Unbounded Allocation in Bounded Heaps

Jurriaan Rot, Frank De Boer, Marcello Bonsangue

Unbounded Allocation in Bounded Heaps

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

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

Abstract. In this paper we introduce a new *symbolic* semantics for a class of recursive programs which feature dynamic creation and unbounded allocation of objects. We use a symbolic representation of the program state in terms of equations to model the semantics of a program as a pushdown system with a *finite* set of control states and a *finite* stack alphabet. Our main technical result is a rigorous proof of the equivalence between the concrete and the symbolic semantics.
Adding pointer fields gives rise to a Turing complete language. However, assuming the number of reachable objects in the visible heap is bounded in all the computations of a program with pointers, we show how to construct a program without pointers that simulates it. Consequently, in the context of bounded visible heaps, programs with pointers are no more expressive than programs without them.

1 Introduction

In this paper we investigate the interplay between dynamic creation of objects and recursion. To this end we introduce a core programming language which features dynamic object creation, global variables, static scope and recursive methods with local variables, but which does not include (abstract) pointers. In order to focus on the main issue of dynamic object creation in the context of recursion, we further restrict the data types to that of objects. Other *finite* data domains could have been added without problem, but would have increased the complexity of the model without strengthening our main result.

We first define a concrete operational semantics for our language based on a standard implementation of recursion using a stack. This semantics uses an explicit representation of objects which immediately gives rise to an infinite name space because an unbounded number of objects can be stored on the stack using local variables. Consequently, decidability results for pushdown systems (for which the stack alphabet is finite) and existing model checking techniques of pushdown systems against temporal formulas [2] are not applicable.

Our solution is to abstract from the concrete representation of objects by representing states, i.e., assignments of objects to the program variables, symbolically as conjunctions of equations over the program variables, identifying those

⋆ The research of this author has been funded by the Netherlands Organisation for Scientific Research (NWO), CoRE project, dossier number: 612.063.920.

variables which refer to the same object. In this symbolic setting we show how to describe the basic computation steps, e.g., object allocation, recursive calls and returns, in terms of a strongest postcondition semantics. The resulting symbolic semantics can be modeled formally as a pushdown system with a *finite* set of control states and a *finite* stack alphabet. Our main technical result is a rigorous proof of the equivalence between the concrete and the symbolic semantics.

Adding (abstract) pointers however allows to program dynamically linked data structures, like lists or trees, and allows the simulation of a 2-counter machine. As such, reachability is undecidable [9] for this extended language. We show in this paper that if a (recursive) program with pointers gives rise to computations in which the number of reachable objects in the heap is bounded a priori, then it can be simulated by a program without pointers. Therefore, in the context of bounded visible heaps, (recursive) programs with pointers are no more expressive than programs without them.

Related work. Our main contribution is a new symbolic semantics and a corresponding direct proof of decidability of a class of recursive programs which feature dynamic creation and unbounded allocation of objects, but do not include (abstract) pointers. The decidability of our core language itself follows from the general result of [3] for recursive programs with (abstract) pointers which generate only bounded visible heaps. On the other hand, the general result of [3] concerning pointers can be derived from our basic decidability result by our simulation of programs with pointers. In fact, our simulation shows that restricting to bounded (visible) heaps basically boils down to restricting to programs without pointers! Moreover, the general result of [3] is based on a complex mechanism for "merging" upon the return of a method the local part of the "old" heap (the heap before the call) and the current heap (see [11] for a more extensive discussion). This is because a reuse is required in order to model the semantics as a pushdown system with a finite stack alphabet (as explained above, by means of the local variables an unbounded number of objects can be stacked). In contrast, in the absence of pointers in our strongest postcondition semantics such name clashes are resolved symbolically by a simple substitution of fresh variables. The underlying equational logic then allows for a simple elimination of these implicitly existentially quantified variables.

More recently, [1] introduces an algorithm for reachability in pushdown automata with gap-ordered constraints, a model which allows to represent the behavior of our language. The symbolic semantics introduced in this paper is based on ordinary pushdown systems and therefore we can use standard algorithms for model checking [2]. Furthermore, it is similar in spirit to the one used in high level allocation Büchi automata [6] for model checking of a possibly unbounded number of objects with pointers but for a language with a *restricted form of recursion* (tail recursion) and *no block structure*. Full recursion, but with a *fixed-size number of objects* is instead considered in jMoped [7], using a pushdown structure to generate an infinite state system. Similar and even stronger restrictions limiting either size of heap and stack, or the number of objects are considered by current model checkers for object-oriented languages,

such as Java Path Finder [8], JCAT [5], Bandera [4] (possibly combined with the Symbolic Analysis Laboratory model checker [10]).

Plan of the paper. 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 with infinite states and a symbolic model based on pushdown systems. We end the section by studying the relationship between these two models. In Section 4 we discuss the extension with pointers. Finally, the last section discusses some possible future research and tool development.

2 A core language for allocation

We introduce a core programming language that focuses on dynamic allocation, global and local variables, and recursive procedures. To simplify the presentation it is restricted to a single data structure, that of objects. Objects can be dynamically allocated and referenced. A program consists of a finite set of procedures, each acting on some global and local state. Procedures can store object references 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, v, w 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. We often write $\bar{l}$ for the sequence of all local variables. We assume a distinguished element $\mathsf{nil} \in G$, used as a constant to refer to the undefined object. For a finite set P 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:

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

The language is statically scoped. The *assignment* statement $x := y$ assigns the reference stored in y (if any) to x. The statement $x := \mathsf{new}$ *allocates* a new object that will be referenced by the program variable x. As for the ordinary assignment, the old value of x is lost. For this reason we will consider only programs in which the variable nil does not appear at the left-hand side of an assignment or allocation statement. *Sequential composition* $B_1; B_2$, *guarded statements* $[x = y]B$ and $[x \neq y]B$, and *nondeterministic choice* $B_1 + B_2$ have the standard interpretation. Execution of a *procedure call* $\mathsf{call}\, p$ consists of the execution of the associated body B with its local variables initialized to nil. Upon termination of the body B of a procedure the previous local state (from which the procedure has been called) is restored.

Example 1. As a basic example of storing an unbounded number of objects, consider the procedure

$$p \ :: \ l := \mathsf{new};\ ((\mathsf{call}\, p;\ g_1 := l) + g_2 := l)$$

A call to p allocates a number of objects not bounded a-priori and stores them in the several copies of the local variable l in the call-stack. The global variable

g_2 refers to the last allocated object (or nil), whereas g_1 refers to the first one (or nil). Note that if all variables are initially nil, then $g_1 \neq g_2$ eventually holds if and only if the program terminates.

The usual while, skip, if-then-else statements, and more general boolean expressions, can easily be encoded in this sequential setting. Procedures with call-by-value parameters and return values can also be modeled using assignments to global variables.

3 A concrete and a symbolic 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 allocation. A *program state* is a variable assignment $s\colon V \longrightarrow \mathbb{N}$, where 0 is used to represent the "undefined" object, and thus we assume $s(\mathsf{nil}) = 0$. To model allocation we distinguish a global "system" variable cnt which is used as a counter, and is not used by programs. We implicitly assume that $s(x) < s(\mathsf{cnt})$, for every state s and variable x different from cnt. Note that this implies that $s(\mathsf{cnt}) \neq 0$, as this is the value of nil.

A *configuration* of a program is a tuple $\langle s, S\rangle$ where s is the current program state and S is a stack of statements and stored return states. The current statement to be executed is on the top of the stack. An *execution step* of a program is a transition from a configuration C to a configuration C', denoted by $C \longrightarrow C'$. A computation is a (possibly infinite) sequence $C_1 \longrightarrow C_2 \longrightarrow \ldots$ of execution steps. The possible execution steps are given below. For modeling function updates we use multiple assignments of the form $f[x_1, \ldots, x_k := y_1, \ldots, y_k]$, where x_i and x_j are distinct elements of the domain of f for $i \neq j$, and all y_i's are in the codomain of f. It denotes the function mapping x_i to y_i if $i \in \{1, \ldots, k\}$, and otherwise x is mapped to the old value $f(x)$. The head of a stack is separated from the tail by means of the right-associative operator $\bullet$: for example, $B \bullet S$ is a stack consisting a statement B and tail S, whereas $s \bullet S$ is a stack consisting a state s as head and tail S.

When an *assignment* is the current statement to be executed, then the current program state is updated accordingly. The tail of the stack is not changed.

$$\langle s, x := y \bullet S\rangle \longrightarrow \langle s[x := s(y)], S\rangle$$

Dynamic allocation is similar to an assignment, but it uses the system variable cnt, which is now increased.

$$\langle s, x := \mathsf{new} \bullet S\rangle \longrightarrow \langle s[x, \mathsf{cnt} := s(\mathsf{cnt}), s(\mathsf{cnt}) + 1], S\rangle$$

The execution of the *sequential composition* of two statements updates the stack so that they are executed in the right order.

$$\langle s, B_1; B_2 \bullet S\rangle \longrightarrow \langle s, B_1 \bullet B_2 \bullet S\rangle$$

Guarded statements are executed only if the current program state satisfy their respective conditions, otherwise they block.

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

A *non-deterministic choice* updates the stack so that only one of the two statements becomes the current one.

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

In a *procedure call* the entire current state is pushed onto the stack so that the local variables can be restored when the procedure returns (we push the entire state only for notational convenience, to avoid irrelevant case distinctions). In the new current state all local variables are set to 0 and the current program on the stack becomes the body of the called procedure.

$$\langle s, \mathsf{call}\, p_i \bullet S\rangle \longrightarrow \langle s[\bar{l} := \bar{0}], B_i \bullet s \bullet S\rangle$$

Here B_i is the body of p_i, $\bar{l}$ denotes the sequence of local variables $l_1, \ldots, l_m$ and $\bar{0}$ is a sequence of length m of 0's, where m is the number of local variables.

When the top of the stack is a program state, a *procedure return* is executed. The local variables are restored using the state stored on the stack.

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

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

3.1 A symbolic semantics

The concrete semantics introduced in the previous section clearly cannot be modeled as a pushdown system because it uses the infinite set of natural numbers to represent objects. However, at any moment of a computation, only finitely many objects are referenced by program variables. In this section we exploit this basic fact and represent a program state *symbolically* as a finite conjunction of equalities, identifying program variables referring to the same object. Subsequently, we define program steps based on a strongest postcondition calculus, which requires the introduction of fresh global variables for assignments. It suffices to assume a distinct logical variable z for each variable in V. We denote by Var the set of program variables V extended with their logical variables. Note that Var is finite. A *symbolic state* φ is a finite conjunction of equalities as given by the grammar

$$\varphi \;::=\; x = y \;\mid\; \varphi \wedge \varphi$$

where x and y range over Var. A symbolic state φ gives rise to a relation $r(\varphi) \subseteq \mathit{Var} \times \mathit{Var}$, inductively defined by

$$r(x = y) = \{(x, y)\} \qquad r(\varphi_1 \wedge \varphi_2) = r(\varphi_1) \cup r(\varphi_2)\,.$$

Next we let $r^*(\varphi) \subseteq \mathit{Var} \times \mathit{Var}$ denote the reflexive, symmetric and transitive closure of $r(\varphi)$. This way, any symbolic state φ gives rise to a partioning of Var. We define $\varphi \models x = y$ if and only if $(x, y) \in r^*(\varphi)$, and extend it to disequalities by the closed world assumption, i.e., $\varphi \models x \neq y$ iff $\varphi \not\models x = y$.

We describe the effect of a basic statement S on a symbolic state φ in terms of its strongest postcondition $SP(S, \varphi)$, i.e., $SP(S, \varphi)$ will be the strongest state formula such that whenever we start from a program state satisfying φ (under the obvious satisfaction relation), after executing S the resulting state satisfies $SP(S, \varphi)$. We denote by $t[z/x]$ the syntactic substitution of x in t by a corresponding fresh (logical) variable z.

$$SP(x := \mathsf{new}, \varphi) = \varphi[z/x]$$
$$SP(x := y, \varphi) = \varphi[z/x] \wedge x = (y[z/x])$$

Note that z in the above two clauses represents the "old" value of x. In an assignment statement $x := y$ as well as in in a dynamic allocation $x := \mathsf{new}$ we have for all variables v, w (syntactically) different from x that $\varphi \models v = w$ if and only if $SP(x := y, \varphi) \models v = w$. Further, for dynamic allocation, $SP(x := \mathsf{new}, \varphi) \models x = y$ if and only if x and y are the same variable (and thus by the closed world assumption $SP(x := \mathsf{new}, \varphi) \models x \neq y$, for every y different from x).

For procedure calls, we assume without loss of generality that for each global variable $g \in G$ there exists a unique *local* variable $g' \in L$ which does not occur in the given program. These so-called *freeze* variables are used to represent locally the global variables *before* a call. This information is needed to relate logically the state before the call and the return state. In the strongest postcondition of a procedure call we model logically the initialization of the local program variables (thus not the freeze variables) to nil, and the assignment to each freeze variable g' of the value of its corresponding global variable g:

$$SP(\mathsf{call}\, p, \varphi) = \varphi[\bar{z}/\bar{l}] \wedge \bigwedge_{g \in G} (g = g') \wedge \bigwedge_{l \in L'} l = \mathsf{nil}\,,$$

where $\bar{z}$ is a sequence of logical variables corresponding to a sequence $\bar{l}$ of all local variables appearing in φ; $\bar{z}$ is used to represent the old values of $\bar{l}$. L' is the set of local variables that are not freeze variables.

To describe the strongest postcondition of the return of a procedure we introduce an auxiliary statement "$\mathsf{ret}\, \psi$" for each symbolic state ψ. The global variables in ψ thus represent the old values of the global variables *before* the call. On the other hand, in the current symbolic state φ the old equalities between the global variables before the call are represented by their corresponding freeze variables. We can identify these simply by replacing the freeze variables in φ and the global variables in ψ by the *same* logical variables. This explains the main idea underlying the following rule:

$$SP(\mathsf{ret}\, \psi, \varphi) = \varphi[\bar{z}/\bar{g}'][\bar{z}'/\bar{l}] \wedge \psi[\bar{z}/\bar{g}]$$

where $\bar{z}$ and $\bar{z'}$ are disjoint sequences of fresh logical variables, $\bar{g}'$ is the sequence of freeze variables and $\bar{l}$ is the sequence of all local variables. So in φ, first the

freeze variables are renamed to $\bar{z}$, and then the other local variables in φ, which are no longer valid, are renamed away into fresh logical variables $\bar{z}'$.

We use the above postcondition calculus to define a symbolic semantics for our programs. An *(abstract) configuration* of a program is a pair $\langle \varphi, \mathcal{S} \rangle$ where φ is a symbolic state *restricted* to the program variables V and $\mathcal{S}$ is a stack of statements and symbolic states also restricted to the program variables V. This restriction is justified because logical variables are implicitly existentially quantified and as such can be *eliminated* (in each step): for any symbolic state φ we can construct a formula $\varphi \downarrow_V$ which only contains variables in V such that $r^*(\varphi)$ restricted to $V \times V$ equals $r^*(\varphi \downarrow_V)$.

Now for dynamic allocation and assignment statements we lift the strongest postconditions defined above to transitions as follows:

$$\langle \varphi, B \bullet \mathcal{S} \rangle \longrightarrow \langle SP(B, \varphi) \downarrow_V, \mathcal{S} \rangle \tag{1}$$

where B is either $x := \mathsf{new}$ or $x := y$ for some program variables x and y.

The transition rules for sequential composition, non-deterministic choice and guarded statements are similar to the corresponding transition rules in the concrete semantics. As an illustration, we give below the two rules for guarded statements:

$$\frac{\varphi \models x = y}{\langle \varphi, [x = y]B \bullet \mathcal{S} \rangle \longrightarrow \langle \varphi, B \bullet \mathcal{S} \rangle} \qquad \frac{\varphi \models x \neq y}{\langle \varphi, [x \neq y]B \bullet \mathcal{S} \rangle \longrightarrow \langle \varphi, B \bullet \mathcal{S} \rangle} \tag{2}$$

On a procedure call p_i, we push the procedure body and current state onto the stack:

$$\langle \varphi, \mathsf{call}\, p_i \bullet \mathcal{S} \rangle \to \langle SP(\mathsf{call}\, p_i, \varphi) \downarrow_V, B_i \bullet \varphi \bullet \mathcal{S} \rangle\,. \tag{3}$$

The transition for procedure return is similar to that of an assignment:

$$\langle \varphi, \psi \bullet \mathcal{S} \rangle \to \langle SP(\mathsf{ret}\, \psi, \varphi) \downarrow_V, \mathcal{S} \rangle\,. \tag{4}$$

Example 2. As a simple example of a symbolic computation, consider the procedure declaration

$$p :: l := \mathsf{new}$$

where l is a local variable. Let g be some global variable. We will consider the execution of the statement $\mathsf{call}\, p$ starting from a symbolic state where g equals l. During the execution of p, l is assigned a new object; however, since l is a local variable, it is restored when the procedure returns, so then we should again have that g is equal to l. We restrict the above definition of the strongest postcondition of a procedure call to the local variable l and some global variable g; then $SP(\mathsf{call}\, p, l = g)$ is

$$z = g \wedge g = g' \wedge l = \mathsf{nil}\,,$$

and thus by eliminating the logical variable z we derive the transition step

$$\langle l = g, \mathsf{call}\, p \rangle \longrightarrow \langle g = g' \wedge l = \mathsf{nil}, l := \mathsf{new} \bullet l = g \rangle\,.$$

Next we compute $SP(l := \mathsf{new}, g = g' \wedge l = \mathsf{nil})$:

$$g = g' \wedge z = \mathsf{nil}\,,$$

and again eliminating the logical variable z we now derive the transition

$$\langle g = g' \wedge l = \mathsf{nil}, l := \mathsf{new} \bullet l = g\rangle \longrightarrow \langle g = g', l = g\rangle\,.$$

As above, restricting the above definition of $SP(\mathsf{ret}\, l = g, g = g')$ to the local variable l and the global variable g, we obtain

$$g = z \wedge l = z\,.$$

Finally, eliminating the logical variable z we arrive at the final transition

$$\langle g = g', l = g\rangle \longrightarrow \langle l = g, \epsilon\rangle\,,$$

where indeed l and g are again identified.

The above semantics gives rise to a finite *pushdown system*. A pushdown system is a triple $\mathcal{P} = (Q, \Gamma, \Delta)$ 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 word 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 given by the set of state formulas restricted to V. 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:

$$\begin{array}{ll} cl(A) = \{A\} & cl([x = y]B) = \{[x = y]B\} \cup cl(B) \\ cl(B_1; B_2) = \{B_1; B_2\} \cup cl(B_1) \cup cl(B_2) & cl([x \neq y]B) = \{[x \neq y]B\} \cup cl(B) \\ cl(B_1 + B_2) = \{B_1 + B_2\} \cup cl(B_1) \cup cl(B_2) & \end{array}$$

where A is an assignment, an allocation or a procedure call. 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. For a pushdown system both the halting problem and reachability are decidable. 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 [2, 7].

3.2 Correctness of the symbolic semantics

In this section we show that the concrete and the abstract semantics are equivalent. First we identify the relevant properties of the concrete semantics satisfied by

any reachable configuration. Basically, (1) on a procedure call, all local variables must be initialized to 0. Thus if an object is referenced by a variable in the current state s and in a stacked state s', then there must be a global variable referencing it in that stacked state s'. Moreover (2) the system variable cnt is greater than all objects currently referenced by variables stored somewhere in the stack. Formally this is the content of the next definition.

Definition 3. *A stack S is* proper *if either S is the empty stack, or $S = B \bullet S'$ for some statement B and proper stack S', or $S = s \bullet S'$ for some program state s and proper stack S' such that for any state s' occurring in S':*

$$(1)\ s(V) \cap s'(V) \subseteq s'(G),$$
$$(2)\ \forall v \in V. s(\mathsf{cnt}) > s'(v).$$

A configuration $\langle s, S\rangle$ is proper if $s \circ S$ is proper.

Properness is preserved by all computation steps:

Lemma 4. *If $\langle s, S\rangle$ is proper and $\langle s, S\rangle \to \langle s', S'\rangle$ then $\langle s', S'\rangle$ is proper.*

For example, the configuration $\langle s, p_0\rangle$ is proper, where p_0 is the main procedure name, and s is the state mapping cnt to 1 and all other variables (including nil) to 0. From the above Lemma, any configuration in a computation starting from this initial one is a proper configuration.

Next we give some basic properties of the concrete semantics of the procedure return. Informally, global variables are not affected by a procedure returns, and local variables get the values they had before the procedure call.

Lemma 5. *If $\langle s, s' \bullet S\rangle \longrightarrow \langle s_r, S\rangle$ then for every $x, y \in V$:*

1. $x, y \in G \Rightarrow (s_r(x) = s_r(y)$ *iff* $s(x) = s(y))$
2. $x, y \in L \Rightarrow (s_r(x) = s_r(y)$ *iff* $s'(x) = s'(y))$
3. $x \in G, y \in L \Rightarrow (s_r(x) = s_r(y)$ *iff* $s(x) = s'(y))$

We proceed with the following relevant properties of the symbolic semantics. First we make precise what is the relation between global variables in the caller's state ψ and freeze variables in the callee's state φ: freeze variables can be equal if and only if their corresponding global variables were equal in the first place.

Definition 6. *We define $\psi \triangleright \varphi$ iff for any two globals $g_1, g_2 \in G$,*

$$\varphi \models g_1' = g_2' \text{ iff } \psi \models g_1 = g_2$$

The definition of properness of abstract configurations is based on the above.

Definition 7. *A stack $\mathcal{S}$ (of symbolic states and statements) is* proper *if either $\mathcal{S}$ is the empty stack, or $\mathcal{S} = B \bullet \mathcal{S}'$ for some statement B and proper stack $\mathcal{S}'$, or $\mathcal{S} = \varphi \bullet \mathcal{S}'$ for some symbolic state φ and proper stack $\mathcal{S}'$ such that for the topmost state φ' occurring in $\mathcal{S}'$, if it exists: $\varphi' \triangleright \varphi$. An abstract configuration $\langle \varphi, \mathcal{S}\rangle$ is proper if $\varphi \bullet \mathcal{S}$ is proper.*

Properness of abstract configurations is preserved by all computation steps:

Lemma 8. *If $\langle \varphi, \mathcal{S} \rangle$ is proper and $\langle \varphi, \mathcal{S} \rangle \to \langle \varphi', \mathcal{S}' \rangle$ then $\langle \varphi', \mathcal{S}' \rangle$ is proper.*

We state the main properties of the procedure return in the abstract semantics.

Lemma 9. *If $\langle \varphi, \psi \bullet \mathcal{S} \rangle$ is proper, and $\langle \varphi, \psi \bullet \mathcal{S} \rangle \longrightarrow \langle \varphi_r, \mathcal{S} \rangle$ then for every $x, y \in V$:*

1. *$x, y \in G$ implies $\varphi_r \models x = y$ iff $\varphi \models x = y$,*
2. *$x, y \in L$ implies $\varphi_r \models x = y$ iff $\psi \models x = y$,*
3. *$x \in G$, $y \in L$ implies $\varphi_r \models x = y$ iff $\exists g \in G$ s.t. $\psi \models y = g$ and $\varphi \models x = g'$*

In order to prove the equivalence between the symbolic semantics and the concrete semantics, we extend the latter so that each procedure body starts with the initialization of the freeze variables to their corresponding global variables. Note that this does not affect the behaviour of a program, since freeze variables by assumption do not occur in it.

Let s be a concrete state, and φ a symbolic state. We define $s \sim \varphi$ if and only if for all variables $x, y \in V$: $s(x) = s(y)$ iff $\varphi \models x = y$. Next this relation is extended to stacks S of statements and (concrete) program states and stacks $\mathcal{S}$ of statements and symbolic states. We define $S \sim \mathcal{S}$ iff S and $\mathcal{S}$ are proper stacks, and one of the following cases hold:

1. S, $\mathcal{S}$ are both empty
2. $S = B \bullet S'$ and $\mathcal{S} = B \bullet \mathcal{S}'$ for some statement B, and $S' \sim \mathcal{S}'$
3. $S = s' \bullet S'$, $\mathcal{S} = \varphi \bullet \mathcal{S}'$, $s \sim \varphi$ and $S' \sim \mathcal{S}'$

Finally we lift the relation to *proper* configurations as follows: $\langle s, S \rangle \sim \langle \varphi, \mathcal{S} \rangle$ iff $s \bullet S \sim \varphi \bullet \mathcal{S}$.

In our setting a *bisimulation* is a relation R, between concrete- and abstract configurations, such that for every $(C, D) \in R$: if $C \longrightarrow C'$ then there is a configuration D' such that $D \longrightarrow D'$ and $(C', D') \in R$, and vice versa; where $C \longrightarrow C'$ is a transition between concrete configurations, given by the concrete semantics (augmented with the initialization of freeze variables), and $D \longrightarrow D'$ is a transition between abstract configurations, given by the symbolic semantics. Our main result states the equivalence between the operational and the symbolic semantics, based on this notion.

Theorem 10. *The above relation $\sim$ between proper configurations is a (strong) bisimulation.*

Proof. Suppose $\langle s, S \rangle \sim \langle \varphi, \mathcal{S} \rangle$. The proof proceeds by cases on top of the stacks, i.e., the program constructs; we only treat the case of procedure return. In this case $S = s' \bullet S'$ and $\mathcal{S} = \psi \bullet \mathcal{S}'$ for some s' and ψ, and there are s_r and φ_r such that

$$\langle s, s' \bullet S' \rangle \longrightarrow \langle s_r, S' \rangle \text{ and } \langle \varphi, \psi \bullet \mathcal{S}' \rangle \longrightarrow \langle \varphi_r, \mathcal{S}' \rangle$$

Note that we have $s' \sim \psi$, $s \sim \varphi$ and $S' \sim \mathcal{S}'$ by assumption. Further note that we can apply Lemma 9 since by assumption $\langle \varphi, \mathcal{S} \rangle$ is proper. We must show that $s_r \sim \varphi_r$ holds. Let $x, y \in V$. We distinguish three cases:

1. $x, y \in G$ (both global variables):

$$\begin{array}{rll} s_r(x) = s_r(y) & \text{iff } s(x) = s(y) & \text{Lemma 5.1} \\ & \text{iff } \varphi \models x = y & \text{assumption} \\ & \text{iff } \varphi_r \models x = y & \text{Lemma 9.1} \end{array}$$

2. $x, y \in L$ (both local variables):

$$\begin{array}{rll} s_r(x) = s_r(y) & \text{iff } s'(x) = s'(y) & \text{Lemma 5.2} \\ & \text{iff } \psi \models x = y & \text{assumption} \\ & \text{iff } \varphi_r \models x = y & \text{Lemma 9.2} \end{array}$$

3. $x \in G$ and $y \in L$. Recall that we have augmented the concrete semantics with the initialization of the freeze variables. It is easy to see that as a consequence $s(g') = s'(g)$ holds, which we will use below.

$$\begin{array}{rll} & s_r(x) = s_r(y) & \\ \text{iff} & s(x) = s'(y) & \text{Lemma 5.3} \\ \text{iff} & \exists g \in G. s(x) = s'(g) \text{ and } s'(g) = s'(y) & \text{properness} \\ \text{iff} & \exists g \in G. s(x) = s(g') \text{ and } s'(g) = s'(y) & \text{above argument} \\ \text{iff} & \exists g \in G. \varphi \models x = g' \text{ and } \psi \models g = y & \text{assumption} \\ \text{iff} & \varphi \models x = y & \text{Lemma 9.3} \end{array}$$

□

4 Adding pointers

In this section we extend our programming language with fields and corresponding updates for modeling linked object structures. In particular we investigate *k-bounded heaps*, in which the number of reachable objects is at most k. This notion is extended to k-bounded programs, in which during execution, the heap is always k-bounded. We show that every k-bounded program P including fields can be rewritten into a program P' without fields and equivalent to P.

The language of Section 2 is extended with statements $x := y.f$ and $x.f := y$, where f ranges over a finite set of (pointer) fields F. Note that the only type of field is that of a pointer to another object. Informally, the statement $x := y.f$ is a basic assignment, updating x to point to (the location referred to by) $y.f$. Field update $x.f := y$ changes to y the field of the object to which x refers via the field f. These two basic operations are sufficient; more general expressions and updates can be encoded. For example, a statement $x := y.f_{i_1}.f_{i_2}.f_{i_3} \ldots f_{i_k}$ is encoded as $x := y.f_{i_1}; x := x.f_{i_2}; x := x.f_{i_3}; \ldots; x := x.f_{i_k}$.

To give a semantics to this language we introduce a *heap* H as a pair $\langle s, h \rangle$ of a variable assignment $s : V \to \mathbb{N}$ such that $s(\mathsf{nil}) = 0$, and a field assignment $h : F \to (\mathbb{N} \to \mathbb{N})$ such that for all f, $h(f)(0) = 0$. We write $H(x)$ for $s(x)$, and $H(f)$ for $h(f)$.

For a set of variables X we denote with $\mathcal{R}_H(X) \subseteq \mathbb{N}$ the set of objects reachable from these variables in H, defined as the least fixpoint of the equation

$$\mathcal{R}_H(X) = \{H(x) \mid x \in X\} \cup \{H(f)(n) \mid f \in F, n \in \mathcal{R}_H(X)\}$$

We abbreviate $\mathcal{R}_H(V)$, where as before V denotes the set of program variables, by $\mathcal{R}_H$. We denote an assignment to a variable by $H[x := n]$, a global field update by $H[f := \rho]$ with $\rho : \mathbb{N} \to \mathbb{N}$ s.t. $\rho(0) = 0$, and local field update by $H[f := \rho[n := m]]$. We use the standard notation and definition of simultaneous assignments and updates.

A configuration is a tuple $\langle H, \Gamma \rangle$ where H is a heap and Γ is a stack of statements and heaps. We only give the transitions for dynamic allocation, and for assignments of the form $x.f := y$ and $x := y.f$. All other transitions are similar to the semantics of the language without fields, and are not repeated here. Again we assume a system variable cnt for the implementation of dynamic allocation. On a *dynamic allocation* all fields of the new object are set to point to 0.

$$\langle H, x := \mathsf{new} \bullet \Gamma \rangle \longrightarrow \langle H[x, \mathsf{cnt} := H(\mathsf{cnt}), H(\mathsf{cnt}) + 1][\bar{f} := \bar{\rho}], \Gamma \rangle$$

where $\bar{\rho}$ is a sequence such that $\rho_i = H(f_i)[H(x) := 0]$. A *field update* $x.f := y$ is as follows:

$$\frac{H(x) \neq H(\mathsf{nil})}{\langle H, x.f := y \bullet \Gamma \rangle \longrightarrow \langle H[f := H(f)[H(x) := H(y)], \Gamma \rangle}$$

Finally an assignment of the form $x := y.f$ is as follows:

$$\frac{H(y) \neq H(\mathsf{nil})}{\langle H, x := y.f \bullet \Gamma \rangle \longrightarrow \langle H[x := H(f)(H(y))], \Gamma \rangle}$$

It is not hard to see that reachability is undecidable for this language. For instance, we can simulate a 2-counter machine [9] by using three variables c_1, c_2, t and a single field f. An increment of c_i is then implemented as a statement $t := \mathsf{new}; t.f := c_i; c_i := t$, a decrement is a simple $c_i := c_i.f$ and we can test for zero by using a guard $[c_i = \mathsf{nil}]$. It is thus not possible to devise, in the same fashion as in Section 3, a precise abstraction of the semantics of our extended language for which reachability is decidable.

In [3] an abstraction of heaps in terms of isomorphic graphs is given, which is applied in a semantics with the property that reachability is decidable for programs in which every heap is k-bounded for some a priori fixed k.

Definition 11. *A heap H is k-bounded if $|\mathcal{R}_H| \leq k$. A computation $\langle H_1, \Gamma_1 \rangle \to \langle H_2, \Gamma_2 \rangle \to \ldots$ is k-bounded if $|\mathcal{R}_{H_i}|$ is k-bounded for all i. A program P with main procedure p_0 is k-bounded if every computation $\langle H_0, p_0 \rangle \to \ldots$ is k-bounded, for some initial heap H_0.*

We show that k-bounded programs with fields can be simulated by programs without fields in our basic language. We do so by means of a transformation from k-bounded programs to equivalent programs not containing any fields.

We first show how to represent k-bounded heaps using only plain variable assignments of the form $s : V \to \mathbb{N}$. To this end let k be a given bound. The correspondence between a k-bounded heap H and a state s is based on an explicit

enumeration of the objects in the visible heap, represented by k global variables $\bar{1}, \ldots, \bar{k}$ such that $s(\bar{i}) \neq s(\bar{j})$, for $i \neq j$. Notice that the undefined object 0 is already represented by the variable nil, and as such we have, in fact, a series $\mathsf{nil}, \bar{1}, \ldots, \bar{k}$ of $k+1$ variables to represent k objects, which will turn out to be of technical convenience. Further, we introduce for each $i = 1, \ldots, k$ and field $f \in F$ a global variable $\bar{i}_f$ which represents $H(f)(s(\bar{i}))$. Without loss of generality we assume that these variables do not appear in the given program. In the sequel we denote by I the set of global variables $\bar{1}, \ldots, \bar{k}$ and by $V(P)$ the (global and local) variables which do occur in P. We next define when a k-bounded heap H is represented by a variable assignment s, denoted by $H \equiv s$.

Definition 12. *Given a k-bounded heap and a variable assignment s we define $H \equiv s$ by*

1. *$s(\bar{i}) \neq s(\bar{j})$ and $s(\bar{i}) \neq 0$, for $i \neq j$, $i = 1, \ldots, k$ and $j = 1, \ldots, k$,*
2. *for every $n \in \mathcal{R}_H(V(P))$ s.t. $n \neq 0$ there is $i = 1, \ldots, k$ such that $s(\bar{i}) = n$,*
3. *$H(f)(s(\bar{i}))) = s(\bar{i}_f)$, for every $i = 1, \ldots, k$ and $f \in F$,*
4. *$H(\mathsf{cnt}) = s(\mathsf{cnt})$.*

The actual program transformation now is defined in terms of a function t which takes a k-bounded program P with fields and translates it into an equivalent program $t(P)$ without fields.

For each of the procedures except the initial one, we define $t(p_i :: B_i) = p_i :: t(B_i)$, where $t(B)$ will be defined by structural induction. We append in front of the initial procedure $p_0 :: B_0$ a series of allocations and assignments to initialize the representation of the heap:

$$\begin{aligned} p_0 :: \; & \Pi_{\bar{i} \in I}(\bar{i} := \mathsf{new}; \Pi_{f \in F} \bar{i}_f := \mathsf{nil};) \\ & \mathsf{true} := \mathsf{new}; \mathsf{false} := \mathsf{new}; \\ & t(B_0) \end{aligned}$$

where the *for all*-construct $\Pi_{l \in L} B$ is a shorthand for a sequential composition of the statements $B_{l'}$, for $l' \in L$, where $B_{l'}$ is obtained from B by substituting l' for l in B. For a better readability we omit the parenthesis in a for-all construct and assume its scope is clear from the context. The fresh global variables true and false, which are assumed not appear in the given program, will be used to encode boolean values.

We proceed to discuss for each of the statements in the language its translation. A simple assignment $x := y$ remains unchanged and the translation of a sequential composition or choice between two statements consists of the sequential composition or choice between the translated statements. Since the conditional statements refer only to plain variables, we simply have

$$t([x = y]B) = [x = y]t(B).$$

For an assignment of the form $x := y.f$, in order to find the variable representation of $y.f$, we must find the variable $\bar{i}$ which represents the object denoted by y.

The variable $\bar{i}_f$ then represents $y.f$. This search and corresponding assignment is described simply by the following non-deterministic choice:

$$t(x := y.f) = \Sigma_{\bar{i} \in I}[\bar{i} = y]x := \bar{i}_f.$$

Note that this "n-ary" non-deterministic choice generalizes binary choice in the usual manner. A field update $x.f := y$ is treated in a similar way:

$$t(x.f := y) = \Sigma_{\bar{i} \in I}[\bar{i} = x]\bar{i}_f := y.$$

Notice that both for field updates $x.f := y$ and $x := y.f$, the condition $H(x) \neq H(\mathsf{nil})$ of the semantic rules are enforced by only considering variables from I, which represent non-null objects.

To simulate a dynamic allocation $x := \mathsf{new}$, we non-deterministically select a variable $\bar{i} \in I$ which denotes an object that is *not* reachable from any variables in $V(P)$. This variable will be (re)used to represent the newly created object assigned to x. Reachability in a k-bounded heap can be implemented by the following statement using for each $\bar{i} \in I$ a fresh variable $\bar{i}_b$ to indicate that $\bar{i}$ is reachable from the variables in X:

$$R_X = \Pi_{\bar{i} \in I}\bar{i}_b := \mathsf{false};\ \Pi_{x \in X}\Sigma_{i=1}^{k}[\bar{i} = x]\bar{i}_b := \mathsf{true};\ B^k$$

where B denotes the statement

$$\Pi_{\bar{i} \in I}([\bar{i}_b = \mathsf{false}]\mathsf{skip} + [\bar{i}_b = \mathsf{true}]\Pi_{f \in F}\Sigma_{\bar{j} \in I}[\bar{i}_f = \bar{j}]\bar{j}_b := \mathsf{true})$$

and B^k denotes the sequential composition of k copies of B. Note that because the visible heap is k-bounded we need to iterate the statement B only k times. Further notice that since the undefined object is always reachable by the variable nil, and the heap is k-bounded, there can only be $k-1$ other reachable objects. So there will always be a representative $\bar{i} \in I$ of a non-null object which is not reachable, i.e., $\bar{i}_b = \mathsf{false}$ after executing the above reachability algorithm. This motivates the following translation of object creation:

$$t(x := \mathsf{new}) = R_{V(P)}; \Sigma_{\bar{i} \in I}([\bar{i}_b = \mathsf{false}]\bar{i} := \mathsf{new}; x := \bar{i}; \Pi_{f \in F}\bar{i}_f := \mathsf{nil})\,.$$

It is worthwhile to note that this translation is based on the *reuse* of a *variable* $\bar{i} \in I$ which in fact can be seen as a *canonical representative* of those variables which refer to the same object.

Finally we consider the case of a procedure call. It is not so difficult to see that upon the return of a procedure call in the *translated* program some objects which are reachable from the restored local variables may no longer be represented by a global variable $\bar{i} \in I$ because their representation may have been *reused* by the creation of new objects. In order to restore the representation of such objects we introduce for each $\bar{i} \in I$ fresh variables $\bar{i}' \in I'$ and $\bar{i}'_f$, for $f \in F$, which are used to store *before* the call a copy of the heap (as represented by the variables in I) by the following statement

$$copy = \Pi_{\bar{i}' \in I'}(\bar{i}' := \bar{i}; \Pi_{f \in F}\bar{i}'_f := \bar{i}_f)\,.$$

After the call we first compute which variables $\bar{i}' \in I'$ represent objects which are reachable from the (restored) local variables $L(P)$ of P in the "old" heap represented by the variables $\bar{i}' \in I'$ and $\bar{i}'_f$, for $f \in F$. Assuming for each variable $\bar{i}' \in I'$ an additional fresh variable $\bar{i}'_b$, this is computed by the statement $R'_{L(P)}$ which is obtained from $R_{L(P)}$, as defined above, by replacing simply the variables $\bar{i}$, $\bar{i}_f$ and $\bar{i}_b$ by $\bar{i}'$, $\bar{i}'_f$ and $\bar{i}'_b$, respectively, for $\bar{i} \in I$ and $f \in F$. Next we compute by $R_{G(P)}$ which variables $\bar{i} \in I$ do *not* represent objects reachable from the global variables $G(P)$ of P in the current heap. The statement

$$R_{\bar{i}'} = b := \mathsf{true}; \Pi_{\bar{j} \in I}([\bar{j} \neq \bar{i}']skip + [\bar{j} = \bar{i}']b := \mathsf{false})$$

checks whether the object denoted by $\bar{i}'$ is already represented by some variable $\bar{i} \in I$. If the object denoted by $\bar{i}'$ is not yet represented the following statement

$$restore = \Sigma_{\bar{j} \in I}([\bar{j}_b = \mathsf{false}]\bar{j} := \bar{i}'; \Pi_{f \in F}\bar{j}_f := \bar{i}'_f; \bar{j}_b := \mathsf{true})$$

restores the representation of $\bar{i}'$. Putting the above statements together

$$\begin{aligned} return = {} & R'_{L(P)}; R_{G(P)}; \\ & \Pi_{\bar{i}' \in I'} R_{\bar{i}'}; ([b = \mathsf{false}]skip + [b = \mathsf{true}]restore) \end{aligned}$$

restores upon return the representation of the old local heap by the variables I.

Summarizing, we have the following translation of procedure calls:

$$t(\mathsf{call}\, p) = copy; \mathsf{call}\, p; return.$$

In order to state the correctness of the translation in terms of a bisimulation relation we first extend pointwise the (representation) relation $H \equiv s$ to the corresponding stacks:

1. if $H \equiv s$ and $\Gamma \equiv S$ then $H \bullet \Gamma \equiv s \bullet S$,
2. if $\Gamma \equiv S$ then $B \bullet \Gamma \equiv t(B) \bullet S$.

Let $\Longrightarrow$ denote the concrete semantics where the translations of assignments, procedure calls and returns are executed *atomically* (i.e., in one step).

Theorem 13. *Given a program P with fields, let $H \bullet \Gamma \equiv s \bullet S$. We have*

1. *if $\langle H, \Gamma \rangle \longrightarrow \langle H', \Gamma' \rangle$ then $\langle s, S \rangle \Longrightarrow \langle s', S' \rangle$, for some $\langle s', S' \rangle$ such that $H' \bullet \Gamma' \equiv s' \bullet S'$, and*
2. *if $\langle s, S \rangle \Longrightarrow \langle s', S' \rangle$ then $\langle H, \Gamma \rangle \longrightarrow \langle H', \Gamma' \rangle$, for some $\langle H', \Gamma' \rangle$ such that $H' \bullet \Gamma' \equiv s' \bullet S'$.*

5 Conclusion

The interplay between unbounded allocation of objects and recursion with local variables gives rise to an infinite state space. By representing the state space symbolically, we have shown that reachability, as well as model checking, is

decidable. Further, we have shown that adding pointer fields to our core language with the restriction that the number of reachable objects is bounded, does not increase the expressiveness of the language.

Our symbolic semantics greatly simplifies the basic mechanism of recursion with local variables in the presence of dynamic object allocation, as is also exemplified by a neat formalization of dynamic deallocation, a feature that is typically problematic in the context of model checking. Consider for example a *deallocation* statement "$\mathsf{del}\, x$" that sets x and all of its aliases to nil. Note that this is different from an assignment $x := \mathsf{nil}$, as the latter statement does not affect any alias of x. Symbolically, the effect of a deallocation statement is formalized in terms of its strongest postcondition $SP(\mathsf{del}\, x, \varphi)$ simply as $\varphi \wedge x = \mathsf{nil}$. An equivalent concrete semantics for deallocation is much more complex because the stack may contain variables still referencing deallocated objects and a program has only access to the top of the stack. One way of implementing a concrete semantics of deallocation is by an explicit recording of the deleted objects.

Finally, the symbolic nature of our semantics provides a promising basis for future model checking tool development using, for example, the Maude implementation of rewriting logic.

References

1. P. Abdulla, M. Atig, G. Delzanno, and A. Podelsk Push-Down Automata with Gap-Order Constraints. In *Proc. FSEN 2013*, this volume.
2. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. CONCUR 97*, volume 1243 of LNCS, pp. 135–150, Springer, 1997.
3. A. Bouajjani, S. Fratani, S. Qadeer. Context-Bounded Analysis of Multithreaded Programs with Dynamic Linked Structures. In *Proc. of Computer Aided Verification (CAV 2007)* vol. 4590 of LNCS, Springer 2007.
4. 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 *Proc. of Int. Conf. on Software Engineering*, pp. 439-448. IEEE, 2000.
5. C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, 29(7):577–603, 1999.
6. D. Distefano, J.-P. Katoen, A. Rensink. Who is Pointing When to Whom? In *Proc. of Int. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004)* volume 3328 of LNCS, pp. 250–262, Springer 2004.
7. J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Proc. of Computer Aided Verification (CAV 2001)*, volume 2102 of LNCS, pp. 324–336, Springer, 2001.
8. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Int. Jour. on Softw. Tools for Technology Transfer*, 2(4):366-381, 2000.
9. M.L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.
10. D. Park, U. Stern, J. Skakkebaek, and D. Dill. Java Model Checking. In *Proc. of the 15th Int. Conf. on Automated Software Engineering*, pp. 253–256. IEEE, 2000.
11. J. Rot, I.M. Asavoae, F. de Boer, M.M. Bonsangue, and D. Lucanu. Interacting via the Heap in the Presence of Recursion In *Proc. of the 5th Interaction and Concurrency Experience (ICE 2012)*, volume 104 of EPTCS, pp. 99–113, 2012.