Internal Report 2010–6 June 2010

Universiteit Leiden
Opleiding Informatica

A Pushdown System Representation for Unbounded Object Creation

Jurriaan Rot

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

A Pushdown System Representation for Unbounded Object Creation

Jurriaan Rot[1], Frank de Boer[1,2], Marcello Bonsangue[1,2]

[1] LIACS – Leiden University
[2] Centrum voor Wiskunde en Informatica (CWI)
jrot@liacs.nl,frb@cwi.nl,marcello@liacs.nl

Abstract. We introduce a block-structured programming language which supports object creation, global variables, static scope and recursive procedures with local variables. Because of the combination of recursion, local variables and object creation, the number of objects stored during a computation is potentially unbounded. However, we show that a program can be viewed as a type of pushdown automata, for which the halting problem as well as LTL and CTL model checking are decidable.

Key words: object creation, model checking, pushdown systems, pushdown automata

1 Introduction

From the 1960s onwards imperative programming has evolved with the introduction of high-level programming constructs for mastering the complexity of software by abstraction, encapsulation and modularity. The initial description of computation in terms of assignment statements to change a program state, sequential and conditional composition, and conditional looping [10] has been extended with procedures in combination with block structures which enable the construction and declaration of complex state changes abstracting from the concrete implementation [19]. Pointers are a very flexible programming mechanism, allowing manipulation of dynamically growing and potentially unbounded data structures. In the eighties mainstream imperative programming languages added the support of objects [23], a collection of procedures acting on an encapsulated state, having an identity that can be referred to by other objects. Other powerful programming techniques like inheritance and polymorphism enable code reuse, and are crucial for programming-in-the-large [22].

The increasing flexibility in programming comes, however, with an increasing complexity in reasoning about programs. Model checking is a technique for exhaustively checking a (model of a) program for possible errors [4]. Traditionally, in order to guarantee termination of a model checking procedure, finite-state models are required, and thus only programs over finite data domains are considered. But to program with dynamical data structures, objects may need to be created, removed and modified when moving from a state to another in a

computation. Thus, by their very nature, objects are unbounded: for example, during a recursive computation new objects can be created infinitely often. In order to achieve finite-state models for object-oriented programs, different types of abstraction and restrictions of programs have been considered (see related work below). Typically one disallows object creation and considers only a finite number of objects already existing before the computation starts, or allows for object creation only within restricted forms of recursion. However, the necessity to restrict programs before their analysis limits the applicability of model checking techniques to modern imperative programming languages.

In this paper we introduce a simple block-structured programming language which supports object creation, global variables, static scope and recursive procedures with local variables. In order to focus on the main issues, we restrict to a single but unbounded data structure, namely that of object identities. Other *finite* data domains could have been added without problem, but would have increased the complexity of the model without strengthening our main result. Although very simple, the language is powerful enough to encode the control flow of high-level imperative programming languages including closed class-based object-oriented programs, like Java. Because of the combination of recursion with local variables, a program may have infinitely many different states. Since we allow to store object references into local variables, the number of objects stored during a computation is potentially unbounded.

For our language we define two semantics: a concrete one that is infinite state, and a symbolic one that is also infinite state but is based on an enhanced version of the model of recursive procedures with local variables via a suitable pushdown system [12]. A pushdown system is a simple type of pushdown automaton used to generate behavior rather than to accept languages [5]. It provides a finite representation generating infinite state systems, where a state consists of a control part and a stack. In our model of a pushdown system global variables and the current local variables form the control states, whereas the current executing statement is on top of the stack. Actually it is more common to model only the global variables in the control state, while the local variables and the control point are (part of) the top of the stack (see e.g. [21]). We chose our approach for convenience in the proofs, but it can easily be modified to the more common approach. In our model, when a procedure is called, a copy of the current local variables is stored on the stack to recover the original values after the procedure returns, and the local variables in the control state are initialized again. In order to achieve finitely many control states, we abstract from the concrete identities of the objects, but maintain their symmetries, i.e. the equality relation among object identities [13]. Our main result is that the concrete and the symbolic semantics are strongly bisimilar.

Reachability for an infinite state system is generally undecidable. However, for a pushdown system it turns out that both the halting problem and reachability are decidable [16]. In fact, it is possible to model check pushdown systems against linear-time or branching-time temporal formulas. For linear-time

temporal formulas the complexity is even of the same order as for finite state systems [5].

This paper is organized as follows. In Section 2, we introduce the syntax of our language and give an informal description of its semantics. Section 3 provides a concrete execution model using a transition system on infinite states, and in Section 4 we describe the construction for the symbolic semantics based on pushdown systems. The relationship between these two models is studied in Section 5. Finally, the last section discusses some relevant consequences of our result, and possible future steps.

Related work. Currently there are several model checkers for object oriented languages. Java Path Finder [14] is basically a Java Virtual Machine that executes a Java program not just once but in all possible ways, using backtracking and restoring the state during the state-space exploration. Even if Java Path Finder is capable of checking every Java program, the number of states stored during the exploration is a limit on what can be effectively checked. As with JCAT [9], Java source code can be translated into Promela, the input language of SPIN [15]. Since Promela does not support dynamic data structures, they have to allocate fixed-size heaps and stacks.

Bandera [8] is an integrated collection of tools for model-checking concurrent Java software using state-of-the art abstraction, partial order reductions and slicing techniques to reduce the state space. It compiles Java source code into a reduced program model expressed in the input language of other existing verification tools. For example, it can be combined with the SAL (Symbolic Analysis Laboratory) model checker [18] that uses unbounded arrays whose sizes vary dynamically to store objects. In order to explore all reachable states model checking is restricted to Java programs with a bounded (but not fixed a priori) number of objects.

Model checking of a possibly unbounded number of objects but for a language with a restricted form of recursion (tail recursion) and no block structure has been studied using high level allocation Büchi automata [11], a generalization of history dependent automata [17] that enables for a finite state symbolic semantics very similar to ours. Full recursion, but with a fixed-size number of objects is instead considered in jMoped [12], using a pushdown structure to generate an infinite state system.

The current state of the art of model checking approaches for languages with object creation and full recursion in terms of concrete memory addresses, require an a priori bound on the size of the heap for reachability analysis (e.g. [6]). In order to overcome this problem, the main contribution of this paper is the precise abstraction of the heap in terms of equivalence classes of program variables which refer to the same memory address.

2 A simple imperative language with object creation

This section introduces a simple programming language that supports object creation, global and local variables, and recursive procedures. To simplify the

presentation it is restricted to a single data structure, that of object identities. A program consists of a finite set of procedures, each acting on some global and local state. Procedures can store identities in global or local variables, compare them, and call other procedures.

We assume a finite set of *program variables* V ranged over by $x, y, \ldots$ such that $V = G \cup L$, where G is a set of *global variables* $\{g_1, g_2, \ldots, g_n\}$ and L is a set of *local variables* $\{l_1, l_2, \ldots, l_m\}$, with G and L disjoint. For P a finite set of *procedure names* $\{p_0, \ldots, p_k\}$, a program is a set of *procedure declarations* of the form $p_i :: B_i$, where B_i, denoting the *body* of the procedure p_i, is a statement defined by the following grammar

$$B \ ::= \ x := y \mid x := \text{new} \mid B; B \mid [x = y]B \mid [x \neq y]B \mid B + B \mid p\,.$$

Here x and y are program (local or global) variables in V, and p is a procedure name in P. The procedure $p_0 \in P$ is called the *initial* procedure of a program.

The language is statically scoped. The *assignment* statement $x := y$ assigns the identity stored in y (if any) to x. If x was already referring to an object identity, this gets lost. In particular, if x is the only variable of the program referring to an object o, then after an assignment $x := y$, the object o cannot be referenced anymore and gets lost forever. The statement $x :=$ new *creates* a new object that will be referred to by the program variable x. As for the ordinary assignment, the old value of x is lost. In a program execution, a program variable x is said to be *defined* if there was an assignment or object creation statement earlier in the execution with the variable x at left–hand side. *Sequential composition* $B_1; B_2$, *conditional statements* $[x = y]B$ and $[x \neq y]B$ and *nondeterministic choice* $B_1 + B_2$ have the standard interpretation. A *procedure call* p means that the body B associated with p is executed next on the same global state but on a new fresh local state. After the procedure body terminates, its local state is destroyed forever and the previous local state (from which the procedure has been called) is restored. Changes to the global state, however, remain.

More general boolean expressions in conditional statements can be obtained by using sequential composition and nondeterministic choice. In fact $(b_1 \wedge b_2)B$ can be written as $(b_1)b_2B$, whereas $(b_1 \vee b_2)B$ as $(b_1B) + (b_2B)$. Negation of a boolean expression b can be obtained by transforming b into an equivalent boolean expression in conjunctive disjunctive normal form, for which negation of the simple expression $[x = y]$ and $[x \neq y]$ is defined as expected.

Ordinary while, skip, and if-then-else statements can be expressed easily in the language, using recursive procedures, conditional statements and nondeterministic choice. For the sake of simplicity, we allow creation and assignment of a single object identity only; generalizations to simultaneous assignments and object creation can be added in a straightforward manner. We assume automatic garbage collection of object identities that are not referenced anymore by any global variables or instances of local variables in a program execution.

The language does not directly support parameter passing. However, it is worthwhile to note that we can model procedures with call-by-value parameters by means of global variables. Let $p(v_1, \ldots, v_n)$ be a procedure with formal

parameters $v_1, \ldots, v_n$. We see the formal parameters as local variables and introduce for each parameter v_i a corresponding global variable g_i (which does not appear in the given program). Every procedure call $p(x_1, \ldots, x_n)$ can be modeled by the statement $g_1 := x_1; \ldots; g_n := x_n; p$ whereas the body B of $p(v_1, \ldots, v_n)$ can be modeled by $v_1 := g_1; \ldots; v_n := g_n; B$. A similar approach can be taken to model procedures with return values. Finally, method calls $x.m(x_1, \ldots, x_n)$ then can be modeled by introducing the called object x as an additional 'parameter' of the procedure m.

3 Transition System Semantics

In this section, we introduce a semantics of the programming language which is defined in terms of an explicit representation of objects by natural numbers. This representation allows a simple implementation of object creation. A *program state* of a program is a function

$$s : V \longrightarrow \mathbb{N}_\perp,$$

where $\mathbb{N}_\perp = \mathbb{N} \cup \{\perp\}$ ($\perp$ is used to denote "undefined"). To model object creation we distinguish a global "system" variable c which is used as a counter, and is not used by programs. We implicitly assume that $s(c) \neq \perp$, for every state s.

A *configuration* of a program is a pair $\langle s, S\rangle$ where s is a program state and S is a stack of statements and local states. An *execution step* of a program is a transition from a configuration C to a configuration C', denoted by $C \longrightarrow C'$. The possible execution steps are given below. For modeling state updates we use multiple assignments of the form $s[x_1, \ldots, x_n := v_1, \ldots, v_n]$, where x_i and x_j are distinct, for $i \neq j$. The head of a stack is separated from the tail with the right-associative operator $\bullet$; for example, $S' = e \bullet S$ is the stack consisting of head e and tail S.

$$\langle s, B_1; B_2 \bullet S\rangle \longrightarrow \langle s, B_1 \bullet B_2 \bullet S\rangle \tag{1}$$

$$\frac{s(y) \neq \perp}{\langle s, x := y \bullet S\rangle \longrightarrow \langle s[x := s(y)], S\rangle} \tag{2}$$

$$\langle s, x := \text{new} \bullet S\rangle \longrightarrow \langle s[x, c := c, c+1], S\rangle \tag{3}$$

$$\frac{s(x) = s(y) \quad s(x) \neq \perp}{\langle s, [x = y]B \bullet S\rangle \longrightarrow \langle s, B \bullet S\rangle} \tag{4}$$

$$\frac{s(x) \neq s(y) \quad s(x) \neq \perp \quad s(y) \neq \perp}{\langle s, [x \neq y]B \bullet S\rangle \longrightarrow \langle s, B \bullet S\rangle} \tag{5}$$

$$\langle s, B_1 + B_2 \bullet S\rangle \longrightarrow \langle s, B_i \bullet S\rangle \quad (i \in \{1, 2\}) \tag{6}$$

$$\langle s, p_i \bullet S\rangle \longrightarrow \langle s', B_i \bullet s \bullet S\rangle \tag{7}$$

where $s'(l) = \perp$, for every local variable l and $s'(g) = s(g)$, for every global variable g.

$$\langle s, s' \bullet S\rangle \longrightarrow \langle s[\bar{l} := s'(\bar{l})], S\rangle \tag{8}$$

where $\bar{l}$ denotes the sequence of local variables $l_1, \ldots, l_m$ and $s'(\bar{l})$ denotes the sequence of values $s'(l_1), \ldots, s'(l_m)$.

For technical convenience only, a procedure call pushes onto the stack as local environment the entire state. Further, we assume a distinguished global variable 'nil' such that $s(\text{nil}) = \perp$, for every state s. The following corollary states some basic properties of the semantic rule 8.

Corollary 1. *If $\langle s, s' \bullet S\rangle \longrightarrow \langle s'', S\rangle$ then for every $x, y \in V$:*

1. *x and y are both global implies $s''(x) = s''(y)$ iff $s(x) = s(y)$,*
2. *x and y are both local implies $s''(x) = s''(y)$ iff $s'(x) = s'(y)$,*
3. *x is global and y is local implies $s''(x) = s''(y)$ iff $s(x) = s'(y)$.*

Proof. It suffices to observe that by definition of the rule 8 we have $s''(g) = s(g)$ and $s''(l) = s'(l)$, for every global g and every local l. □

Further, we have the following invariance property about the flow of information between the current state and the stacked states.

Lemma 1. *For every computation $\langle s_0, p_0\rangle \longrightarrow^* \langle s, S\rangle$, variable z, local variable l and local state s' appearing in S, we have $s(z) = s'(l)$ iff there exists a global variable g such that $s(z) = s'(g)$ and $s'(l) = s'(g)$.*

Proof. The proof is by induction on the length of the computation. The basis of the induction is trivial because the stack of the initial configuration only contains the statement p_0.

It suffices to show that every production respects the property. First note that for any rule except 7, no new states are added to the stack so we only need show that the resulting state still satisfies the equivalence. In rule 1, 4, 5 and 6 we have $s = s'$ so the equivalence holds by the induction hypothesis. In the assignment rule 2, the resulting state is $s[x := s(y)]$. Now if $z \equiv x$ ($\equiv$ denotes *syntactic* identity) then $s[x := s(y)](z) = s(y)$ and if $z \not\equiv x$ then $s[x := s(y)](z) = s(z)$. In both cases, the result follows from the induction hypothesis. In rule 3, the resulting state is $s[x, c := c, c + 1]$. It follows that $s[x, c := c, c + 1](y) = s(y)$ and $s[x, c := c, c + 1](x) \neq s(y)$, for all $y \not\equiv x$. The result then follows from the induction hypothesis. Rule 7, the procedure call, adds the state s to the top of the stack. Since the values of the locals are $\perp$ in the resulting state, and the values of globals in the resulting state are equal to their value in s, by the induction hypothesis the equivalence holds for every state in the resulting stack including s. Lastly for rule 8 does not alter the globals. For the locals, the result follows from the induction hypothesis for the popped abstract state. Note that there exists a computation $\langle s_0, p_0\rangle \longrightarrow \langle s', S\rangle$, where s' denotes the popped abstract state. □

4 Pushdown System Semantics

The above semantics gives rise to an infinite state system because of unbounded recursion, and because of the representation of objects by natural numbers used to model unbounded object creation. Since we can only test objects for equality we can reduce this state-space by the introduction of equivalence classes of variables, that is, two variables belong to the same equivalence class if they denote the same object. However local variables can generate again an unbounded number of equivalence classes. We show in this section how we can restrict to an *apriori finite* number of equivalence classes of variables by the introduction of so-called "freeze" variables, which will be used to compare the partitions of variables before and after executing a procedure call. This will allows for a reallocation of the global variables with respect to the local variables of the caller. To do this, we associate with each global variable g a fresh and unique local variable g' (which we assume does not appear in the given program).

An *abstract* program state now consists of a partition of global and local variables (including the freeze variables). To facilitate easy treatment of such a partition, we represent it as a function

$$\sigma : V \longrightarrow |V| + 1$$

where $|V|$ is the cardinality of the set of variables V, and $|V| + 1$ is identified with the set $\{0, \ldots, |V|\}$. Thus two (distinct) variables x and y belong to the same equivalence class iff $\sigma(x) = \sigma(y)$. We use zero for the equivalence class of variables which are undefined, e.g., $\sigma(\mathrm{nil}) = 0$, for every abstract state σ.

A *configuration* of a program now is a pair $\langle \sigma, \Sigma \rangle$ where σ is an abstract state as defined above and Σ is a stack of statements and abstract states. Because of the way we model partitions of the set of variables V, rules 1, 2, 4, 5 and 6 directly apply in this model and are therefore not repeated here. The rule for object creation is modified as follows.

$$\langle \sigma, x := \mathrm{new} \bullet \Sigma \rangle \longrightarrow \langle \sigma', \Sigma \rangle \tag{9}$$

where $\sigma' = \sigma$, if all indices except zero are used in σ, else $\sigma' = \sigma[x := i]$, where $i \neq 0$ is the smallest index not already used by σ.

This new rule for object creation describes it in terms of an update of the current partition of the variables V by isolating the variable x. This is achieved by assigning to the variable x an index different from zero not in use. Note that in case such an index does not exist the partition represented by σ consists of singleton sets only and therefore is not affected by object creation, i.e., we do *not* need to assign a new index to x because it is already isolated.

The rule for procedure calls is modified as follows.

$$\langle \sigma, p_i \bullet \Sigma \rangle \longrightarrow \langle \sigma', B_i \bullet \sigma \bullet \Sigma \rangle \tag{10}$$

where $\sigma' = \sigma[\overline{l} := \overline{0}][\overline{g'} := \sigma(\overline{g})]$, $\overline{g'}$ denotes the sequence $g'_1, \ldots, g'_n$ of freeze variables and $\sigma(\overline{g})$ denotes the sequence of indices $\sigma(g_1), \ldots, \sigma(g_n)$. Note that $\sigma[\overline{l} := \overline{0}](l) = 0$, for every local variable $l \in L$.

A procedure call now additionally initializes the freeze variables by the values of their corresponding global variables and stores the old abstract state onto the stack. Note that execution of B_i does not affect the freeze variables.

Finally, the rule for returns from a procedure call is modified as follows.

$$\langle \sigma, \sigma' \bullet \Sigma \rangle \longrightarrow \langle \sigma_n, \Sigma \rangle \tag{11}$$

where $\sigma_0 = \sigma'$ and for $0 < i \leq n$ (where n is the number of globals) we define σ_i by the following cascade of if-then-else statements:

- if $\sigma(g_i) = 0$ then $\sigma_i = \sigma_{i-1}[g_i := 0]$ else
- if $\sigma(g_i) = \sigma(g_j)$, for some $j < i$, then $\sigma_i = \sigma_{i-1}[g_i := \sigma_{i-1}(g_j)]$ else
- if $\sigma(g_i) = \sigma(g')$, for some freeze variable g', then $\sigma_i = \sigma_{i-1}[g_i := \sigma'(g)]$ else
- if in σ_{i-1} all indices except 0 are used then $\sigma_i = \sigma_{i-1}$ else
- $\sigma_i = \sigma_{i-1}[g_i := k']$, where $k' \neq 0$ is the smallest index not already used by σ_{i-1}.

Upon return, which consitutes the 'heart of the matter', we need to update the stored partition σ' by reallocating the global variables according to the new partition described by σ. We do so by means of the freeze variables which represent in σ the partitioning of the global variables in σ' and as such form a reference point for comparison with the local variables in σ'. In other words, a partition in σ' containing global variables is represented in σ by the corresponding freeze variables. Therefore, in case in σ a global variable g_i is identified with a freeze variable g' we have to identify it with all the local variables which belong to the partition of g in σ'. This is simply obtained by setting the index of g_i to $\sigma'(g)$. Note that in fact $\sigma'(g) = \sigma(g')$. However, $\sigma'(g) = \sigma'(g')$ does *not* hold in general because the freeze variable g' represents the *initial* value of its global variable g which may have been affected by the computation which led to σ'. Further, we observe that the choice of a particular freeze variable does not affect the reallocation because if two distinct freeze variables are identified in σ, then so are their corresponding global variables in σ'. Finally, we note that global variables which are "drifted away" from these freeze variables can only denote objects which are different from those denoted by the local variables in σ'. Therefore for these variables new partitions have to be created. In order to obtain suitable indices for these global variables we have defined the overall update of σ' incrementally by processing the global variables one by one. For each global variable g_i its reallocation is defined by σ_i as follows: if g_i is undefined in σ then so it is in σ_i, else if g_i is identified by σ with some already processed g_j $(j < i)$ then we set its index to that of g_j in σ_{i-1}, else if g_i is identified by σ with some freeze variable then we set its index to that of the corresponding global variable in σ'. In case none of the above holds then we have to create a new partition for g_i as in the new rule for object creation.

Example 1. We give an example of a derivation which illustrates the procedure call and return. The state is represented as a partition. We assume p is a procedure name with body $p :: B = g_2 := \text{new}$. Furthermore g_1, g_2 are global variables, l_1, l_2 are local variabes.

$$
\begin{array}{ll}
 & \langle\{\{g_1, l_1\}, \{g_2, l_2\}\}, p \bullet \Sigma\rangle \longrightarrow \\
\text{(call)} & \langle\{\{g_1, g_1'\}, \{g_2, g_2'\}, \{l_1, l_2\}\}, g_2 := \text{new} \bullet \{\{g_1, l_1\}, \{g_2, l_2\}\} \bullet \Sigma\rangle \longrightarrow \\
\text{(creation)} & \langle\{\{g_1, g_1'\}, \{g_2'\}, \{g_2\}, \{l_1, l_2\}\}, \{\{g_1, l_1\}, \{g_2, l_2\}\} \bullet \Sigma\rangle \longrightarrow \\
\text{(return)} & \langle\{\{g_1, l_1\}, \{g_2\}, \{l_2\}\}, \Sigma\rangle
\end{array}
$$

The first transition step pushes the current state unto the stack. The new state separates all the local variables l_1 and l_2 (this set is indexed by zero which indicates "undefinedness") and introduces the freeze variables. The execution of $g_2 := \text{new}$ in the next transition step isolates the variable g_2. Finally, upon returning, g_1 is still identified with l_1 but both l_2 and g_2 are now isolated. It is important to note that in the above computation we can also replace l_1 and l_2 by freeze variables of earlier procedure calls.

Each set of variables identified by an abstract state defines an object. Further, as explained above, two sets of variables V_i and V_{i+1} identified by the respective abstract states σ_i and σ_{i+1}, which are stored consecutively (from bottom to top) on a given stack Σ, define the same object if and only if there exists a global variable $g \in V_i$ for which its freeze variable g' is in V_{i+1}. The equivalence relation induced by this relation between the sets of variables stored on a given a stack Σ represents the objects generated by Σ. Figure 1 depicts a chain of sets of variables which denote the same object.

The following corollary states some basic properties of the semantic rule 8.

Corollary 2. *If* $\langle \sigma, \sigma' \bullet \Sigma\rangle \longrightarrow \langle \sigma'', \Sigma\rangle$ *then for every* $x, y \in V$*:*

1. *x and y are both global implies $\sigma''(x) = \sigma''(y)$ iff $\sigma(x) = \sigma(y)$*
2. *x and y are both local implies $\sigma''(x) = \sigma''(y)$ iff $\sigma'(x) = \sigma'(y)$*
3. *x is global and y is local implies $\sigma''(x) = \sigma''(y)$ iff there exists a global variable g such that $\sigma'(y) = \sigma'(g)$ and $\sigma(x) = \sigma(g')$*

Proof. The equivalences follow immediately from the construction of σ'', stated in rule 8. □

Clearly the above semantics can be represented as a *pushdown system* (PDS). A pushdown system is a triplet $\mathcal{P} = (Q, \Gamma, \Delta)$ where Q is a finite set of *control locations*, Γ is a finite *stack alphabet*, and $\Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$ is a finite set of *productions*. A transition $(q, \gamma, q', \bar{\gamma})$ is enabled if control is at location q and γ is at the top of the stack then control can move to location q' by replacing γ by the possible empty work of stack symbols $\bar{\gamma}$.

In our case, for a given program $p_1 :: B_1, \ldots, p_n :: B_n$, the set of control locations is defined by the finite abstract state space $V \longrightarrow |V| + 1$. In order to define the stack alphabet we introduce the finite set $\bigcup_{i=1}^{k} cl(B_i)$ of possible reachable statements where the closure of a statement B, denoted as $cl(B)$, is defined as follows.

- $cl(x := y) = \{x := y\}$
- $cl(x := \text{new}) = \{x := \text{new}\}$
- $cl(B; B) = cl(B) \cup cl(B)$

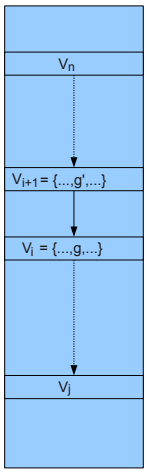

Fig. 1. Chain in a Stack

- $cl([x = y]B) = \{[x = y]B\} \cup cl(B)$
- $cl([x \neq y]B) = \{[x \neq y]B\} \cup cl(B)$
- $cl(B + B) = cl(B) \cup cl(B)$
- $cl(p) = \{p\}$

The stack alphabet Γ is then defined by the union of the abstract state space and the above set of possible reachable statements. Finally, it is straightforward to transform the rules of the above semantics into rules of a pushdown system, simply by removing the common stack tail from the left- and righthand sides.

5 Equivalence between the two models

In this section the behavioural equivalence between the two models is shown by establishing *bisimilarity*, which is widely accepted as the finest behavioural equivalence one would want to impose. A (binary) symmetric relation $\mathcal{R}$ on the states of a transition system which satisfies

$$\text{if } P \longrightarrow P' \text{ then there is a } Q' \text{ such that } Q \longrightarrow Q' \text{ and } (P', Q') \in \mathcal{R},$$

is called a *bisimulation relation* [20].

This definition applies to a single transition system – in our case, we use it to establish equivalence between the two models. The states of the transition system are pairs of configurations, and the transitions are execution steps of the respective models.

We first define the following relation between abstract and concrete states.

Definition 1. *We define $s \sim \sigma$ by $s(x) = s(y)$ iff $\sigma(x) = \sigma(y)$, for every pair of variables x and y.*

Next we extend this relation to stacks and configurations as follows.

Definition 2. *We define $S \sim \Sigma$ inductively by*

- *if S and Σ are both empty then $S \sim \Sigma$*
- *if $S \sim \Sigma$ then $B \bullet S \sim B \bullet \Sigma$, for any statement B*
- *if $s \sim \sigma$ and $S \sim \Sigma$ then $s \bullet S \sim \sigma \bullet \Sigma$*

We define $\langle s, S\rangle \sim \langle \sigma, \Sigma\rangle$ by $s \sim \sigma$ and $S \sim \Sigma$.

In order to prove equivalence of the concrete and abstract semantics, we introduce the freeze variables also as *auxiliary* variables into the concrete semantics. We do so by implicitly assuming that the rule for procedure calls additionally initializes each freeze variable to the value of its corresponding global variable. Note that this does not affect the behaviour of the program (which is assumed not to contain freeze variables). It therefore suffices to relate this concrete semantics extended with freeze variables and the abstract semantics.

Theorem 1. *The above relation $\langle s, S\rangle \sim \langle \sigma, \Sigma\rangle$ is a bisimulation relation for* reachable *configurations $\langle s, S\rangle$ for which there exists an initial configuration $\langle s_0, p_0\rangle$ such that $\langle s_0, p_0\rangle \longrightarrow^* \langle s, S\rangle$.*

Proof. Let $\langle s, S\rangle \sim \langle \sigma, \Sigma\rangle$, where $\langle s, S\rangle$ is a reachable configuration. We must show that for every execution step applicable to one configuration, there is an execution step for the other configuration such that the resulting configurations are again related by $\sim$.

If the top of the stack is any statement except $S + S$, it uniquely determines the next step for both models. We choose the same step for both models for the case $S + S$, so we can consider the resulting configurations of applying an execution step to both configurations. If the execution steps have preconditions (rules 2, 4, 5) then satisfaction of these preconditions must be equivalent in s and σ. It is easy to see this follows from the definition of the relation $\sim$ on states. Now we can establish execution steps $\langle s, S\rangle \longrightarrow \langle s', S'\rangle$ and $\langle \sigma, \Sigma\rangle \longrightarrow \langle \sigma', \Sigma'\rangle$. It rests to prove that the resulting configurations are again equivalent –

$$\langle s', S'\rangle \sim \langle \sigma', \Sigma'\rangle$$

must hold.

We prove the equivalence by considering all semantic rules. We consider the main rules for object creation, procedure calls and returns.

Rule 3 ($z :=$ new). For variables x and y distinct from z, we have $s'(x) = s(x)$, $\sigma'(y) = \sigma(y)$, $s'(x) \neq s'(z)$ and $\sigma'(x) \neq \sigma'(z)$. This proves $s' \sim \sigma'$. Next observe that S' and Σ' equals S and Σ, respectively. So we obtain the desired result.

Rule 7 (call p). By definition we have $s'(l) = \bot$ and $\sigma'(l) = 0$, for every local variable l, and $s(g) = s'(g)$ and $\sigma(g) = \sigma'(g)$, for every global variable g. It

follows that $s \sim \sigma$ implies $s' \sim \sigma'$. Further, by definition S' and Σ' equals $s \bullet S$ and $\sigma \bullet \Sigma$, respectively. By assumption, $s \sim \sigma$ and $S \sim \Sigma$, and so by definition $s \bullet S \sim \sigma \bullet \Sigma$.

Rule 8. By definition S and Σ equals $s'' \bullet S'$ and $\sigma'' \bullet \Sigma'$, respectively, for some states s'' and σ''. From the assumption $S \sim \Sigma$ it thus follows that $s'' \sim \sigma''$ and $S' \sim \Sigma'$. Remains to prove that $s' \sim \sigma'$. We distinguish the following three cases:

1. x and y are both global variables:

$$\begin{aligned} s'(x) = s'(y) &\overset{\text{(Corollary 1.1)}}{\iff} s(x) = s(y) \\ &\overset{\text{(Assumption)}}{\iff} \sigma(x) = \sigma(y) \\ &\overset{\text{(Corollary 2.1)}}{\iff} \sigma'(x) = \sigma'(y) \end{aligned}$$

2. x and y are both local variables:

$$\begin{aligned} s'(x) = s'(y) &\overset{\text{(Corollary 1.2)}}{\iff} s''(x) = s''(y) \\ &\overset{\text{(Assumption)}}{\iff} \sigma''(x) = \sigma''(y) \\ &\overset{\text{(Corollary 2.2)}}{\iff} \sigma'(x) = \sigma'(y) \end{aligned}$$

3. x is global and y is local:

$$\begin{aligned} s'(x) = s'(y) &\overset{\text{(Corollary 1.3)}}{\iff} s(x) = s''(y) \\ &\overset{\text{(Lemma 1)}}{\iff} s(x) = s''(g) \text{ and } s''(g) = s''(y), \text{ for some global variable } g \\ &\overset{\text{(Freeze var.)}}{\iff} s(x) = s(g') \text{ and } s''(g) = s''(y), \text{ for some global variable } g \\ &\overset{\text{(Assumption)}}{\iff} \sigma(x) = \sigma(g') \text{ and } \sigma''(g) = \sigma''(y) \text{ for some global variable } g \\ &\overset{\text{(Corollary 2.3)}}{\iff} \sigma'(x) = \sigma'(y) \end{aligned}$$

   Note that because of the introduction of freeze variables in the concrete semantics we indeed have $s''(g) = s(g')$ (this can be proved in a straightforward manner by induction on the length of the computation).

This concludes the proof of Theorem 1. □

6 Conclusions

Pushdown systems naturally model the control flow of sequential computation in programming languages with local variables and recursive procedures. In this paper we provided a generalization of this model by adding unbounded object creation. We have shown that imperative programs with object creation, recursive procedures, and local variables without any restriction can be given a symbolic semantics through a finite pushdown system such that the infinite state system generated is strongly bisimilar to the ordinary operational semantics of the program.

Applications to static analysis. Starting from an initial stack containing the initial procedure p_0, a program P is executed and eventually terminates when the stack is empty. If we consider a singleton alphabet symbol labeling all transitions of our pushdown system, we obtain an ordinary pushdown automaton (with acceptance by empty stack). Clearly, the language accepted by this pushdown automaton is non-empty if and only if there exists an execution of the program P that terminates. Since the emptiness problem is decidable for pushdown automata [16], we have an algorithm for deciding termination of programs in our language. Similarly, because the halting problem for pushdown automata is decidable, we have an algorithm for deciding if a program blocks, for example because of an assignment with an undefined variable at the right-hand side.

Applications to model checking. More recently, the problem of checking ω-regular properties (like those expressible in linear-time temporal logics or linear-time μ-calculus) or properties expressed as formulas of the alternation-free modal μ-calculus (including CTL properties) of pushdown systems have been shown to be decidable, leading to efficient model checkers for the generated infinite state systems (see e.g. [5,12]). For instance, to verify whether a program P in our language satisfies a linear time temporal formula ϕ, we first derive a symbolic pushdown system for P with finitely many control states and stack symbols, then construct the finite state Büchi automaton for the negation of ϕ, and finally use the algorithm of [5] to check if there is no execution of the program P that satisfies the negation of ϕ. Interestingly, the complexity of this model checking problem for a fixed LTL formula is polynomial in the size of the pushdown system, a complexity that is not much worse than that for finite transition systems [5].

In the future we plan to investigate the integration of our technique with jMoped, a Java model checker based on pushdown systems [12]. As for the model checking, there are at least two directions that could be explored. On the one hand we intend to look for extension of temporal logic with support for a primitive for object creation (and destruction) [11,3]. On the other hand, we would like to investigate model checking of some non ω-regular properties, allowing, for example, matching of procedure calls and returns. While the problem is in general undecidable, it seems possible to turn our pushdown systems into visibly pushdown automata, a class of pushdown automata with desirable closure properties and interesting tractable decision problems [1].

Language considerations. We have presented a language that supports unbounded object creation by using recursive procedures with global and local variables. The language can not be extended with higher-order features like passing procedures and internal procedures as parameters of procedure calls, as well as it cannot include features like call-by-name parameter passing because the halting problem for these two class of programs is known to be undecidable [7]. It would be interesting, however, to see what happens if we change static scope to dynamic scope or if we disallow internal procedures as parameters.

Our language does not have any concrete data but for object identities, and does not support object fields. Data can be added but in order to model computations by a *finite* pushdown system, we need to consider only finite data domains. The language can be extended with object fields $f_1, \ldots, f_n$, by simply adding expressions of the form $x.f$ as variables, and as such they will be included in the partitions. More general navigation expressions can be reduced to the above in the obvious way.

The language does not have a syntactic construct to destroy object identities. We can give a concrete semantic for it without the needs of inspecting the call-stack (for example by storing in extra variables the names of the objects destroyed and assuming they will not be reused, so that local variables in the stack can be reset when a procedure returns). This observation can be combined with the concept of chains in the stack of variables referring to the same object to allow deletion within the pushdown system representation, by simply keeping track of the chains which refer to deleted objects. This way of deletion would work also for encoded object fields, which implies on-the-fly garbage collection. We plan to work out the details in a future work.

Finally, our language is sequential. It is not a problem to add bounded concurrency within the body of a procedure by using, e.g. a parallel operator of the form $B_1||B_2$, as we can give an interleaving semantic to it using rules of the form

$$\frac{\langle s, B_1 \bullet S\rangle \longrightarrow \langle s, B \bullet S\rangle}{\langle s, B_1||B_2 \bullet S\rangle \longrightarrow \langle s, B||B_2 \bullet S\rangle} \quad \text{and} \quad \frac{\langle s, B_2 \bullet S\rangle \longrightarrow \langle s, B \bullet S\rangle}{\langle s, B_1||B_2 \bullet S\rangle \longrightarrow \langle s, B_1||B \bullet S\rangle}$$

However for more global notions of concurrency, like threads, we need to store the local variables of the program for each thread. Therefore, to keep the stack alphabet and the number of control states finite in our pushdown system, we have to restrict to a bounded number of threads [2]. It can be interesting to combine our results with those of [6], so to allow reachability analysis of multithreaded programs.

We leave these considerations for future work.

References

1. R. Alur, P. Madhusudan. Visibly pushdown languages. In *Proc. of Annual ACM Symposium on Theory of Computing (STOC 2004)*, pages 202-211, ACM, 2004.
2. F.S. de Boer and I. Grabe. Automated Deadlock Detection in Synchronized Reentrant Multithreaded Call-Graphs. In *Proc. of 36th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2010)*, volume 5901 of *Lecture Notes in Computer Science*, pages 200-211, Springer, 2010.
3. M.M. Bonsangue, A.Kurz. Pi-Calculus in Logical Form, In *Proc 22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pp. 303-312, IEEE, 2007.
4. C. Baier and J.-P. Katoen. *Principles of Model Checking* The MIT press, 2008.
5. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking In *Proceedings Concur 97*, volume 1243 of *Lecture Notes in Computer Science*, pp. 135–150, Springer, 1997.

6. A. Bouajjani, S. Fratani, S. Qadeer. Context-Bounded Analysis of Multithreaded Programs with Dynamic Linked Structures. In *Proc. Intern. Conf. on Computer Aided Verification (CAV'07* volume 4590 of *Lecture Notes in Computer Science*, Springer 2007.
7. E.M. Clarke. Programming language constructs for which it is impossible to obtain good Hoare-like axioms. *Journal of the ACM* 26:126-147, 1979.
8. J. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings 22nd International Conference on Software Engineering*, pp. 439-448. IEEE Computer Society, 2000.
9. C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, 29(7):577–603, 1999.
10. E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall Series in Automatic Computation, 1976.
11. D. Distefano, J.-P. Katoen, A. Rensink. Who is Pointing When to Whom? In *Proceedings of 24th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004)* volume 3328 of *Lecture Notes in Computer Science*, pp. 250-262, Springer 2004.
12. J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Proceedings of CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pp. 324–336, Springer, 2001.
13. M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In *Proceedings of 14th LICS*, pp. 214-224, IEEE Computer Society Press, 1999.
14. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366-381, 2000.
15. G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–94, 1997.
16. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 3rd edition, 2006.
17. U. Montanari and M. Pistore. An Introduction to History Dependent Automata. In *Proceeding 2nd Workshop on Higher-Order Operational Techniques in Semantics*, volume 10 of *Electonic Notes in Theoretical Computer Science*, pp. 170-188, Elsevier, 1998.
18. D. Park, U. Stern, J. Skakkebaek, and D. Dill. Java Model Checking In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering.* pp. 253–256. IEEE, 2000.
19. B. Randell and L.J. Russell. *ALGOL 60 Implementation: The Translation and Use of ALGOL 60 Programs on a Computer.* Academic Press, 1964.
20. D. Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science* 8(5):447-479, 1998.
21. S. Schwoon. Model-checking pushdown systems. PhD thesis, Technische Universität München, 2002.
22. R. Sebesta. *Concepts of Programming Languages.* Addison-Wesley, 9th edition, 2009.
23. B. Stroustrup *The C++ Programming Language.*