Monotonic Abstraction for Programs with Dynamic Memory Heaps

Parosh Aziz Abdulla[1], Ahmed Bouajjani[2], Jonathan Cederberg[1], Frédéric Haziza[1], and Ahmed Rezine[1,2]

[1] Uppsala University, Sweden
[2] LIAFA, University of Paris 7, France

Abstract. We propose a new approach for automatic verification of programs with dynamic heap manipulation. The method is based on symbolic (backward) reachability analysis using upward-closed sets of heaps w.r.t. an appropriate preorder on graphs. These sets are represented by a finite set of minimal graph patterns corresponding to a set of bad configurations. We define an abstract semantics for the programs which is monotonic w.r.t. the preorder. Moreover, we prove that our analysis always terminates by showing that the preorder is a well-quasi ordering. Our results are presented for the case of programs with 1-next selector. We provide experimental results showing the effectiveness of our approach.

1 Introduction

Software verification needs the use of efficient algorithmic techniques for the analysis of infinite-state models. The sources of infiniteness are multiple and can be related to complex control such as (potentially recursive) procedure calls and dynamic creation of processes, or to the manipulation of (unbounded-size) dynamic data-structures and variables ranging over infinite data domains. A lot of work has been devoted in the last decade to the design of automatic verification techniques for infinite-state models, and several general approaches and formal frameworks have emerged allowing either to establish decidability results and derive verification algorithms (e.g., [20,2]), or to define generic exact/abstract analysis procedures (e.g., [7,11,22,30]).

One of the widely adopted frameworks in this context of infinite-state verification is based on the concept of *monotonic systems w.r.t. a well-quasi ordering* [2,20]. This framework provides a scheme for proving the termination of the (backward) reachability analysis, and it has been used for the design of verification algorithms for various models including Petri nets, lossy channel systems, timed Petri nets, broadcast protocols, etc. (see, e.g., [5,6,18,19]). The idea is, given a class of models, to define a preorder $\preceq$ on the configuration space such that (1) $\preceq$ is a simulation relation on the considered models, and (2) $\preceq$ is a well-quasi ordering (WQO for short). If such a preorder can be defined, then it can be proved that the reachability problem of an upward-closed set of configurations (w.r.t. $\preceq$) is decidable. Indeed, (1) monotonicity implies that for any

A. Gupta and S. Malik (Eds.): CAV 2008, LNCS 5123, pp. 341–354, 2008.

upward-closed set, the set of its predecessors is an upward-closed set, and (2) the fact that $\preceq$ is a WQO implies that every upward-closed set can be characterized by its *finite* set of minimal elements. Therefore, starting from an upward-closed set of configurations U, the iterative computation of the backward reachable configurations from U necessarily terminates since only a finite number of steps are needed to capture all minimal elements of the set of predecessors of U. Obviously, this requires that upward-closed sets can be effectively represented and manipulated (i.e., there are procedures for, e.g., computing immediate predecessors and unions, and for checking entailment). This general scheme can be applied for the verification of safety properties since this problem can be reduced to checking the reachability of a set of bad configurations which is typically an upward-closed set w.r.t. the considered preorder. (For instance, mutual exclusion is violated as soon as there are (at least) two processes in the critical section.)

Unfortunately, many systems do not fit into this framework, in the sense that there is no nontrivial (useful) WQO for which these systems are monotonic. Nevertheless, a natural approach to overcome this problem is, given a preorder $\preceq$, to define an abstract semantics of the considered systems which forces their monotonicity. Basically, the idea is to consider that a transition is possible from a configuration c_1 to c_2 if it is possible from any smaller configuration $c_1' \preceq c_1$ to c_2. This simple idea has been used recently in works concerning the verification of parametrized networks of (finite/infinite-state) processes, and surprisingly, it leads to quite efficient abstract analysis techniques which allow to handle *fully automatically* several non-trivial examples of such systems [3,4]. This encourages us to investigate its application to other classes of complex systems.

In this paper, our aim is to develop a framework based on the approach introduced above for the verification of sequential iterative programs manipulating dynamic memory heaps. The issue of verifying automatically such programs has received a lot of attention in the last few years, and many approaches and techniques have been developed including static-analysis and abstraction-based frameworks (see, e.g., [29]), logic-based frameworks(see, e.g., [25,28]), automata-based frameworks (see, e.g., [14,21]), etc. Here, we introduce a framework based on symbolic (backward) reachability analysis using upward-closed sets of heap graphs (w.r.t. some appropriate preorder). As a first step toward this framework, we present in this paper the results of our investigations concerning the case of programs manipulating heap structures with *one* next-selector, i.e., heaps of programs manipulating lists with possibility of sharing and circularity.

More precisely, we consider that heaps are represented as labeled graphs, where labels correspond to positions of program variables. We propose a preorder $\preceq$ between heap graphs which corresponds basically to the following: Given two graphs g_1 and g_2, we have $g_1 \preceq g_2$ if g_1 can be obtained from g_2 by a sequence of transformations consisting of either deleting an edge, a variable, or an isolated vertex, or of contracting segments (i.e., sequence of vertices) without sharing in the graph.

Actually, our graph representations correspond in general to sets of heaps instead of a single one. They can be seen as minimal patterns (w.r.t. $\preceq$), and they

represent all the heaps that subsume (w.r.t. $\preceq$) these patterns. Therefore, our graph representations define upward-closed sets of heap graphs.

Then, we provide procedures for computing sets of predecessors w.r.t. the abstract semantics we consider (introduced above), and for checking entailment. These procedures allow to define a *simple algorithm* which computes an over-approximation of the set of backward reachable configurations starting from an upward-closed set of heap graphs (effectively given as a finite set of minimal elements). We show that this algorithm *always terminates* by proving that the preorder we have defined on heap graphs is a WQO.

Our analysis allows to check properties such as absence of null dereferencing as well as absence of garbage creation. Moreover, it allows to check shape (well-formedness) properties of the heaps (for instance the fact that the output is always a list without sharing). We show indeed that these kinds of verification problems can be reduced to the problem of reaching sets of bad configurations corresponding to the existence in the heap graph of some *minimal bad patterns*. We also provide experimental results showing the effectiveness of our approach.

Related work. As mentioned before, several approaches to the automatic analysis of programs with dynamic linked data structures have been proposed (see, e.g., [14,17,21,29]). Shape analysis as introduced in [29] is based on the computation of abstract shape graphs using the so-called instrumentation predicates. An automata-based approach using abstract regular model checking (ARMC) [15] has been proposed in [13,14]. In [10,17], an automatized analysis approach based on separation logic combined with abstraction techniques (close to widening techniques) has been proposed. With respect to these approaches, the one we present in this paper is conceptually and technically different and simpler. In particular, the ARMC-based approach needs the manipulation of quite complex encodings of the heap graphs into words or trees (in order to represent sets of heap encodings using finite-state automata), and use a sophisticated machinery for manipulating these encodings based on representing program statements as (word/tree) transducers. In contrast, the approach presented here uses a natural representation of heaps as graphs and employs direct procedures for computing operations on these graphs. This direct approach has already shown its advantages w.r.t. the approach using transducers in the context of regular model checking for parametrized networks of processes [3]. Also, the approach we present uses a built-in abstraction principle which is different from the ones used in the existing approaches, and which makes the analysis fully automatic.

The existing approaches mentioned above (shape analysis, abstract regular model checking, separation logic) can handle some classes of general heap structures (including doubly linked lists, lists of lists, trees, etc.). Although the techniques presented in this paper concern the case of heap structures with 1-next selector, our approach (based on upward-closed abstractions w.r.t. preorders on graphs) can in principle be extended to more general classes of heaps.

Concerning the particular class of programs manipulating heaps with 1-next selector, there are many other verification approaches which have been developed recently (see, e.g., [8,12,13,16,23,24]). Almost all these works use the fact that in

this case (1) the heap graphs are collections of reversed trees potentially having their roots connected to a loop, and moreover (2) the number of leaves and shared vertices in these graphs is bounded linearly in terms of the number of program variables. For instance, in [24], these properties are used to define an abstraction which consists of contracting all segments without sharing. In our case, we use these properties in order to prove that the preorder we propose on graph representations is a WQO. However, our abstraction is different from the one proposed for instance in [24] since we can have graphs which are not minimal w.r.t. to contraction (e.g., we can express the fact that the length of a segment is at least some given natural number), and we can also have graphs corresponding to a partial description of the heap where only a *part* of the reachable heap from *some* of the program variables is constrained.

In [9,12], translations from programs with lists to counter automata have been defined based on the representation of heap graph as its contracted version supplied with the information about the length of each contracted sharing-free segments. These translations allow to use various existing techniques for the analysis of counter systems in order to check safety properties involving constraints relating the lengths of different lists, or to check termination. Such analysis involving quantitative reasoning cannot be done with the techniques presented in this paper. As said above, these techniques can handle some reasoning about the sizes of the lists, but only concerning constraints on minimal lengths. However, extensions of our techniques to more general constraints (e.g., gap-order constraints [27]) are possible.

Outline. In the next section, we introduce the class of programs we consider together with their graph representations. In Section 3 we describe a set of graph operations which we use in the subsequent sections. Section 4 introduces the ordering on configurations. In Section 5, we introduce a relation which we use as the basic step in the reachability algorithm. Section 6 introduces the backward reachability algorithm, and proves its partial correctness. The termination of the algorithm is shown in Section 7. Section 8 reports the results of applying a prototype, based on the method, to a number of simple programs. Finally, in Section 9 we give some conclusions.

2 Preliminaries

We consider programs that operate on data structures with one next-pointer such as traditional singly-linked lists and circular lists (possibly sharing their parts). We represent the store as a graph, where the vertices represent the list cells, and the successor of a vertex represents the cell pointed to by the current one. The graphs are of a special form in the sense that each vertex has at most one successor. A program also uses a finite set of pointers which we call *variables*. A cell is labeled by the (possibly empty) set of variables pointing to it.

For simplicity of presentation, we will treat the constant *null* as a variable, with the special property that whenever a vertex is labeled by *null*, the successor of the cell is undefined.

For a partial function f, we write $f(a) = \perp$ to denote that $f(a)$ is undefined. For a (partial) function f , we use $f\,[a \leftarrow b]$ to denote the function f' such that $f'(a) = b$ and $f'(x) = f(x)$ if $x \neq a$.

Formally, we assume a finite set X of variables including the element *null*. A *program* P is a pair (Q, T) where Q is a finite set of *control states* and T is a finite set of *transitions*. A transition is a triple (q_1, a, q_2) where $q_1, q_2 \in Q$ are control states and a is an *action*. An *action* is of one of the following forms $x = y$, $x \neq y$, $x := y$ where $x \neq null$, $x.next = y$ where $x \neq null$, or $x := y.next$ where $x, y \neq null$. The transition corresponds to the program changing control state from q_1 to q_2 while performing the operation described in a on the data structure. We choose to work with the above minimal set of operations. Other operations, e.g., $x = y.next$, $x \neq y.next$, etc, can be expressed using the given set.

A *graph* g is a triple $(V, succ, \lambda)$ where V is a finite set of *vertices*, $succ$ is a partial function from V to V, and λ is a partial function from X to V. Furthermore, it is always the case that $succ(\lambda(null)) = \perp$. Intuitively, the vertices correspond to the list cells. The function $succ$ defines the successors of the cells. If $succ(v) = \perp$, the cell represented by v has currently no successor. The function λ defines the cell to which a given variable points. If $\lambda(x) = \perp$, the value of variable (pointer) x is undefined.

A *configuration* c is a pair (q, g) where $q \in Q$ is a control state and g is a graph. We define a transition relation on configurations as follows. Let $t = (q_1, a, q_2)$ be a transition and let $c = (q, g)$ and $c' = (q', g')$ be configurations. We write $c \xrightarrow{t} c'$ to denote that $q = q_1$, $q' = q_2$, and $g \xrightarrow{a} g'$, where $g \xrightarrow{a} g'$ holds if one of the following conditions is satisfied:

- a is of the form $x = y$, $\lambda(x) \neq \perp$, $\lambda(y) \neq \perp$, $\lambda(x) = \lambda(y)$, and $g' = g$.
- a is of the form $x \neq y$, $\lambda(x) \neq \perp$, $\lambda(y) \neq \perp$, $\lambda(x) \neq \lambda(y)$, and $g' = g$.
- a is of the form $x := y$, $\lambda(y) \neq \perp$, $succ' = succ$, and $\lambda' = \lambda\,[x \leftarrow \lambda(y)]$.
- a is of the form $x := y.next$, $\lambda(y) \neq \perp$, $succ(\lambda(y)) \neq \perp$, $succ' = succ$, and $\lambda' = \lambda\,[x \leftarrow succ(\lambda(y))]$.
- a is of the form $x.next := y$, $\lambda(x) \neq \perp$, $\lambda(y) \neq \perp$, $\lambda(x) \neq \lambda(null)$, $\lambda' = \lambda$, and $succ' = succ\,[\lambda(x) \leftarrow \lambda(y)]$.

We define $\longrightarrow$ as $\bigcup_{t \in T} \xrightarrow{t}$ and use $\xrightarrow{*}$ to denote the reflexive transitive closure of $\longrightarrow$. For sets C_1 and C_2 of configurations, we use $C_1 \longrightarrow C_2$ to denote that $c_1 \longrightarrow c_2$ for some $c_1 \in C_1$ and $c_2 \in C_2$. By $c_1 \longrightarrow C_2$ we mean $\{c_1\} \longrightarrow C_2$. We define $c_1 \xrightarrow{*} C_2$, $C_1 \xrightarrow{*} C_2$, etc in a similar manner to above.

3 Operations on Graphs

In this section, we define a number of operations on graphs which we use in the subsequent sections. In the rest of the section, we assume a graph $g = (V, succ, \lambda)$.

For $v_1, v_2 \in V$, we use $(g.succ)\,[v_1 \leftarrow v_2]$ to denote the graph $g' = (V', succ', \lambda')$ where $V' = V$, $\lambda' = \lambda$, and $succ' = succ\,[v_1 \leftarrow v_2]$. Intuitively, we only modify g so that v_2 becomes the successor of v_1. We define $(g.\lambda)\,[x \leftarrow v]$ analogously. That is, we make x point to v.

For a vertex $v \in V$, we say that v is *simple* if $|succ^{-1}(v)| = 1$, $succ(v) \neq \bot$, and there is no $x \in X$ with $\lambda(x) = v$. In other words, v has exactly one predecessor, one successor and no label. We say that v is *isolated* in g if $succ(v) = \bot$, $succ^{-1}(v) = \emptyset$, and there is no $x \in X$ with $\lambda(x) = v$. In other words, v has no successors or predecessors and it is not labeled by any variable.

Operations on vertices. For $v \notin V$, we use $g \oplus v$ to denote the graph $g' = (V', succ', \lambda')$ such that $V' = V \cup \{v\}$, $succ' = succ$, and $\lambda' = \lambda$, i.e. we add a new vertex to g. Observe that the added vertex is then isolated.

For an *isolated* vertex $v \in V$, we use $g \ominus v$ to denote the graph $g' = (V', succ', \lambda')$ such that $V' = V - \{v\}$, $succ' = succ$, and $\lambda' = \lambda$.

Operations on variables. We define $g \oplus x$ to be the set of graphs we get from g by letting x point anywhere inside g. Formally, we define $g \oplus x$ to be the smallest set containing each graph g' such that one of the following conditions is satisfied: (i) there is a $v \notin V$ such that $g' = ((g \oplus v).\lambda)\,[x \leftarrow v]$, *i.e.* we add a vertex to g and make x point to it. (ii) there is a $v \in V$ such that $g' = (g.\lambda)\,[x \leftarrow v]$, *i.e.* we make x point to some vertex in g. (iii) there are $v_1 \in V$, $v_2 \notin V$, and graphs $g_i = (V_i, succ_i, \lambda_i)$ for $i = 1, 2, 3$, such that $succ(v_1) \neq \bot$, $g_1 = g \oplus v_2$, $g_2 = (g_1.succ)\,[v_2 \leftarrow succ_1(v_1)]$, $g_3 = (g_2.succ)\,[v_1 \leftarrow v_2]$, and $g' = (g_3.\lambda)\,[x \leftarrow v_2]$, *i.e.* we add a new vertex v_2 in between v_1 and its successor and make x point to v_2.

For variables x and y with $\lambda(x) \neq \bot$, we define $g \oplus_{=x} y$ to be the graph $g' = (g.\lambda)\,[y \leftarrow \lambda(x)]$, *i.e.* we make y point to the same vertex as x. Furthermore, we define $g \oplus_{\neq x} y$ to be the smallest set containing each graph g' such that $g' \in (g \oplus y)$ and $\lambda'(y) \neq \lambda'(x)$, *i.e.* we make y point anywhere inside g except to the vertex pointed to by x.

For variables x and y with $\lambda(x) \neq \bot$ and $succ(\lambda(x)) \neq \bot$, we use $g \oplus_{x\rightarrow} y$ to denote the graph $(g.\lambda)\,[y \leftarrow succ(\lambda(x))]$, *i.e.* we make y point to the successor of the vertex pointed to by x. For variables x and y with $\lambda(x) \neq \bot$, we define $g \oplus_{x\leftarrow} y$ to be the set of graphs we get from g by letting y point to any vertex where the successor is either undefined or pointed to by x. Formally, we define $g \oplus_{x\leftarrow} y$ to be the smallest set containing each graph g' such that one of the following conditions is satisfied: (i) there is a $v \notin V$ such that $g' = ((g \oplus v).\lambda)\,[y \leftarrow v]$. (ii) there is a $v \in V$ such that $v \neq \lambda(null)$, either $succ(v) = \bot$ or $succ(v) = \lambda(x)$, and $g' = (g.\lambda)\,[y \leftarrow v]$. That is, we place y on the vertices without a successor or the ones whose successor is pointed to by x. (iii) there are $v_1 \in V$, $v_2 \notin V$, and graphs $g_i = (V_i, succ_i, \lambda_i)$ for $i = 1, 2, 3$, such that $succ(v_1) = \lambda(x)$, $g_1 = g \oplus v_2$, $g_2 = (g_1.succ)\,[v_2 \leftarrow \lambda(x)]$, $g_3 = (g_2.succ)\,[v_1 \leftarrow v_2]$, and $g' = (g_3.\lambda_3)\,[y \leftarrow v_2]$. Intuitively, we add a new vertex v_2 in between the vertex pointed by x and its predecessors and make y point to v_2.

For a variable x, we use $g \ominus x$ to denote $(g.\lambda)\,[x \leftarrow \bot]$.

Operations on edges. If $\lambda(x) \neq \bot$, $\lambda(y) \neq \bot$ and $\lambda(x) \neq \lambda(null)$, we use $g \boxplus (x \rightarrow y)$ to denote $(g.succ)\,[\lambda(x) \leftarrow \lambda(y)]$, *i.e.* we delete the edge between the vertex $\lambda(x)$ and its successor (if any) and add an edge from $\lambda(x)$ to $\lambda(y)$.

If $\lambda(x) \neq \bot$ and $\lambda(x) \neq \lambda(null)$, we define $g \boxplus (x \rightarrow)$ to be the set of graphs we get from g by letting $x.next$ point anywhere inside g. Formally, we define $g \boxplus (x \rightarrow)$ to be the smallest set containing each graph g' such that one of the following conditions is satisfied: (i) there is a $v \notin V$ such that $g' = ((g \oplus v).succ)\,[\lambda(x) \leftarrow v]$. (ii) there is a $v \in V$ such that $g' = (g.succ)\,[\lambda(x) \leftarrow v]$. (iii) there are $v_1 \in V$, $v_2 \notin V$, and graphs $g_i = (V_i, succ_i, \lambda_i)$ for $i = 1, 2, 3$, such that $succ(v_1) \neq \bot$, $g_1 = g \oplus v_2$, $g_2 = (g_1.succ)\,[v_2 \leftarrow succ_1(v_1)]$, $g_3 = (g_2.succ)\,[v_1 \leftarrow v_2]$, and $g' = (g_3.succ)\,[\lambda_3(x) \leftarrow v_2]$.

If $\lambda(x) \neq \bot$, we denote $g \boxminus (x \rightarrow)$ as $(g.succ)\,[\lambda(x) \leftarrow \bot]$, *i.e.* we remove the edge from the vertex pointed to by x and its successor (if any).

4 Ordering

In this section, we introduce an ordering on configurations. Based on the ordering, we will define the coverability problem which we use to check safety properties, and define the abstract transition relation. The latter is an over-approximation of the concrete transition relation.

Ordering. Let $g = (V, succ, \lambda)$ and $g' = (V', succ', \lambda')$. We write $g \lhd g'$ to denote that one of the following properties is satisfied: (i) *Variable Deletion*: $g = g' \ominus x$ for some variable x, (ii) *Vertex Deletion*: $g = g' \ominus v$ for some isolated vertex $v \in V'$, (iii) *Edge Deletion*: $g = (g'.succ)\,[v \leftarrow \bot]$ for some $v \in V'$, or (iv) *Contraction*: there are vertices $v_1, v_2, v_3 \in V'$ and graphs g_1, g_2 such that v_2 is simple, $succ'(v_1) = v_2$, $succ'(v_2) = v_3$, $g_1 = (g'.succ)\,[v_2 \leftarrow \bot]$, $g_2 = (g_1.succ)\,[v_1 \leftarrow v_3]$ and $g = g_2 \ominus v_2$.

We write $g \preceq g'$ to denote that there are $g_0 \lhd g_1 \lhd g_2 \lhd \cdots \lhd g_n$ with $n \geq 0$, $g_0 = g$, and $g_n = g'$. That is, we can obtain g from g' by performing a finite sequence of variable deletion, vertex deletion, edge deletion and contraction operations. For configurations $c = (q, g)$ and $c' = (q', g')$, we write $c \preceq c'$ to denote that $q' = q$ and $g \preceq g'$.

For a configuration c, we use $c{\uparrow}$ to denote the *upward closure* of c, *i.e.* $c{\uparrow} = \{c' \mid c \preceq c'\}$. We use $c{\downarrow}$ to denote the *downward closure* of c, *i.e.* $c{\downarrow} = \{c' \mid c' \preceq c\}$. For a set C of configurations, we define $C{\uparrow}$ as $\bigcup_{c \in C} c{\uparrow}$. We define $C{\downarrow}$ analogously.

Safety Properties. In order to analyze safety properties, we study the *coverability problem* defined below.

Coverability
Instance:
Sets C_{Init} and C_F of configurations.
Question: Is it the case $C_{Init} \xrightarrow{*} C_F{\uparrow}$?

Intuitively, $C_F{\uparrow}$ represents a set of "bad" states which we do not want to reach during the execution of the program. This set is represented by a set C_F of minimal elements.

In Section 8, we describe how to encode properties such as garbage generation, dereferencing and shape violation as reachability of upward closed sets of configurations represented by finite sets of minimal elements. Therefore, checking safety with respect to these properties amounts to solving the coverability problem.

Abstract Transition Relation. We write $c_1 \xrightarrow{t}_A c_2$ to denote that there is a c_3 such that $c_3 \preceq c_1$ and $c_3 \xrightarrow{t} c_2$. In other words, a step of the abstract transition relation consists of first moving to a smaller configuration (wrt $\preceq$) and then performing a step of the concrete transition relation. Notice that the abstraction corresponds to an over-approximation and therefore any safety property which holds in the abstract system will also hold in the concrete one.

5 Computing Predecessors

The main idea behind our algorithm to solve the coverability problem, is to perform backward reachability analysis. The basic step of the algorithm uses a relation $\rightsquigarrow$ defined on the set of configurations. Intuitively, $c \rightsquigarrow c'$ means that, from c', we can perform a transition and reach a configuration in the upward closure of c. First, we give the formal definition of $\rightsquigarrow$, and then describe some of its properties, and in particular how it relates to the transition relation $\longrightarrow$.

For a graph $g = (V, succ, \lambda)$, a graph g', and an action a, we write $g \overset{a}{\rightsquigarrow} g'$ to denote that one of the following conditions is satisfied:

1. a is of the form $x = y$ and one of the following conditions is satisfied:
   (a) $\lambda(x) \neq \bot$, $\lambda(y) \neq \bot$, $\lambda(x) = \lambda(y)$ and $g' = g$.
   (b) $\lambda(x) \neq \bot$, $\lambda(y) = \bot$ and $g' = g \oplus_{=x} y$.
   (c) $\lambda(x) = \bot$, $\lambda(y) \neq \bot$ and $g' = g \oplus_{=y} x$.
   (d) $\lambda(x) = \bot$, $\lambda(y) = \bot$ and $g' = g_1 \oplus_{=x} y$ for some $g_1 \in (g \oplus x)$.
   In order to be able to perform the action, the variables x and y should point to the same vertex. If one (or both) of them are missing, then we add them to the graph (with the restriction that they point to the same vertex).
2. a is of the form $x \neq y$ and one of the following conditions is satisfied:
   (a) $\lambda(x) \neq \bot$, $\lambda(y) \neq \bot$, $\lambda(x) \neq \lambda(y)$ and $g' = g$.
   (b) $\lambda(x) \neq \bot$, $\lambda(y) = \bot$ and $g' \in g \oplus_{\neq x} y$.
   (c) $\lambda(x) = \bot$, $\lambda(y) \neq \bot$ and $g' \in g \oplus_{\neq y} x$.
   (d) $\lambda(x) = \bot$, $\lambda(y) = \bot$ and $g' \in g_1 \oplus_{\neq x} y$ for some $g_1 \in (g \oplus x)$.
   We proceed as in case 1, but now under the restriction that x and y point to different vertices (rather than to the same vertex).
3. a is of the form $x := y$ and one of the following conditions is satisfied:
   (a) $\lambda(x) \neq \bot$, $\lambda(y) \neq \bot$, $\lambda(x) = \lambda(y)$ and $g' = g \ominus x$.
   (b) $\lambda(x) \neq \bot$, $\lambda(y) = \bot$ and $g' = g_1 \ominus x$ where $g_1 = g \oplus_{=x} y$.
   (c) $\lambda(x) = \bot$, $\lambda(y) \neq \bot$ and $g' = g$.
   (d) $\lambda(x) = \bot$, $\lambda(y) = \bot$ and $g' \in (g \oplus y)$.
   In difference to case 1 is that the variable x may have had any value before performing the assignment. Therefore, we remove x from the graph.
4. a is of the form $x := y.next$ and one of the following conditions is satisfied:
   (a) $\lambda(x) \neq \bot$, $\lambda(y) \neq \bot$, $succ(\lambda(y)) \neq \bot$, $succ(\lambda(y)) = \lambda(x)$ and $g' = g \ominus x$.
   (b) $\lambda(x) \neq \bot$, $\lambda(y) \neq \bot$, $\lambda(y) \neq \lambda(null)$, $succ(\lambda(y)) = \bot$ and $g' = g_1 \ominus x$, where $g_1 = g \boxplus (y \to x)$.
   (c) $\lambda(x) \neq \bot$, $\lambda(y) = \bot$ and there are graphs g_1, g_2 such that $g' = g_2 \ominus x$, $g_2 = g_1 \boxplus (y \to x)$ and $g_1 \in g \oplus_{x\leftarrow} y$.

(d) $\lambda(x) = \bot$, $\lambda(y) \neq \bot$, $succ(\lambda(y)) \neq \bot$ and $g' = g$.
(e) $\lambda(x) = \bot$, $\lambda(y) \neq \bot$, $\lambda(y) \neq \lambda(null)$, $succ(\lambda(y)) = \bot$ and $g' \in g \boxplus (y \rightarrow)$.
(f) $\lambda(x) = \bot$, $\lambda(y) = \bot$ and there are graphs g_1, g_2, g_3 such that $g_1 \in g \oplus x$, $g_2 \in g_1 \oplus_{x\leftarrow} y$, $g_3 = g_2 \boxplus (y \rightarrow x)$ and $g' = g_3 \ominus x$.

Similarly to case 3 we remove x from the graph. The successor of y should be defined and point to the same vertex as x. In case the successor is missing, we add an edge explicitly from the vertex labeled by y to the vertex labeled by x. Furthermore, if x is missing then the successor of y may point anywhere inside the graph.

5. a is of the form $x.next := y$ and one of the following conditions is satisfied:
(a) $\lambda(x) \neq \bot$, $succ(\lambda(x)) \neq \bot$, $\lambda(y) \neq \bot$, $succ(\lambda(x)) = \lambda(y)$ and $g' = g \boxminus (x \rightarrow)$.
(b) $\lambda(x) \neq \bot$, $succ(\lambda(x)) \neq \bot$, $\lambda(y) = \bot$ and $g' = g_1 \boxminus (x \rightarrow)$, where $g_1 = g \oplus_{x\rightarrow} y$.
(c) $\lambda(x) \neq \bot$, $succ(\lambda(x)) = \bot$, $\lambda(y) \neq \bot$, $\lambda(x) \neq \lambda(null)$ and $g' = g$.
(d) $\lambda(x) \neq \bot$, $succ(\lambda(x)) = \bot$, $\lambda(y) = \bot$, $\lambda(x) \neq \lambda(null)$ and $g' \in g \oplus y$.
(e) $\lambda(x) = \bot$, $\lambda(y) \neq \bot$ and $g' = g_1 \boxminus (x \rightarrow)$, where $g_1 \in g \oplus_{y\leftarrow} x$.
(f) $\lambda(x) = \bot$, $\lambda(y) = \bot$ and there are graphs g_1, g_2 such that $g_1 \in g \oplus y$, $g_2 \in g_1 \oplus_{y\leftarrow} x$ and $g' = g_2 \boxminus (x \rightarrow)$.

After performing the action, the successor of the vertex labeled by x should be the same vertex as the one labeled by y. Before performing the action, the successor could have been anywhere inside the graph, and the corresponding edge is therefore removed.

Remark. In the above definition, we assume that x and y are different variables. It is straightforward to handle the case where they are the same variable.

For a transition $t = (q_1, a, q_2)$ and configurations $c = (q, g)$ and $c' = (q', g')$, we write $c \overset{t}{\rightsquigarrow} c'$ to denote that $q = q_1$, $q' = q_2$ and $g \overset{a}{\rightsquigarrow} g'$. We use $c \rightsquigarrow c'$ to denote that $c \overset{t}{\rightsquigarrow} c'$ for some $t \in T$. For a set C of configurations and a configuration c, we define $Rank(C)(c)$ to be the smallest n such that there is a sequence $c_0 \rightsquigarrow c_1 \rightsquigarrow \cdots \rightsquigarrow c_n$ where $c_0 = c$ and there is a $c' \in C$ such that $c_n \preceq c'$.

The following lemma states that small configurations simulate larger ones with respect to the backward relation.

Lemma 1. *For configurations c_1, c_2 and c_3, if $c_1 \rightsquigarrow c_2$ and $c_3 \preceq c_1$ then there is a configuration c_4 such that $c_3 \rightsquigarrow c_4$ and $c_4 \preceq c_2$.*

The following lemma relates the backward and forward transition relations.

Lemma 2. *Consider configurations c_1 and c_2. If $c_1 \rightsquigarrow c_2$ then $c_2 \longrightarrow c_1{\uparrow}$. If $c_1 \longrightarrow c_2{\uparrow}$ then $c_2 \rightsquigarrow c_1{\downarrow}$.*

6 Algorithm

We present here the reachability algorithm and show its partial correctness.

The algorithm inputs two sets C_{Init} and C_F of configurations and checks whether $C_{Init} \xrightarrow{*}_A C_F\uparrow$. The algorithm maintains two sets of configurations: a set `ToExplore`, initialized to C_F, of configurations that have not yet been analyzed; and a set `Explored`, initialized to the empty set, of configurations that contains information about the configurations that have already been analyzed. The algorithm preserves the following two invariants: (i) $C_{Init} \xrightarrow{*}_A$ (`ToExplore` $\bigcup$ `Explored`)$\uparrow$ implies $C_{Init} \xrightarrow{*}_A C_F\uparrow$; and (ii) If $C_{Init} \xrightarrow{*}_A C_F\uparrow$, then there is $c \in$ `ToExplore` such that both $Rank(C_{Init})(c) < \infty$ and $\forall c' \in$ `Explored`. $Rank(C_{Init})(c) < Rank(C_{Init})(c')$.

Input: Two sets C_{Init} and C_F of configurations.

Output: $C_{Init} \xrightarrow{*}_A C_F\uparrow$?

`ToExplore` := C_F
`Explored` := $\emptyset$
while `ToExplore` $\neq \emptyset$ **do**
  remove some c from `ToExplore`
  if $\exists c' \in C_{Init}.\, c \preceq c'$ **then**
    return true
  else if $\exists c' \in$ `Explored`$.\, c' \preceq c$ **then**
    discard c
  else
    `ToExplore` := `ToExplore` $\bigcup \{c' \mid c \rightsquigarrow c'\}$
    `Explored` :=
    $\{c\} \bigcup \{c' \mid c' \in$ `Explored` $\wedge (c \not\preceq c')\}$
  end if
end while
return false

Due to the invariants, the following two conditions can be checked during each step of the algorithm: (i) From the second invariant, if `ToExplore` becomes empty then the algorithm terminates with a negative answer; and (ii) From the first invariant and the definition of $\longrightarrow_A$, if a configuration in $C_{Init}\downarrow$ is detected then the algorithm terminates with a positive answer.

The following theorem follows immediately from the invariants together with Lemmas 1 and 2.

Theorem 1. *The reachability algorithm is partially correct.*

7 Termination

In this section, we give an overview of the termination proof for the reachability algorithm. The full details can be found in [1]. Let $\mathbb{N}^{>0}$ denote the set of positive integers. For a set A and a preorder on A, we say that $\preceq$ is a *well quasi-ordering (WQO)* on A if the following property is satisfied: for any infinite sequence $a_0, a_1, a_2, \ldots$ of elements in A, there are i, j such that $i < j$ and $a_i \preceq a_j$. A simple example of a WQO is the standard ordering on natural numbers. We extend the ordering $\preceq$ to an ordering $\preceq^*$ on the set A^* of finite words over A as follows: $w_1 \preceq^* w_2$ if there is an order-preserving injection from w_1 to w_2 such that each element in w_1 is mapped to an element in w_2 which is larger wrt $\preceq$. It is well-known that $\preceq^*$ is a WQO (see e.g.[2]). Since multisets and vectors are special cases of words it

follows, for instance, that vectors of multisets of vectors of natural numbers are WQO (this particular property will be used later in the proof).

Consider a graph $g = (V, succ, \lambda)$. A graph b is said to be a *block* of g if b is a maximal part of g which is connected. A vertex is said to be *unguarded* if it is a leaf and there is no variable $x \in X$ with $\lambda(x) = v$. For a graph g, we define the *degree* of g, denoted $deg\,(g)$, to be the number of unguarded vertices in g. A graph is said to be *compact* if it does not contain simple vertices. Intuitively, a graph is *compact* if it cannot be reduced due to contraction. An *encoding* is a tuple $e = (V, succ, \lambda, \#)$ where $g = (V, succ, \lambda)$ is a compact graph, and $\# : V \times V \to \mathbb{N}^{>0}$ is a partial mapping such that $\#(v_1, v_2) \neq \perp$ iff $v_2 = succ(v_1)$. In other words, $\#$ associates a positive integer to each edge in g.

Fix a graph $g = (V, succ, \lambda)$. We define $enc\,(g)$ to be the encoding we get from g by applying contraction as much as possible (until the resulting graph cannot be reduced any more using contraction). Furthermore, for vertices v and v', the value of $\#(v, v')$ gives the number of edges between v and v' in g. We will define an ordering $\sqsubseteq$ on graphs (the formal definition of the relation is in [1]). Consider graphs g, g' with encodings $enc\,(g) = (V, succ, \lambda, \#)$ and $enc\,(g') = (V', succ', \lambda', \#')$. Roughly speaking, $g \sqsubseteq g'$ means that for each block b in $enc\,(g)$ there is an corresponding isomorphic block b' in $enc\,(g')$ such that such that $\#(v_1, v_2) \leq \#'(v_1', v_2')$ for all vertices v_1, v_2 in b and their images v_1', v_2' in b'. The ordering $\sqsubseteq$ is a WQO on the set of compact graphs whose degrees are bounded by some $k \in \mathbb{N}^{>0}$. The reason is that in any such a graph, there number of leaves is bounded by $|X| + k$, and hence there are only finitely many types of blocks which may occur in any such graph. This means that an encoding of such a graph can be represented by vectors of multisets of vectors of natural numbers as follows. Suppose that there are ℓ types of blocks. Then, an element of the representation will be of the form $(m_1, \ldots, m_\ell)$, where each m_i is a multiset of vectors of natural numbers corresponding to one type of block: an entry of the vector corresponds to an edge in the block; the natural numbers correspond to the ones which appear on the edges. We need multisets since there is no bound on the number of blocks (of a certain type) which may appear in the graph. This means that $\sqsubseteq$ is a WQO. Also, $g \sqsubseteq g'$ implies $g \preceq g'$, since if $g \sqsubseteq g'$ then we can derive g from g' through the application of a finite sequence of variable deletion, vertex deletion, edge deletion, and contraction operations. Finally, we observe that in the definition $\rightsquigarrow$ no unguarded vertices are introduced. This means that all the configurations which are generated in the reachability algorithm are bounded by some k, and are therefore WQO. This gives the following theorem.

Theorem 2. *The reachability algorithm is guaranteed to terminate.*

8 Experimental Results

Based on our method, we have implemented a prototype in Java. We consider three classes of properties: null-dereferencing, well-formedness of output, and

garbage creation. We consider a set of programs taken from the PALE website [26]. The results, obtained on a 1.1 GHz Pentium M, are summarised below.

Prog.	Prop.	Time	#C$^{ons.}$	#Iter.	Prog.	Prop.	Time	#C$^{ons.}$	#Iter.
`Concat`	Deref	0.4 s	7	3	`Delete`	Deref	0.4 s	8	4
`Fumble`	Deref	0.3 s	3	2	`Reverse`	Deref	0.3 s	2	1
`Walk`	Deref	0.4 s	9	3	`Zip`	Deref	1.9 s	206	12
`Fumble`	Garbage	0.7 s	38	14	`Reverse`	Garbage	0.8 s	55	24
`Reverse`	Well-form.	1.7 s	48	20					

The entry #C$^{ons.}$ gives the total number of minimal configurations added to `ToExplore` in the analysis. The entry **#Iter.** is the number of iterations of the **while**-loop of the algorithm.

For each of the three properties, we give a finite set of minimal configurations violating the property. For instance, for null-dereferencing, the set contains all configurations of the form $c = (q, g)$ defined as follows. There is a transition of the form (q, a, q') where a is of one of the forms $y := x.next$ or $x.next := y$. Also, g is the graph consisting of a single vertex labeled with *null* and x.

9 Conclusions

We have presented a new approach for automatic verification of programs with dynamic heaps. The proposed approach is based on a simple algorithmic principle, and is fully automatic. The main idea is to perform an abstract (over-approximate) reachability analysis using upward-closed sets w.r.t. a suitable preorder on heap graphs. This preorder is shown to be a well-quasi ordering, which guarantees the termination of the analysis.

The results of this paper concern the case of heap structures with 1-next selector. Our approach can however be generalized to heap structures with multiple next selectors. Several extensions of our framework can be done by refining the considered preorder (and the abstraction it induces). For instance, it could be possible (1) to take into account data values attached to objects in the heap, (2) to consider constraints on (and relating) the lengths of (contracted) paths, and (3) to consider in integer program variables whose values are related to the lengths of paths in the heap. Such extensions with arithmetical reasoning can be done in our framework by considering preorders involving for instance gap-order constraints.

References

1. Abdulla, P.A., Bouajjani, A., Cederberg, J., Haziza, F., Rezine, A.: Monotonic abstraction for programs with dynamic memory heaps. Technical Report 2008-015, Dept. of Information Technology, Uppsala University, Sweden (April 2008)
2. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: Algorithmic analysis of programs with well quasi-ordered domains. Inf. Comput. 160(1-2), 109–127 (2000)

3. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Regular model checking without transducers (on efficient verification of parameterized systems). In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007)
4. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized Verification of Infinite-State Processes with Global Conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 145–157. Springer, Heidelberg (2007)
5. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Inf. Comput. 127(2), 91–101 (1996)
6. Abdulla, P.A., Jonsson, B.: Model checking of systems with many identical timed processes. Theor. Comput. Sci. 290(1), 241–264 (2003)
7. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A Survey of Regular Model Checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004)
8. Balaban, I., Pnueli, A., Zuck, L.D.: Shape analysis of single-parent heaps. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 91–105. Springer, Heidelberg (2007)
9. Bardin, S., Finkel, A., Lozes, É., Sangnier, A.: From pointer systems to counter systems using shape analysis. In: Proceedings of the 5th Intern. Workshop on Automated Verification of Infinite-State Systems (AVIS 2006) (2006)
10. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
11. Bouajjani, A.: Languages, rewriting systems, and verification of infinite-state systems. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 24–39. Springer, Heidelberg (2001)
12. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with Lists Are Counter Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 517–531. Springer, Heidelberg (2006)
13. Bouajjani, A., Habermehl, P., Moro, P., Vojnar, T.: Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 13–29. Springer, Heidelberg (2005)
14. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)
15. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract Regular Model Checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004)
16. Distefano, D., Berdine, J., Cook, B., O'Hearn, P.: Automatic Termination Proofs for Programs with Shape-shifting Heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006)
17. Distefano, D., O'Hearn, P., Yang, H.: A Local Shape Analysis Based on Separation Logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
18. Emerson, E.A., Namjoshi, K.S.: On model checking for non-deterministic infinite-state systems. In: LICS, pp. 70–80 (1998)
19. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: Proceedings of LICS 1999, pp. 352–359. IEEE Computer Society, Los Alamitos (1999)

20. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! TCS 256(1-2), 63–92 (2001)
21. Jensen, J., Jørgensen, M., Klarlund, N., Schwartzbach, M.: Automatic Verification of Pointer Programs Using Monadic Second-order Logic. In: Proc. of PLDI 1997 (1997)
22. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. Theor. Comput. Sci., 93–112 (2001)
23. Lahiri, S.K., Qadeer, S.: Verifying properties of well-founded linked lists. In: POPL, pp. 115–126 (2006)
24. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, M.: Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 181–198. Springer, Heidelberg (2005)
25. O'Hearn, P.W.: Separation logic and program analysis. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, p. 181. Springer, Heidelberg (2006)
26. PALE - the Pointer Assertion Logic Engine, `http://www.brics.dk/PALE/`
27. Revesz, P.Z.: A closed-form evaluation for datalog queries with integer (gap)-order constraints. Theor. Comput. Sci. 116(1&2), 117–149 (1993)
28. Reynolds, J.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proc. of LICS 2002. IEEE CS Press, Los Alamitos (2002)
29. Sagiv, S., Reps, T., Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic. TOPLAS 24(3) (2002)
30. Wolper, P., Boigelot, B.: Verifying systems with infinite but regular state spaces. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 88–97. Springer, Heidelberg (1998)