it offers a compiler from Jinja source code to Jinja bytecode, an elaborated type system and a bytecode verifier. Since our analysis will be based on the bytecode and its semantics induced by the virtual machine, we will restrict the description to Jinja BC and Jinja VM.

Actually, the Jinja programming language is not really designed and used for real-world programs. But there are several reasons why to pick Jinja as our target language: (i) The Jinja VM and the operational semantics of the Jinja BC instruction set is formally specified. This allows to reason about it properly. (ii) Though, Jinja does not support all constructs and features from Java, it has a concise and expressive set of bytecode instructions. In the sequel section we will compare Jinja with Java and illustrate that Jinja inherits the basic elements of Java. We will show that the instruction set and the information given in the bytecode programs are similar. Thus, Jinja is a reasonable alternative to Java for us.

In the following, we will give a summary of Jinja BC and Jinja VM: A *Jinja bytecode program* 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*. 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 *methodbody*. The method body is a triple (mxs, mxl, ins), where mxs denotes the maximal size of the operand stack, mxl the number of registers (not including the `this` pointer and the parameters) and ins a sequence of Jinja BC instructions.

A *Jinja value* is either:

- a Boolean,
- an integer,
- a reference (an address),
- the null reference null, or
- the dummy value unit.

We consider Jinja BC programs to be well-formed [10]. Further, we presuppose normal evaluation, that is, no exceptions are raised and demand that all data structures are non-cyclic. The handling of exceptions are formally defined in [10] and no problems would arise taking them into account. Though, it makes the definition of a Jinja VM state unnecessarily more complicated, without gaining much. On the other hand, we can not deal with cyclic-data without further examination.

The *Jinja virtual machine state* s is a tuple $s = (heap, frms)$, where *heap* is a global memory, mapping references to objects, and $frms$ is a list of frames.

A single frame frm provides an execution environment for a single method call and is a quintuple $frm = (stk, loc, cn, mn, pc)$. The operand stack stk is used to store partial results during the evaluation of an expression. The registers loc consists of the `this` reference, the parameters and the local variables of the invoked method. The operand stack and the registers store Jinja values. Further, cn represents the class where the current method mn is defined. Finally, pc represents the program counter.

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). Furthermore, to ease the presentation, we usually write os_i (loc_i) to denote an index $i \in \mathsf{dom}(stk)$ ($i \in \mathsf{dom}(loc)$). Moreover, in the following $\preceq$ denotes the subclass relation.

The bytecode instructions of Jinja are depicted in Figure 1.

`Load` n	load from register
`Store` n	store in register
`Push` v	push constant
`New` cn	create class instance
`Getfield` fn cn	fetch field form instance
`Putfield` fn cn	set field in instance
`Checkcast` cn	check if instance is of class cn
`Invoke` mn n	invoke instance method with n parameters
`Return`	return from method
`Pop`	pop top of the stack
`IAdd`	integer addition
`Goto` n	relative jump
`CmpEq`	equality comparison
`IfFalse` n	branch

Figure 1: The bytecode instruction set of Jinja.

Definition 3.1. A *heap* is a mapping from *addresses* to *objects*, where an object is a pair (cn, fdt):

- cn denotes the *class name*, and
- fdt denotes the fieldtable, i.e., a mapping from (cn', fn) to Jinja values, where fn is a *field name* and cn' is a (not necessarily proper) superclass of cn, denoted $cn \preceq cn'$.

Definition 3.2. We now illustrate the one-step relation of Jinja BC instructions. We first consider the easy cases:

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

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

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

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

$$\frac{(heap, (i_2 :: i_1 :: stk, loc, cn, mn, pc) :: frms)}{(heap, ((i_1 + i_2) :: stk, loc, cn, mn, pc + 1) :: frms)}\ \texttt{IAdd}$$

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

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

$$\frac{(heap, (v_2 :: v_1 :: stk, loc, cn, mn, pc) :: frms)}{(heap, ((v_1 = v_2) :: stk, loc, cn, mn, pc + 1) :: frms)}\ \texttt{CmpEq}$$

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

For the rest of the relations, a little more description is necessary: Let a be a fresh address and obj be a new class instance of cn', with its fields set to the default value. Then,

$$\frac{(heap, (stk, loc, cn, mn, pc) :: frms)}{(heap\{a \mapsto obj\}, (a :: stk, loc, cn, mn, pc + 1) :: frms)\,.}\ \texttt{New}\ cn'$$

Let a be an address and $heap(a) = (cn'', fdt)$. Either the requested field fn is defined in cn'' or inherited from a superclass. We have $cn'' \preceq cn'$. Then

$$\frac{(heap, (a :: stk, loc, cn, mn, pc) :: frms)}{(heap, (fdt(cn', fn) :: stk, loc, cn, mn, pc + 1) :: frms)\,.}\ \texttt{Getfield}\ fn\ cn'$$

Let v be a value and a be an address such that $heap(a) = (cn'', fdt)$. We set $fdt' := fdt\{(cn', fn) \mapsto v\}$ to denote the updating of field fn in fdt. Then,

$$\frac{(heap, (v :: a :: stk, loc, cn, mn, pc) :: frms)}{(heap\{a \mapsto (cn'', fdt')\}, (stk, loc, cn, mn, pc + 1) :: frms)\,.}\ \texttt{Putfield}\ fn\ cn'$$

Since, we only consider exception-free evaluation, all casts in the source program are valid. Therefore,

$$\frac{(heap, (v :: stk, loc, cn, mn, pc) :: frms)}{(heap, (v :: stk, loc, cn, mn, pc + 1) :: frms)\,.}\ \texttt{Checkcast}\ cn$$

Let $method(cn, mn) = (cn', ts, t, mbody)$ denote that method mn with type signature $mn\colon ts \to t$ is defined in the (not necessarily proper) superclass cn' of cn and its body is $mbody$. By definition $mbody = (mxs, mxl, ins)$, where mxs denotes the maximal size of the operation stack, mxl denotes the maximal number of registers and ins the instruction list. See [10] for a suitable implementation of $method$.

Suppose a denotes the address of the calling object and $heap(a) = (cn', fdt)$. Moreover, $method(cn', mn') = (cn'', ts, t, (mxs, mxl, ins))$. Then,

$$\frac{(heap, (p_{n-1} :: \cdots :: p_0 :: a :: stk, loc, cn, mn, pc) :: frms)}{(heap, frm' :: (stk, loc, cn, mn, pc) :: frms)\ ,}\ \texttt{Invoke}\ mn'\ n$$

where $loc' := [a, p_0, \ldots, p_{n-1}]$ @ $units$ and $frm' = ([], loc', cn'', mn', 0)$. Here $units$ denotes an array of unit-values of size mxl.

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

Let v be the return value, $frm = (stk', loc', cn', mn', pc')$ and $frm' = (v :: stk', loc', cn', mn', pc' + 1)$. Then

$$\frac{(heap, (v :: stk, loc, cn, mn, pc) :: frm :: frms)}{(heap, frm' :: frms)\,.}\ \texttt{Return}$$

Example 3.3. Consider Figure 2, which illustrates a program that concatenates two lists. For a better understanding, we also depict the source code in a Java like syntax. Here, register 0 refers to *this*, and register 1 and 2 refer to variables *ys* and *cur*, respectively.

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

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

```
0: Load 0
1: Store 2
2: Push unit
3: Pop
4: Load 2
5: GetField next List
6: Push null
7: CmpNeq
8: IfFalse 7
9: Load 2
10: GetField next List
11: Store 2
12: Push unit
13: Pop
13: Goto -10
15: Push unit
16: Pop
17: Load 2
18: Load 1
19: PutField next List
20: Push unit
```

Figure 2: The append function.

4 Observations on Jinja and Java

In the following we state similarities and differences between Jinja and Java. Jinja is designed to exhibit the core-features of Java. Obviously these languages have many things in common. But, at first glance it is not clear to what extent the specification of Jinja coincide with Java's informal description. The main goal is to show that Jinja is an adequate alternative for Java, i.e., the simplifications induced by Jinja are of conceptual interest rather than technical. We will show that all informations of a Jinja BC program and the instructions of the Jinja VM exist in a some form in Java. On the other hand, we will comment on the additional features which are offered by Java.

We refer to the official Java language [7] and Java virtual machine [11] specification. The specification of the Java virtual machine (Java VM) is kept very general. This is due to the reason that the specification is designed to support different implementations, while maintaining compatibility between them. Thus, allowing to run code on different systems that for example are optimized for speed or resource usage. This makes it harder to compare the two languages and their corresponding virtual machines. Nevertheless, the description of the memory model of the Java VM, the instruction set of Java bytecode (Java BC) and the class file format of a Java BC program give a good starting point.

First, we summarize the *runtime data area* of the Java VM. The Java VM supports the execution of multiple threads. Therefore, each thread has its own *program counter*, and has access to a private *Java virtual machine stack*. The Java VM stack stores *frames*. A frame has an array of *local variables* and an *operand stack*. Additionally, each frame has a reference to the runtime constant pool. The *heap* and the *method-area* are shared among all threads. The heap stores *class instances* and *arrays* and deallocation of memory is performed by a *garbage collector*. Among other things, the method-area contains the *runtime constant pool*, field and method data, and the code array for methods and constructors. The runtime constant pool is allocated for every class and serves as symbol table, containing constants known at compile time and method and field references that must be resolved at runtime. The Java VM specification assumes no particular type for the implementation for its components. For example the size of the Java VM stack can be of fixed size or dynamically expanded. But it enforces safety conditions, e.g., an error have to be thrown if no more memory can be allocated during runtime.

Recall the definition of the Jinja BC program and the Jinja VM state: The Jinja BC program itself corresponds to the method-area, containing information of classes, fields and methods. Frame and heap are equivalent and fulfill the same purpose. The runtime constant pool just offers a way for resolving informations efficiently. If we ignore threads, the memory model is almost equal.

Next, we have a look on the Java BC instruction set. The instructions of Java are not polymorphic, i.e., many instructions come in similar forms for every primitive type, for references, and for arrays. For example, there exist instructions `iload` n, `bload` n loading an integer value or an boolean value from register n, respectively. Further, instructions may be ambiguous. So `iload_0` is the same instruction as `iload 0`, but the operator is implicit in the first case.

In the second case an extra byte is needed for the operand. That is why, there are about 33 different `load` bytecode instructions and over 200 in total.

Figure 3 depicts an overview over the Java bytecode instruction set, as given in [11]. It is easy to see, that all instructions of Jinja have a counterpart in Java.

		0	no operation
1	-	17	push constant onto operand stack
18	-	20	load constant from runtime constant pool
21	-	53	load value from local variable
54	-	86	store value into local variable
87	-	95	stack operations
96	-	131	arithmetic and logical operations
		132	local integer increment
133	-	147	type conversion
148	-	152	comparison operation
153	-	166	conditional branch
		167	unconditional branch
168	-	169	handle subroutine (finally)
170	-	171	compound conditional branch (switch)
172	-	177	return instructions
178	-	179	getstatic and putstatic
180	-	181	getfield and putfield
182	-	186	method invocation
187	-	189	object creation
		190	arraylength
		191	throwing exception
		192	checkcast
		193	instanceof
194	-	195	synchronization
		196	widening
		197	creation of multinewarray
198	-	199	conditional branch (null)
200	-	201	wide branch
		202	breakpoint
254	-	255	reserved opcodes

Figure 3: Overview over the Java bytecode instruction set.

Next, we will give an overview over the class file format. Thus we will know which informations are available at runtime and let us compare it to the information given in a Jinja BC program. The class file is designed to minimize the size of the file. Numeric, string and other constants are stored in the constant pool. Classes are then described by nested structures, referencing elements in the constant pool. Figure 4 depicts some components of the Java class file format in a simplified manner. Note that details are omitted. But it let us describe how a method invocation is performed and how a method is represented in the class file. Structures with the prefix `CONSTANT` are elements of the constant pool of the class. Fields with suffix `index` refer to an index in the constant pool. A *descriptor* is a string representation of a type. A method descriptor represents the type or the parameters and the return type of the

```
ClassFile {
  cp_info constant_pool[];
  short this_class_index;
  short super_class_index;
  field_info fields[];
  method_info methods[];
}

CONSTANT_Methodref_info {
  short class_index;
  short name_and_type_index;
}

CONSTANT_NameAndType_info {
  u2 name_index;
  u2 descriptor_index;
}
```

```
CONSTANT_Utf8_info {
  byte bytes[];
}

CONSTANT_Class_info {
  short name_index;
}

method_info {
  short name_index;
  short descriptor_index;
  attribute_info attributes[];
}

Code_attribute {
  short max_stack;
  short max_locals;
  byte code[];
}
```

Figure 4: The class file format.

method. The `invokestatic` instruction gets an index for the runtime constant pool as argument. The entry at this index is a `CONSTANT_Method_ref` structure. The `CONSTANT_Method_ref` structure represents a symbolic reference to a method and contains enough information to resolve a method during runtime. The actual code is in the `method_info` structure. Recall the description of a Jinja BC program. It is easy to see that the `method_info` structure contains the same information for methods. (Actually, it contains even more information). Namely, an identifier, the types of the parameter and the return type, the maximal size of the local variables, the maximal size of the operand stack, and the instruction list. The same goes for fields.

Now we will focus on some differences between Jinja and Java. We argue informally how they are realized in Java or how they could be realized in Jinja. Note that this list is not complete, but rather gives a good overview.

Support for additional primitive types: Java supports integral types of different length: `byte`, `short`, `int`, `long` and `char`. Furthermore, floating-point types `float` and `double`, and a `Boolean` type is available. On the bytecode level `byte`, `short`, `char`, and `Boolean` are encoded as `int`. Therefore, *conversion* instructions are necessary.

Support for arrays: Java supports arrays by providing an additional reference type, $arrayref$. Like class instances arrays are also objects, but created and manipulated with a distinct set of instructions. For example, the instructions `newarray 10`, `iaload` and `iastore` are used to create an array whose components are of type integer, to access an integer value from an array and to store an integer value into an array, respectively.

Support for threads: As already stated the Java VM supports the execution of multiple threads. Concurrent access to an object can be prevented by allowing the thread to gain ownership of the monitor, which is associated to each object.

Note that, there exists an extension of Jinja, named "Jinja with Threads" [12], which also supports threads.

Support for visibility annotations: On the bytecode level visibility annotations affects the *resolution* of fields and methods. Here, resolution means to resolve the concrete class, field or method from a reference. If the lookup succeeds but the field is not accessible, due to the annotation, then an exception is thrown.

Support for Strings: Although, there exists no dedicated type for strings, strings are directly supported by the Java VM. Due to optimization, strings are handled specially, i.e., the Java VM tries to map equal string literals to the same entry in the constant pool. Further, every type can be conversed to a string.

Support for static fields/methods: Static fields and static methods can be accessed even if there exist no instance of the corresponding class in the heap. Therefore, the code array of the method can be obtained directly from the runtime constant pool. Considering fields, then somewhere in the runtime data area have to reside the value of the field. Though, the Java VM specifies not where exactly.

Support for method overloading: Java defines the method signature by the method identifier and the amount and the type of its parameters. Overloading allows to define multiple methods with an equal identifier, as long as the signature is unique. This is not directly supported in the Jinja VM, but could easily be managed using a preprocessor renaming the methods in the source program.

We conclude that the difference between Jinja and Java is actually not so big and therefore Jinja is an adequate language for our intentions.

5 Abstract States

Until now, we have recalled the Jinja programming language and compared it to the Java programming language. Now, we introduce abstract states. Abstract states represent sets of states in the Jinja VM and will later represent nodes in our computation graph.

We extend Jinja by abstract variables `Class` for each class considered. Further, $\texttt{Bool} := \{\mathsf{true}, \mathsf{false}\}$ denotes an *abstract Boolean value* and any interval $I \subseteq [-\infty, \infty]$ denotes an *abstract integer value*. We write `Int` instead of I, if the concrete interval is not relevant and we identify the interval $[z, z]$, where $z \in \mathbb{Z}$ with the integer z. An *abstract value* is either a Jinja value or an abstract Boolean or integer value. Furthermore we make use of an infinite supply of *abstract locations* $\zeta_0, \zeta_1, \zeta_2, \ldots$ The intuition being that abstract locations will be used to address non-address values in the heap.

Definition 5.1. Recall Definition 3.1, defining the *heap* of a Jinja VM state. We lift the definition to abstract states, i.e., a *heap* is a mapping from *addresses* to *objects*, where an object is either a pair (cn, fdt) or an abstract variable. The projection functions cl and ft are now defined as follows:

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

Registers and operand stack of a frame, now store abstract values. Furthermore, we define *annotations* of addresses in a state s, denoted as iu. Formally, the annotations are pairs $p \neq q$ of addresses, where $p, q \in heap$ and $p \neq q$. The intuition of iu is to express that for $p \neq q \in iu$, we disallow sharing of these addresses in concrete states represented by the state s.

Definition 5.2. An *abstract state* is a triple consisting of a *heap*, a list of *frames*, and a set of *annotations*. An abstract state which does not contain abstract variables and where addresses cannot be shared further is a *concrete* (or *Jinja*) state.

In the presence of abstract variables an abstract state represents a set of Jinja a states. In the following we do not notationally distinguish between Jinja states and abstract states, as this will always be clear from context.

Let s be a state and let *heap* denote the heap of s. In preparation of Definition 5.3 below we define a bijection ϕ that essentially associates every non-address value in *heap* with an abstract location. To define ϕ we define its graph as function of *heap*:

$$\begin{aligned}\Phi(heap) :=\{&(\zeta, val) \mid \exists \text{ address } a\colon \mathsf{rg}(\mathsf{ft}(heap(a))) \ni val,\\ &val \text{ not an address, } \zeta \text{ fresh}\}\,.\end{aligned}$$

Finally, we set $\phi(\zeta) := val$ if $(\zeta, val) \in \Phi(heap)$.

Definition 5.3. Let *heap* denote the heap. We represent *heap* as a directed graph with edge labels $H = (V_H, Succ_H, L_H, E_H)$, where the nodes, the successor relations and the labeling functions are defined as follows:

$$V_H := \mathsf{dom}(heap) \cup \mathsf{dom}(\phi)$$

$$Succ_H(u) := \begin{cases} [f^*((cn_1, fn_1)), \ldots, f^*((cn_k, fn_k))] & \text{if } u \text{ is an address, } \mathsf{ft}(heap(u)) = f, \mathsf{dom}(f) = \{(cn_1, fn_1), \ldots, (cn_k, fn_k)\} \\ [] & \text{otherwise} \end{cases}$$

$$L_H(u) := \begin{cases} \mathsf{cl}(heap(u)) & \text{if } u \text{ is an address} \\ \phi(u) & \text{if } u \text{ is an abstract location} \end{cases}$$

$$E_H(u \to v) := \begin{cases} (cn, fn) & \text{if } u \text{ is an address, } \mathsf{ft}(heap(u)) = f \text{ and } f((cn, fn)) = v \\ \epsilon & \text{otherwise} \end{cases}$$

Here $f^*((cn, fn)) := f((cn, fn))$, if $f((cn, fn))$ is an address and $f^*((cn, fn)) := \phi^{-1}(f((cn, fn)))$ otherwise.

We usually confuse the heap with its representation as graph. In particular we call a value *val* reachable from an address *a* in *heap*, if there exists a path from *a* to *val* in the heap graph of *heap*.

For a state $s = (heap, frms, iu)$ with top-frame $frm = (stk, loc, cn, mn, pc)$, we are only concerned with the part of the heap that is reachable from frm. Suppose $os_1, \ldots, os_m$ and $l_1, \ldots, l_n$ denote the operand stack and the registers in frm. Then $heap \restriction frm$ denotes the restriction of *heap* to all nodes reachable from $\{stk(i) \mid i \in \{1, \ldots, m\} \cup \{loc(i) \mid i \in \{1, \ldots, n\}\}$.

In preparation of the next definition, we extend the assignment of abstract locations ϕ defined above. The resulting function will again be denoted as ϕ. No confusion can arise from this. Let the graph Φ of ϕ be defined as a function of *stk*, *loc*, and *heap* as follows: $\Phi(stk, loc, heap) := \{(\zeta, val) \mid$ *val* is a non-address value in *stk*, *loc*, or *heap* and ζ is fresh$\}$. Finally, we set $\phi(\zeta) := val$ if $(\zeta, val) \in \Phi(stk, loc, heap)$.

Definition 5.4. Let $s = (heap, frms, iu)$ and $frm = (stk, loc, cn, mn, pc)$ be the top-frame. Let $\mathsf{dom}(stk) = \{1, \ldots, m\}$ and let $\mathsf{dom}(loc) = \{1, \ldots, n\}$. Moreover suppose H denotes $heap \restriction frm$. We define the *state graph of s* as 5-triple $S = (V_S, Succ_S, L_S, E_S, iu)$, where the first four components denote a directed graph with edge labels and iu denotes a set of annotations. The nodes, the successor relation, and the labeling function of the directed graph are defined as follows:

$$V_S := \{1, \ldots, m + n\} \cup V_H \cup \mathsf{dom}(\phi)$$

$$Succ_S(u) := \begin{cases} [stk^*(u)] & \text{if } u \in \{1, \ldots, m\}, \text{ i.e., } u \text{ is a stack index} \\ [loc^*(u - m)] & \text{if } u \in \{m + 1, \ldots, n\}, \text{ i.e., } u - m \text{ is a register index} \\ Succ_H(u) & \text{if } u \in V_H\,. \end{cases}$$

$$L_S(u) := \begin{cases} os_u & \text{if } u \in \{1, \ldots, m\}, \text{ i.e., } u \text{ is a stack index} \\ l_{u-m} & \text{if } u \in \{m+1, \ldots, n\}, \text{ i.e., } u - m \text{ is a register index} \\ \phi(u) & \text{if } u \text{ is an abstract location} \\ L_H(u) & \text{if } u \in V_H \\ \phi(u) & \text{otherwise}. \end{cases}$$

$$E_S(u \to v) := \begin{cases} E_H(u \to v) & \text{if } u, v \in H \\ \epsilon & \text{otherwise}. \end{cases}$$

Here $stk^*(u)$ and $loc^*(u)$ is defined like f^* as introduced in Definition 5.3, i.e., $stk^*(u) := stk(u)$ ($loc^*(u) := loc(u)$), if $stk(u)$ ($loc(u)$) is an address and $stk^*(u) := \phi^{-1}(stk(u))$ ($loc^*(u) := \phi^{-1}(loc(u))$) otherwise.

We often confuse a state s and its representation as a state graph and addresses v with its corresponding object $heap(v)$.

Definition 5.5. We define a binary relation $\sqsubseteq$ on abstract values. Let v, w be values. Then $v \sqsubseteq w$ if

1. $v \in \{\mathsf{null}, \mathsf{unit}\}$ and either $v = w$ or $w \in \{\mathsf{null}, \mathtt{Class}, \mathtt{Bool}, \mathtt{Int}\}$, or
2. $v, w \subseteq [-\infty, \infty]$ and $v \subseteq w$,
3. $v, w \in \{\mathsf{true}, \mathsf{false}, \mathtt{Bool}\}$ and either $v = w$ or $w = \mathtt{Bool}$,
4. v, w are class names or abstract class variables and $v \preceq w$. Further, if v is a class variable then so is w.

Based on the definition of $\sqsubseteq$ we introduce the following variant of graph morphism, called *state morphism*.

Definition 5.6. Let S and T be state graphs. A *state morphism* from T to S (denoted $m\colon T \to S$) is a function $m\colon V_T \to V_S$ such that

1. for all $u \in T$, if $L_T(u) = os_u$ or l_{u-m}, then $u = m(u)$ and $L_T(u) = L_S(m(u))$. Otherwise, $L_T(u) \sqsupseteq L_S(m(u))$,
2. for all $u \in T$, $m^*(Succ_T(u)) = Succ_S(m(u))$, and
3. for all $u \xrightarrow{\ell} v \in T$ and $m(u) \xrightarrow{\ell'} m(v) \in S$, $\ell = \ell'$.

If no confusion can arise we refer to a state morphism 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. We are in the position to define when a state s' is an *instance* of s. In our construction the sharing information is implicitly given by possible morphisms that respects the instance relation of values. Thus, we generally overapproximate sharing.

Definition 5.7. Let $s = (heap, frms, iu)$ and $t = (heap', frms', iu')$ be states. Then t is an *instance* of s (denoted as $t \sqsubseteq s$) if the following conditions hold:

1. all corresponding program counters in the frame lists $frms$ and $frms'$ coincide,
2. there exists a morphism $m\colon s \to t$, and
3. for all $p \neq q \in iu$ we have that $m(p) \neq m(q)$ and $iu' \subseteq m^*(iu)$.

If t an instance of s, then we call s an *abstraction* of t, denoted as $s \sqsupseteq t$.

It is an easy consequence of the composability of morphism that the instance relation $\sqsubseteq$ is transitive.

Definition 5.8. Let $s = (heap, frms, iu)$ be a state and let p, q denote distinct addresses in $heap$ such that $p \neq q \notin iu$. Then we say p and q are *unifiable* (denoted as $p \stackrel{?}{=} q$) if there is a state t and a morphism $m\colon s \to t$, such that $m(p) = m(q)$.

Let s and t be states. If there exists a third state p such that $s \sqsubseteq p$ and $t \sqsubseteq p$, then p is said to be an *abstraction*.

Definition 5.9. Let s be a state and let $S = (V_S, Succ_S, L_S, E_S, iu)$ be its state graph. The *size* of s, denoted $|s|$, is defined as follows: $\sum_{l \in L_S} |l|$, where $|l|$ is $\mathsf{abs}(l)$ if $l \in \mathbb{Z}$, otherwise 1. (As usual $\mathsf{abs}(l)$ denotes the absolute value of the integer l.)

Definition 5.10. Let $s = (heap, frms, iu)$ be a state and $S = (V, Succ, L, E, iu)$ be its state graph. In the following we define an alternative representation of the graph.

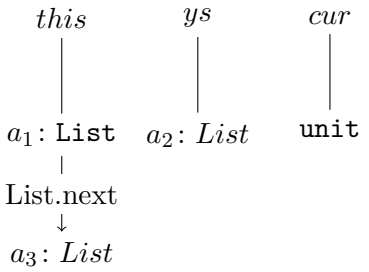
$$\mathsf{n}(u) := \begin{cases} \phi(u) & \text{if } u \text{ is an abstract location} \\ u & \text{otherwise}\,. \end{cases}$$

$$\mathsf{node}(u) := \begin{cases} u = L(u)(e_1 = \mathsf{n}(v_1), \ldots, e_n = \mathsf{n}(v_n)) & \text{if } u \text{ is an address,} \\ & Succ(u) = [v_1, \ldots, v_n] \text{ and} \\ & E(u \to v_i) = e_i \\ u = \mathsf{n}(v) & \text{otherwise, where } Succ(u) = v\,. \end{cases}$$

Let A be the set of abstract locations. The *node specification* of S, denoted $\mathsf{ns}(S)$, is then defined as follows:

$$\mathsf{ns}(S) := \bigcup_{u \in L \setminus A} \{\mathsf{node}(u)\}\,.$$

Example 5.11. Consider the `append` example depicted in Figure 2. Figure 5 illustrates the state graph of the initial state. Here, addresses a_2 and a_3 are abstract class variables. Using the node specification and the additional information of program counter and stack, we get a compact representation of the state.



00	$this = a_1, ys = a_2, cur = \texttt{unit} \mid \epsilon$
	$a_1 = \texttt{List(List.next} = a_3\texttt{)}$
A	$a_2 = List, a_3 = List$

Figure 5: State graph and node representation of the initial state of append.

6 Computation Graphs

Let P denote a well-formed Jinja BC program that allows a normal execution without exceptions and employs only acyclic data-structures. P is fixed for the remainder of this paper.

We write $instr\_of(cn, mn)$ to denote the *instruction list* of method mn in class cn of the considered program. See [10] for a suitable implementation of the function $instr\_of$.

Definition 3.2 presents the single-step execution of each Jinja BC command. Based on these instructions, and actually mimicking them quite closely, we define how abstract states are symbolically evaluated. The generalisation of Definition 3.2 is in most cases standard. Let $s = (heap, frms, iu)$ be an abstract state with top-frame $frm = (stk, loc, cn, mn, pc)$. Suppose $instr = instr\_of(cn, mn)(pc)$, i.e., the current instruction. By case distinction on $instr$ one defines the *symbolic evaluation* of P. In Definition 6.1 we have worked out the cases for `IAdd`, `Putfield`, and `Invoke`. The other cases are left to the reader.

Definition 6.1. First, we consider the instruction `IAdd`. For that we extend addition to abstract values as usual: $\texttt{Int} + i = \texttt{Int}$, where $i \in \mathbb{Z} \cup \{\texttt{Int}\}$. We define the corresponding symbolic single-step execution as follows:

$$\frac{(heap, (i_2 :: i_1 :: stk, loc, cn, mn, pc) :: frms, iu)}{(heap, ((i_1 + i_2) :: stk, loc, cn, mn, pc + 1) :: frms, iu)}.$$

Now, we consider a `Putfield` $fn\ cn'$ instruction. Let v be a value and a be an address such that $heap(a) = (cn'', fdt)$. We set $fdt' := fdt\{(cn', fn) \mapsto v\}$ to denote the updating of field fn in fdt. Suppose there exists no address $p \in heap$ such that $a \stackrel{?}{=} p$. Then we define the following step:

$$\frac{(heap, (v :: a :: stk, loc, cn, mn, pc) :: frms, iu)}{(heap\{a \mapsto (cn'', fdt')\}, (stk, loc, cn, mn, pc + 1) :: frms, iu)}.$$

Finally, we consider an instruction `Invoke` $mn'\ n$. Suppose a denotes the address of the calling object and $\mathsf{cl}(heap(a)) = cn'$. Moreover, $method(cn', mn') = (cn'', ts, t, (mxs, mxl, ins)$. We set:

$$\frac{(heap, (p_{n-1} :: \cdots :: p_0 :: a :: stk, loc, cn, mn, pc) :: frms, iu)}{(heap, frm' :: (stk, loc, cn, mn, pc) :: frms, iu)},$$

where $loc' := [a, p_0, \ldots, p_{n-1}]$ @ $units$ and $frm' = ([], loc', cn'', mn', 0)$. Here $units$ denotes an array of unit-values of size mxl.

In addition to symbolic evaluations, we define refinement steps on abstract states s if the information given in s is not concrete enough to execute a given instruction. Following [5] we make use of *case distinction*, *class instance*, and *unsharing*.

Definition 6.2. Let $s = (heap, frms, iu)$ be state, let a be an address and $heap(a) \in \{\texttt{Int}, \texttt{Bool}\}$. Suppose X denotes a refinement of $heap(a)$. Then we obtain the following refinement step:

$$\frac{(heap, frms, iu)}{(heap\{a \mapsto X\}, frms, iu)}.$$

Definition 6.3. Let $s = (heap, frms, iu)$ be state, let a be an address and $heap(a) = \texttt{Class}$. Suppose $subclasses := \{cn \mid cn \preceq \texttt{Class}\}$ and let $cn \in subclasses$. Furthermore, suppose $(cn_1, fn_1), \ldots, (cn_n, fn_n)$ denote fields of cn (together with the defining classes). We obtain the following two refinement steps, where the second takes care of the case, where `Class` is replaced by the null-pointer:

$$\frac{(heap, frms, iu)}{(heap\{a \mapsto (cn, fdt')\}, frms, iu)} \qquad \frac{(heap, frms, iu)}{(heap\{a \mapsto \mathsf{null}\}, frms, iu)}.$$

Here $fdt'((cn_i, fn_i)) := v_i$ such that $v_i \in \{\texttt{Int}, \texttt{Bool}, \texttt{Class}\}$ is defined in correspondence to the definition of C_i.

Definition 6.4. Let $s = (heap, frms, iu)$ be state, let $S = (V, Succ, L, E, iu)$ denote its state graph, and let p and q denote address in $heap$ such that $p \stackrel{?}{=} q$, i.e., p and q potentially represents the same object. We obtain the following refinement steps: The first case forces these addresses to be distinct. The second case substitutes all occurrences of q with p.

$$\frac{(heap, frms, iu)}{(heap, frms, iu \cup \{p \neq q\})} \qquad \frac{(heap, frms, iu)}{(heap', frms', iu)},$$

where $heap'$ ($frms$') is equal to $heap$ ($frms$) with all occurrences of q replaced by p.

Let s and s' be abstract states such that s' is obtained from s due to Definition 6.1—6.4. Then we say s' is obtained from s by an *abstract computation*.

Definition 6.5. A *computation graph* $G = (V_G, E_G)$ is a directed graph, where V_G are abstract states and $s \to t \in E_G$ if either

- t is obtained from s by a symbolic evaluation,
- t is obtained by a refinement step, or
- $s \sqsubseteq t$ holds.

Let G be a computation graph. We write $G\colon s \rightharpoonup_G t$ to indicate that state t is directly reachable in G from s. If s is reachable from t in G we write $G\colon s \stackrel{*}{\rightharpoonup}_G t$.

In the remainder of this section, we will state and prove soundness of computation graphs. Note that our definition of the computation graph is non-constructive. While it is easy to see that we can always enforce a *finite* computation graph, if we make use of suitable abstract states, we are not interested here in providing an actual construction. There may be different strategies,

when and how abstraction is employed to summarise states present in the graph. For example, one could summarise states whenever there are two states with the same program counter. On the other hand the resulting abstract states could be more exactly, if a while construction is evaluated more often before summarised. However, in proving soundness we can assume the existence of a finite computation graph for P.

Let s and t be concrete states. Then we denote by $P\colon s \xrightarrow{\mathsf{jvm}}_1 t$ the one-step transition relation of the Jinja VM. If there exists a (normal) evaluation of s to t, we write $P\colon s \xrightarrow{\mathsf{jvm}} t$. The next lemma states that any single-step execution on the Jinja VM can be simulated by a least one step in the computation graph.

Lemma 6.6. *Let s be a state and let s' be a concrete state such that $s' \sqsubseteq s$. Then $P\colon s' \xrightarrow{\mathsf{jvm}}_1 t'$ implies the existence of a state t such that $t' \sqsubseteq t$ and $G\colon s \xrightarrow{+}_G t$.*

Proof. In proof it suffices to observe that there are only two reasons why a symbolic evaluation is not directly applicable to s. Either (i) s is missing concrete information on values or (ii) an execution of a `Putfield` operation is prohibited as the considered reference r is not unique. For case (i) Definition 6.2 or 6.3 are used to concretize s. With respect to case (ii), we employ Definition 6.4 to make the reference r unique. □

Example 6.7. Consider the `append` example depicted in Figure 2. Figure 6 illustrates the computation graph of `append`. For the sake of readability we omit the *val* field of the list. Note that this graph is not complete and its exact construction goes beyond the scope of this report. But, every possible run of `append` has a path in this graph. We argue informally:

First, consider the initial node A. It is easy to see that A is an abstraction of all concrete initial states.

Next, consider node B. The *this* reference can not be null. Otherwise, the method could not be invoked. Therefore *cur* is never null. After pushing the reference of *cur.next* and null onto the operand stack, we reach node C. At $\mathsf{pc} = 7$ we want to compare the reference of *cur.next* with null. But, *cur.next* is not concrete. Therefore, an instance refinement is performed, yielding nodes C' and C''. When refining *cur.next* we can not unify a_5 and a_1 anymore. Therefore, we introduce the unsharing information $a_5 \neq a_1$. The branch where a_5 could be equal to a_1 is of no relevance and is omitted here.

First, we consider that *cur.next* is not null, but references an arbitrary instance. This is illustrated in node C'. The step from C' to D is trivial. The state graphs of D and B are depicted in Figure 7. Let *id* denote the identity function and $m = id(V_B)$. Then $m\{a_4 \mapsto a_5, a_5 \mapsto a_6\}$ is a state morphism from B to D. Therefore, D is an instance of B.

Second, we consider the case when *cur.next* is null, which is depicted in node C''. Node E is obtained from C'' after performing the `Putfield` instruction. We add $a_5 \neq a_2$. Otherwise the list could be cyclic.

We define the *runtime* of a Jinja VM for a given normal evaluation $P\colon s \xrightarrow{\mathsf{jvm}} t$ as the number of single-step execution in the course of the evaluation from s to

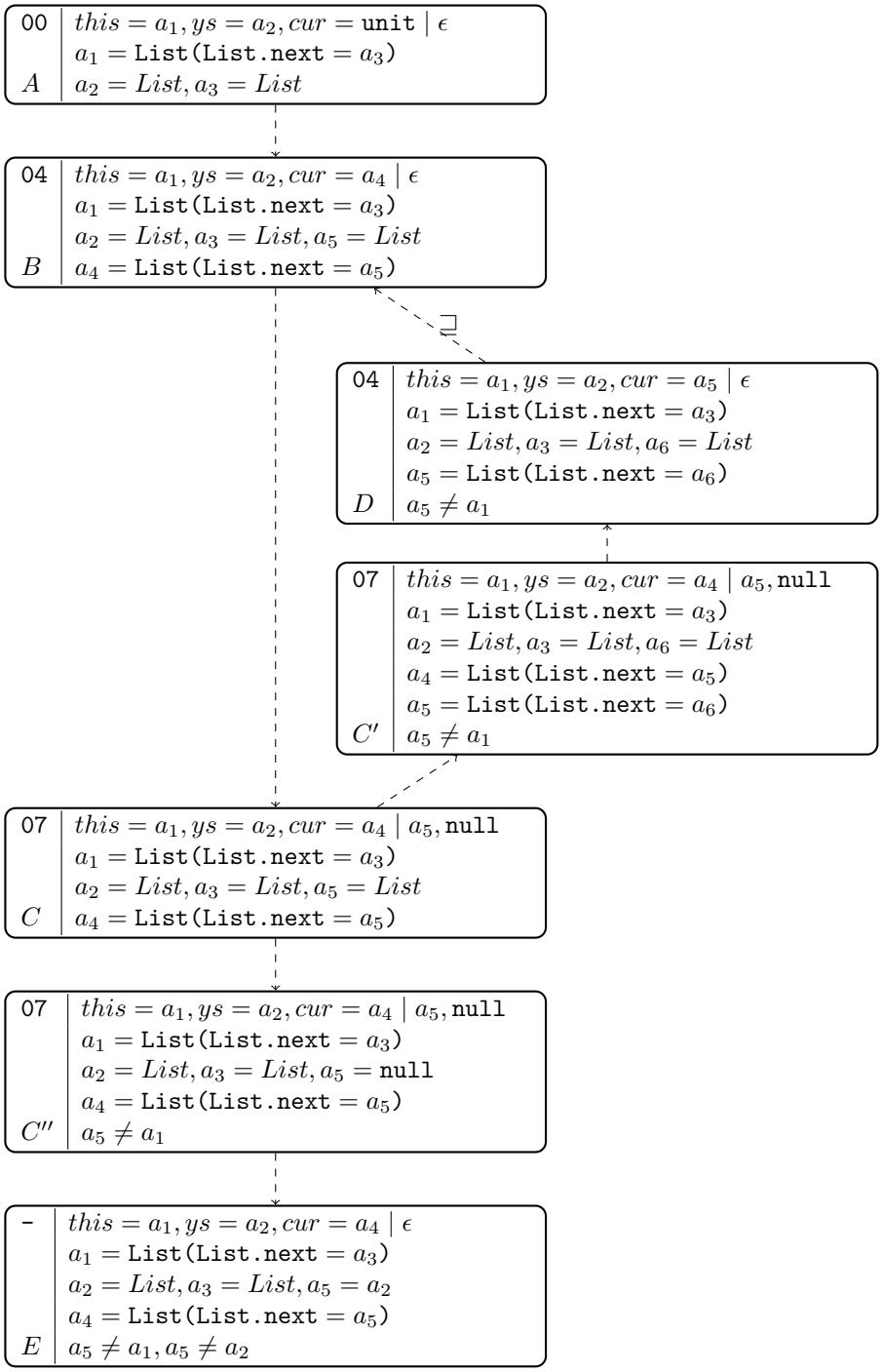


Figure 6: The (incomplete) computation graph of List.append.

t. On the other hand the *computation length* denotes the maximal length of a path in the computation graph G such that $G\colon s \twoheadrightarrow^*_G t$. Let s be a state and let S be the state graph of s. Recall Definition 5.9, defining the size of some state s.

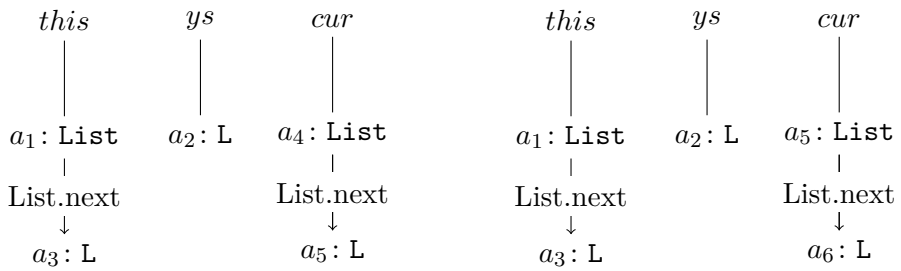


Figure 7: State graph of B and D.

Definition 6.8. We define the *runtime complexity* with respect to P as follows:

$$\mathsf{rcjvm}(n) := \max\{m \mid P\colon s \xrightarrow{\mathsf{jvm}} t \text{ holds such that the runtime is } m \text{ and } |s| \leqslant n\}\,.$$

Similarly the *computation complexity* with respect to G is defined as $\mathsf{cc}(n) := \max\{m \mid G\colon s \xrightarrow{*}_G t \text{ holds such that the computation length is } m \text{ and } |s| \leqslant n\}$.

Theorem 6.9. *Let s' and t' be concrete states. Suppose $P\colon s' \xrightarrow{\mathsf{jvm}} t'$. Then there exists an abstraction s of s' and a computation $G\colon s \xrightarrow{*}_G t$ such that $t' \sqsubseteq t$. Furthermore $\mathsf{rcjvm} \in O(\mathsf{cc})$.*

Proof. Suppose $|s'| \leqslant n$, where $n \in \mathbb{N}$ is arbitrary. Let m denote the runtime of the execution $P\colon s' \xrightarrow{\mathsf{jvm}} t'$. By induction on m in conjunction with Lemma 6.6 we conclude the existence of states s and t such that $G\colon s \xrightarrow{*}_G t$. Moreover, suppose the computation length of $G\colon s \xrightarrow{*}_G t$ is m'. Then $m \leqslant m'$. In particular $m \leqslant m' \leqslant \mathsf{cc}\, n$ as $|s| \leqslant |s'|$ by definition. Thus, we conclude $\mathsf{rcjvm} \in O(\mathsf{cc})$. □

7 Conclusion

In this report we have shown that Jinja is a reasonable alternative to Java. Further, we propose computation graphs as suitable representation of the execution of a Jinja VM. We have shown that this representation is complexity preserving. In future work it needs to be clarified whether our result on complexity preservation still holds, if the *cycles* of the computation graph are considered separately [14]. Furthermore, the notion of (innermost) runtime complexity for integer rewrite systems need to be clarified (cf. [9, 3] for the standard definition of runtime complexity of a rewrite system). Finally, methods for runtime complexity need to be adapted to integer rewrite systems (cf. [2, 13, 9] for examples of existing techniques).

References

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *J. Autom. Reason.*, 46:161–203, February 2011.

[2] M. Avanzini and G. Moser. Dependency Pairs and Polynomial Path Orders. In *Proc. of 20th RTA*, volume 5595 of *LNCS*, pages 48–62. Springer Verlag, 2009.

[3] M. Avanzini and G. Moser. Closing the gap between runtime complexity and polytime computability. In *Proc. of 21th RTA*, Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010.

[4] M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive java bytecode programs by term rewriting. In *Proc. of 22nd RTA*, LIPIcs, pages 155–170, 2011.

[5] M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl. Termination Graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, volume 6463 of *LNCS*, pages 17–37, 2010.

[6] C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving Termination of Integer Term Rewriting. In *Proc. of 20th RTA*, volume 5595 of *LNCS*, pages 32–47, 2009.

[7] J. Gosling, B. Joy, G. Steel, and G. Bracha. *The Java^{TM} Language Specification*. Addison Wesley, third edition, 2005.

[8] P. M. Hill, E. Payet, F. Spoto, Hill@comp. Leeds. Ac. Uk Iremia, and Université De La Réunion.

[9] N. Hirokawa and G. Moser. Automated complexity analysis based on the dependency pair method. *CoRR*, abs/1102.3129, 2011. submitted.

[10] G. Klein and T. Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[11] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Prentice Hall, second edition, 1999.

[12] A. Lochbihler. Jinja with threads. In *The Archive of Formal Proofs*. `http://afp.sf.net/entries/JinjaThreads.shtml`, December 2007. Formal proof development.

[13] L. Noschinski, F. Emmes, and J. Giesl. A dependency pair framework for innermost complexity analysis of term rewrite systems. In *Proc. of 23rd CADE*, volume 6803 of *LNCS*, pages 422–438, 2011.

[14] C. Otto, M. Brockschmidt, C. v. Essen, and J. Giesl. Automated termination analysis of Java bytecode by term rewriting. In *Proc. of 21th RTA*, pages 259–276, 2010.

[15] S. E. Panitz and M. Schmidt-Schauß. Tea: Automatically proving termination of programs in a non-strict higher-order functional language. In *Proc. 4th Int. Static Analysis Symp*, pages 345–360. Springer-Verlag, 1997.