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 and [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., [2, 20]), or to define generic exact/abstract analysis procedures (e.g., [30, 22, 11, 7]).

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, 18, 19, 6]). 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 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., [28, 25]), automata-based frameworks (see, e.g., [21, 14]), 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., [29, 17, 21, 14]). 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 [17, 10], 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., [24, 13, 23, 12, 16, 8]). 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 [12, 9], 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 $x \neq \mathit{null}$, $x.\mathit{next} = y$ where $x \neq \mathit{null}$, or $x := y.\mathit{next}$ where $x, y \neq \mathit{null}$. 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., $x = y.\mathit{next}$, $x \neq y.\mathit{next}$, etc, can be expressed using the given set.

A *graph* g is a triple $(V, \mathit{succ}, \lambda)$ 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 $\mathit{succ}(\lambda(\mathit{null})) = \perp$. Intuitively, the vertices correspond to the list cells. The function *succ* defines the successors of the cells. If $\mathit{succ}(v) = \perp$, 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 $x := y$, $\lambda(y) \neq \perp$, $\mathit{succ}' = \mathit{succ}$, and $\lambda' = \lambda\,[x \leftarrow \lambda(y)]$.
- a is of the form $x := y.\mathit{next}$, $\lambda(y) \neq \perp$, $\mathit{succ}(\lambda(y)) \neq \perp$, $\mathit{succ}' = \mathit{succ}$, and $\lambda' = \lambda\,[x \leftarrow \mathit{succ}(\lambda(y))]$.
- a is of the form $x.\mathit{next} := y$, $\lambda(x) \neq \perp$, $\lambda(y) \neq \perp$, $\lambda(x) \neq \lambda(\mathit{null})$, $\lambda' = \lambda$, and $\mathit{succ}' = \mathit{succ}\,[\lambda(x) \leftarrow \lambda(y)]$.

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 $g = (V, \mathit{succ}, \lambda)$.

For $v_1, v_2 \in V$, we use $(g.\mathit{succ})\,[v_1 \leftarrow v_2]$ to denote the graph $g' = (V', \mathit{succ}', \lambda')$ where $V' = V$, $\lambda' = \lambda$, and $\mathit{succ}' = \mathit{succ}\,[v_1 \leftarrow v_2]$. 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 $|succ^{-1}(v)| = 1$, $succ(v) \neq \perp$, 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 $succ(v) = \perp$, $succ^{-1}(v) = \emptyset$, 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 $succ(v_1) \neq \perp$, $g_1 = g \oplus v_2$, $g_2 = (g_1.succ)[v_2 \leftarrow succ_1(v_1)]$, $g_3 = (g_2.succ)[v_1 \leftarrow v_2]$, 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 $\lambda(x) \neq \perp$, 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 $\lambda(x) \neq \perp$ and $succ(\lambda(x)) \neq \perp$, 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 $\lambda(x) \neq \perp$, 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 $succ(v) = \perp$ 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 $(g.\lambda)[x \leftarrow \perp]$.

Operations on edges. If $\lambda(x) \neq \perp$, $\lambda(y) \neq \perp$ 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 $\lambda(x) \neq \perp$ 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 $succ(v_1) \neq \perp$, $g_1 = g \oplus v_2$, $g_2 = (g_1.succ)\,[v_2 \leftarrow succ_1(v_1)]$, $g_3 = (g_2.succ)\,[v_1 \leftarrow v_2]$, and $g' = (g_3.succ)\,[\lambda_3(x) \leftarrow v_2]$.

If $\lambda(x) \neq \perp$, we denote $g \boxminus (x \rightarrow)$ as $(g.succ)\,[\lambda(x) \leftarrow \perp]$, *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*: $g = (g'.succ)\,[v \leftarrow \perp]$ 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, $succ'(v_1) = v_2$, $succ'(v_2) = v_3$, $g_1 = (g'.succ)\,[v_2 \leftarrow \perp]$, $g_2 = (g_1.succ)\,[v_1 \leftarrow v_3]$ 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.* $c\!\uparrow = \{c' \,|\, c \preceq c'\}$. We use $c\!\downarrow$ to denote the *downward closure* of c, *i.e.* $c\!\downarrow = \{c' \,|\, c' \preceq c\}$. 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 will 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 $g' \in (g \oplus y)$.
   The difference compared 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) $\lambda(x) \neq \bot$, $\lambda(y) \neq \bot$, $\lambda(y) \neq \lambda(\mathit{null})$, $\mathit{succ}(\lambda(y)) = \bot$ 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) $\lambda(x) = \bot$, $\lambda(y) \neq \bot$, $\mathit{succ}(\lambda(y)) \neq \bot$ and $g' = g$.
(e) $\lambda(x) = \bot$, $\lambda(y) \neq \bot$, $\lambda(y) \neq \lambda(\mathit{null})$, $\mathit{succ}(\lambda(y)) = \bot$ and $g' \in g \boxplus (y \to)$.
(f) $\lambda(x) = \bot$, $\lambda(y) = \bot$ and there are graphs g_1, g_2, g_3 such that $g_1 = g \oplus x$, $g_2 \in g_1 \oplus_{x\leftarrow} y$, $g_3 = g_2 \boxplus (y \to x)$ 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 $x.\mathit{next} := y$ and one of the following conditions is satisfied:
(a) $\lambda(x) \neq \bot$, $\mathit{succ}(\lambda(x)) \neq \bot$, $\lambda(y) \neq \bot$, $\mathit{succ}(\lambda(x)) = \lambda(y)$ and $g' = g \boxminus (x \to)$.
(b) $\lambda(x) \neq \bot$, $\mathit{succ}(\lambda(x)) \neq \bot$, $\lambda(y) = \bot$ and $g' = g_1 \boxminus (x \to)$, where $g_1 = g \oplus_{x\to} y$.
(c) $\lambda(x) \neq \bot$, $\mathit{succ}(\lambda(x)) = \bot$, $\lambda(y) \neq \bot$, $\lambda(x) \neq \lambda(\mathit{null})$ and $g' = g$.
(d) $\lambda(x) \neq \bot$, $\mathit{succ}(\lambda(x)) = \bot$, $\lambda(y) = \bot$, $\lambda(x) \neq \lambda(\mathit{null})$ and $g' \in g \oplus y$.
(e) $\lambda(x) = \bot$,$\lambda(y) \neq \bot$ and $g' = g_1 \boxminus (x \to)$, 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 $g' = g_2 \boxminus (x \to)$.

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 $\mathit{Rank}(C)(c)$ 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

In this section, we present 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) $C_{Init} \xrightarrow{*}_A$ $(\texttt{ToExplore} \bigcup \texttt{Explored})\uparrow$ implies $C_{Init} \xrightarrow{*}_A C_F\uparrow$; and (ii) If $C_{Init} \xrightarrow{*}_A C_F\uparrow$, then there is $c \in \texttt{ToExplore}$ such that both $Rank(C_{Init})(c) < \infty$ and $\forall c' \in \texttt{Explored}.\ Rank(C_{Init})(c) < Rank(C_{Init})(c')$.

```
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 $\neq \emptyset$
            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 $\wedge (c \not\preceq c')\}$
        return (false)
        end
```

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].In the proof, we will rely on the theory of well quasi-orderings. Furthermore, we will exploit particular properties of our graphs implied by the fact that each vertex is restricted to have at most one successor. 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 natural numbers are WQO (this particular property will be later used in the proof).

Consider a graph $g = (V, \mathit{succ}, \lambda)$. A graph $g' = (V', \mathit{succ}', \lambda')$ is said to be a *block* in g if V' is a maximal subset of V satisfying the following property: V' contains a vertex v such that, for all $v' \in V'$, there is path along the edges of g from v' to v. 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 as $\mathit{deg}\,(v) := d_2 - d_1$ where d_2 is the number of unguarded vertices and d_1 is the number of roots 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 $e = (V, \mathit{succ}, \lambda, \#)$ where $g = (V, \mathit{succ}, \lambda)$ 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 $v_2 = \mathit{succ}(v_1)$. In other words, $\#$ associates a positive integer to each edge in g. We call g the *signature* of e and denote it by $\mathit{sig}\,(e)$.

Fix a graph $g = (V, \mathit{succ}, \lambda)$. We define $\mathit{enc}\,(g)$ to be the encoding we get from g by keeping only the non-simple vertices. Furthermore, for vertices v and v', the value of $\#(v, v')$ gives the number of edges between v and v'. We will define an ordering $\sqsubseteq$ on graphs (the formal definition of the relation is in [1]).Consider graphs g, g' with encodings $\mathit{enc}\,(g) = (V, \mathit{succ}, \lambda, \#)$ and $\mathit{enc}\,(g') = (V', \mathit{succ}', \lambda', \#')$. Roughly speaking, $g \sqsubseteq g'$ means that there is an injection from V to V' such that $\#(v, v') \leq \#'(h(v), h(v'))$ for all $v, v' \in V$. 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 are at most k blocks with a positive degree. Also, there are only finitely many types of blocks which are of degree 0. This means that an encoding of such a graph can be represented by vectors of multisets of natural numbers: an entry of a vector corresponds to an edge in the graph; the natural numbers correspond to the ones which appear on the edge. We need multisets since there is no bound on the number of blocks of degree 0 which may appear in the graph. Furthermore, 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; which implies that $g \preceq g'$. 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.

These observations give termination of the reachability algorithm as follows. Suppose that the algorithm does not terminate. This means that we add to the set `ToExplore` an infinite sequence $c_0, c_1, c_2, \ldots$ of configurations, where c_i is of the form (q, g_i), and and where $c_i \not\preceq c_j$ for all i, j with $i < j$. We know that

that the set $\{g_0, g_1, g_2, \ldots\}$ has bounded degree, and therefore there are i, j such that $i < j$ and $g_i \sqsubseteq g_j$. It follows that $g_i \preceq g_j$, and hence $c_i \preceq c_j$ which is a contradiction. 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. P. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, and A. Rezine. Monotonic abstraction for programs with dynamic memory heaps. Technical Report 2008-015, Dept. of Information Technology, Uppsala University, Sweden, April 2008.
2. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Inf. Comput.*, 160(1-2):109–127, 2000.
3. P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *TACAS'07*, pages 721–736. LNCS 4424, 2007.
4. P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV'07*, pages 145–157. LNCS 4590, 2007.
5. P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Inf. Comput.*, 127(2):91–101, 1996.
6. P. A. Abdulla and B. Jonsson. Model checking of systems with many identical timed processes. *Theor. Comput. Sci.*, 290(1):241–264, 2003.
7. P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A Survey of Regular Model Checking. In *CONCUR'04*, pages 35–48. LNCS 3170, 2004.
8. I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis of single-parent heaps. In *VMCAI'07*, pages 91–105. LNCS 4349, 2007.
9. S. Bardin, A. Finkel, É. Lozes, and A. Sangnier. From pointer systems to counter systems using shape analysis. In *Proceedings of the 5th Intern. Workshop on Automated Verification of Infinite-State Systems (AVIS'06)*, 2006.
10. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV'07*, pages 178–192. LNCS 4590, 2007.
11. A. Bouajjani. Languages, rewriting systems, and verification of infinite-state systems. In *ICALP'01*, pages 24–39. LNCS 2076, 2001.
12. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists are Counter Automata. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.
13. A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In *Proc. of TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.
14. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.
15. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. of CAV'04*, volume 3114 of *LNCS*. Springer, 2004.
16. D. Distefano, J. Berdine, B. Cook, and P. O'Hearn. Automatic Termination Proofs for Programs with Shape-shifting Heaps. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.
17. D. Distefano, P. O'Hearn, and H. Yang. A Local Shape Analysis Based on Separation Logic. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.
18. E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *LICS*, pages 70–80, 1998.
19. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proceedings of LICS '99*, pages 352–359. IEEE Computer Society, 1999.

20. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *TCS*, 256(1-2):63–92, 2001.
21. J. Jensen, M. Jørgensen, N. Klarlund, and M. Schwartzbach. Automatic Verification of Pointer Programs Using Monadic Second-order Logic. In *Proc. of PLDI'97*, 1997.
22. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theor. Comput. Sci.*, 256(1-2):93–112, 2001.
23. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL*, pages 115–126, 2006.
24. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*. Springer, 2005.
25. P. W. O'Hearn. Separation logic and program analysis. In *SAS'06*, page 181. LNCS 4134, 2006.
26. PALE - the Pointer Assertion Logic Engine. `http://www.brics.dk/PALE/`.
27. P. Z. Revesz. A closed-form evaluation for datalog queries with integer (gap)-order constraints. *Theor. Comput. Sci.*, 116(1&2):117–149, 1993.
28. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE CS Press, 2002.
29. S. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.
30. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *CAV'98*, pages 88–97. LNCS 1427, 1998.

A Proofs of Lemmas - Section 5

Lemma 1

For graphs g_1, g_2, and g_3, we want the following. If $g_3 \lhd g_1$ and $g_1 \overset{a}{\rightsquigarrow} g_2$ for some action a, then there should exist a g_4 such that $g_3 \overset{a}{\rightsquigarrow} g_4$ and $g_4 \unlhd g_2$. We define g_4 by considering the four cases corresponding to the definition of $g_3 \lhd g_1$. For each case of the ordering definition, we find such a g_4 for one or two subcases for each of the five actions. The rest of the subcases can be derived in a similar manner.

1. **Variable Deletion:** $g_3 = g_1 \ominus z$ for some variable z.
   If $z \notin \{x, y\}$. Define $g_4 = g_2 \ominus z$. Both $g_4 \lhd g_2$ and $g_3 \overset{a}{\rightsquigarrow} g_4$ follow immediately from the definitions. We assume $z \in \{x, y\}$ and consider the five possible forms of a:
   (a) a is of the form $x = y$. Let $z = y$ (The case $z = x$ is obtained by permuting x and y in the definition of $g_1 \overset{a}{\rightsquigarrow} g_2$). We have $g_3 = g_1 \ominus y$. Say $\lambda_1(x) = \perp$, $\lambda_1(y) \neq \perp$ and $g_2 = g_1 \oplus_{=y} x$, we get $\lambda_3(x) = \perp$ and $\lambda_3(y) = \perp$. We define $g_4 = g_2$, and get $g_4 = g_2 = g_1 \oplus_{=y} x$. Observe $g_1 \in (g_3 \oplus x)$, and hence $g_3 \overset{a}{\rightsquigarrow} g_4$.
   (b) a is of the form $x \neq y$. Let $z = y$. (The case $z = x$ is obtained by permutation). We have $g_3 = g_1 \ominus y$. Consider the case where $\lambda_1(x) = \perp$, $\lambda_1(y) \neq \perp$ and $g_2 \in g_1 \oplus_{\neq y} x$. We get $\lambda_3(x) = \perp$ and $\lambda_3(y) = \perp$. We define $g_4 = g_2$. Observe $g_4 \in g_1 \oplus_{\neq x} y$ and $g_1 \in (g_3 \oplus x)$. We deduce $g_3 \overset{a}{\rightsquigarrow} g_4$.
   (c) a is of the form $x := y$. We look at both cases $z = x$ and $z = y$.
      – In case $z = x$, we get $g_3 = g_1 \ominus x$.
        Suppose $\lambda_1(x) \neq \perp$, $\lambda_1(y) = \perp$ and $g_2 = g \ominus x$ where $g = g_1 \oplus_{=x} y$. We define $g_4 = g_2$. We get $\lambda_3(x) = \perp$ and $\lambda_3(y) = \perp$. Observe g_2 is in $(g_1 \ominus x) \oplus y$. This means $g_4 \in (g_3 \oplus y)$, and hence $g_3 \overset{a}{\rightsquigarrow} g_4$.
      – In case $z = y$, we get $g_3 = g_1 \ominus y$. Suppose $\lambda_1(x) = \perp$, $\lambda_1(y) \neq \perp$ and $g_2 = g_1$. Define $g_4 = g_2$. We have $\lambda_3(x) = \perp$ and $\lambda_3(y) = \perp$. Observe that $g_1 \in (g_1 \ominus y) \oplus y$. Hence, $g_4 \in (g_3 \oplus y)$. We deduce $g_3 \overset{a}{\rightsquigarrow} g_4$.
   (d) a is of the form $x := y.next$. We consider both cases $z = x$ and $z = y$.
      – In case $z = x$, we get $g_3 = g_1 \ominus x$. Say $\lambda_1(x) \neq \perp$, $\lambda_1(y) = \perp$ and there are graphs g, g' such that $g_2 = g \ominus x$, $g = g' \boxplus (y \to x)$ and $g' \in g_1 \oplus_{x\leftarrow} y$. Define $g_4 = g_2$. We have $\lambda_3(x) = \perp$ and $\lambda_3(y) = \perp$. Observe $g_4 = g_2 = g \oplus x$, $g = g' \boxplus (y \to x)$, $g' \in g_1 \oplus_{x\leftarrow} y$ and $g_1 \in (g_1 \ominus x) \oplus x$. $g_3 \overset{a}{\rightsquigarrow} g_4$ follows from the definitions.
      – In case $z = y$, we get $g_3 = g_1 \ominus y$. Suppose $\lambda_1(x) \neq \perp$, $\lambda_1(y) \neq \perp$, $succ_1(\lambda_1(y)) \neq \perp$, $succ_1(\lambda_1(y)) = \lambda_1(x)$ and $g_2 = g_1 \ominus x$. We define $g_4 = g_2$. We have $\lambda_3(x) \neq \perp$, $\lambda_3(y) = \perp$. Observe $g_4 = g_2 = g_1 \ominus x$, $g_1 = g_1 \boxplus (y \to x)$, $g_1 \in (g_1 \ominus y) \oplus_{x\leftarrow} y$. $g_3 \overset{a}{\rightsquigarrow} g_4$ follows from the definitions.
   (e) a is of the form $x.next := y$

- In case $z = x$, we get $g_3 = g_1 \ominus x$. Say $\lambda_1(x) \neq \bot$, $succ_1(\lambda_1(x)) = \bot$, $\lambda_1(y) \neq \bot$, $\lambda_1(x) \neq \lambda_1(null)$ and $g_2 = g_1$. We have $\lambda_3(x) = \bot$, $\lambda_3(y) \neq \bot$, Define $g_4 = g_2$. Observe $g_4 = g_1 = g_1 \boxminus (x \rightarrow)$, and $g_1 \in g_1 \oplus_{y\leftarrow} x$.
- In case $z = y$, we get $g_3 = g_1 \ominus y$. Say $\lambda_1(x) \neq \bot$, $succ_1(\lambda_1(x)) = \bot$, $\lambda_1(y) \neq \bot$, $\lambda_1(x) \neq \lambda_1(null)$ and $g_2 = g_1$. We have $\lambda_3(x) \neq \bot$, $succ_3(\lambda_3(x)) = \bot$, $\lambda_3(y) \neq \bot$, $\lambda_3(x) \neq \lambda_3(null)$ Define $g_4 = g_2$. Observe $g_4 \in (g \boxminus (y \rightarrow)) \oplus y$.

2. **Vertex Deletion:** $g_3 = g_1 \ominus v$ for some isolated vertex v.
   We consider the five possible forms of a:
   (a) a is of the form $x = y$. Say $\lambda_1(x) = \bot$, $\lambda_1(y) = \bot$ and $g_2 = g \oplus_{=x} y$ for a $g \in (g_1 \oplus x)$. We have $\lambda_3(y) = \bot$ and $\lambda_3(x) = \bot$. We differentiate two cases depending on whether v is also isolated in g_2.
      - v is isolated in g_2. We define $g_4 = g_2 \ominus v$. Observe $g_4 = (g \oplus_{=x} y) \ominus v$ for a $g \in (g_1 \oplus x)$. Since v is isolated in g_2, we get $g_4 = g' \oplus_{=x} y$ and $g' \in (g_3 \oplus x)$, hence $g_3 \overset{a}{\rightsquigarrow} g_4$.
      - v is not isolated in $g_2 = (g \oplus_{=x} y)$ with $g \in (g_1 \oplus x)$. We define $g_4 = g_2$. Observe that v is pointed by x and y. Also, $g \in g_3 \oplus x$ (x is made to point to a new vertex).

   (b) a is of the form $x \neq y$. Consider the case where $\lambda_1(x) = \bot$, $\lambda_1(y) = \bot$ and $g_2 \in g \oplus_{\neq x} y$ for a $g \in (g_1 \oplus x)$. We separate two subcases:
      - If $\lambda_2(x) \neq v$ and $\lambda_2(y) \neq v$, then define $g_4 = g_2 \ominus v$. Observe: $g_4 \in (g \ominus v) \oplus_{\neq x} y$ and $(g \ominus v) \in ((g_1 \ominus v) \oplus x)$.
      - If $\lambda_2(x) = v$ ($\lambda_2(y) = v$ is similar), then define $g_4 = g_2$. Observe: $g_2 \in (g \oplus_{\neq x} y)$ and $g \in ((g_1 \ominus v) \oplus x)$. ($x$ is made to point to a new vertex).

   (c) a is of the form $x := y$. Say $\lambda_1(x) = \bot$, $\lambda_1(y) = \bot$ and $g_2 \in (g_1 \oplus y)$. Define $g_4 = g_2$ if $\lambda_2(y) = v$, and $g_4 = g_2 \ominus v$ otherwise. Observe that in both cases we have $g_4 \in g_3 \oplus y$.

   (d) a is of the form $x := y.next$. Say $\lambda_1(x) \neq \bot$, $\lambda_1(y) = \bot$ and there are graphs g, g' such that $g_2 = g' \ominus x$, $g' = g \boxplus (y \rightarrow x)$ and $g \in g_1 \oplus_{x\leftarrow} y$. We find $\tilde{g}, \tilde{g}'$ such that $g_4 = \tilde{g} \ominus x$, $\tilde{g}' = \tilde{g} \boxplus (y \rightarrow x)$ and $\tilde{g} = g_3 \oplus_{x\leftarrow} y$.
      - If $\lambda_2(y) \neq v$, then define $g_4 = g_2 \ominus v$. Choose $\tilde{g}' = g' \ominus v$ and $\tilde{g} = g \ominus v$.
      - If $\lambda_2(y) = v$, then define $g_4 = g_2$. Choose $\tilde{g}' = g'$ and $\tilde{g} = g$. (the vertex removed in $g_3 \ominus x$ is added as a predecessor of x).

   (e) a is of the form $x.next := y$. We Consider the case where: $\lambda_1(x) = \bot$, $\lambda_1(y) \neq \bot$ and $g_2 = g \boxminus (x \rightarrow)$, where $g \in g_1 \oplus_{y\leftarrow} x$. We find a g' such that $g_4 = g' \boxminus (x \rightarrow)$, where $g' \in g_3 \oplus_{y\leftarrow} x$.
      - If $\lambda_2(y) \neq v$, then define $g_4 = g_2 \ominus v$. Choose $g' = g \ominus v$.
      - If $\lambda_2(y) = v$, then define $g_4 = g_2$. Choose $g' = g$. (the vertex removed in $g_3 \ominus x$ is added as a predecessor of x).

3. **Edge Deletion:** We consider the five possible forms of a:
   (a) a is of the form $x = y$. In all four subcases, the deletion of an edge followed by firing the action a results in the same configuration g_4 as firing a then deleting the same edge.

(b) a is of the form $x \neq y$. In all four subcases, the deletion of an edge followed by firing the action a results in the same configuration g_4 as firing a then deleting the same edge.

(c) a is of the form $x := y$. In all four subcases, the deletion of an edge followed by firing the action a results in the same configuration g_4 as firing a then deleting the same edge.

(d) a is of the form $x := y.\mathit{next}$. We look at the case where $\lambda_1(x) \neq \bot$, $\lambda_1(y) \neq \bot$, $\mathit{succ}(\lambda_1(y)) \neq \bot$, $\mathit{succ}(\lambda_1(y)) = \lambda_1(x)$ and $g_2 = g_1 \ominus x$. We distinguish two cases:
  - If the deleted edge is not the one between y and its successor; we define $g_4 = g_3 \ominus x$, and hence $g_3 \overset{a}{\rightsquigarrow} g_4$. Observe $g_4 \lhd g_2$ by deletion of the same edge.
  - If the deleted edge is the one between y and its successor. Observe $\lambda_3(x) \neq \bot$, $\lambda_3(y) \neq \bot$, $\lambda_3(y) \neq \lambda_3(\mathit{null})$ and $\mathit{succ}_3(\lambda_3(y)) = \bot$. We define $g_4 = g \ominus x$, where $g = g_1 \boxplus (y \rightarrow x)$. Observe this means $g_4 = g_2$, and $g_3 \overset{a}{\rightsquigarrow} g_4$.

(e) a is of the form $x.\mathit{next} := y$. Say $\lambda_1(x) \neq \bot$, $\mathit{succ}(\lambda_1(x)) \neq \bot$, $\lambda_1(y) \neq \bot$, $\mathit{succ}(\lambda_1(x)) = \lambda_1(y)$ and $g_2 = g_1 \boxminus (x \rightarrow)$. We distinguish two cases:
  - Suppose the deleted edge is not the one between x and its successor; we define $g_4 = g_3 \boxminus (x \rightarrow)$. $g_4 \lhd g_2$ is obtained by deletion of the same edge.
  - Suppose the deleted edge is the one between x and its successor. This means $g_2 = g_3$. We define $g_4 = g_3$.

4. **Contraction:** We let v be the contracted simple vertex in $g_3 \lhd g_1$.
We consider a case for each of the five possible forms of a:

(a) a is of the form $x = y$. We look at the case where : $\lambda_1(x) = \bot$, $\lambda_1(y) = \bot$ and $g_2 = g \oplus_{=x} y$ for a $g \in (g_1 \oplus x)$. We distinguish two cases:
  - If v is simple in g_2; we define $g_4 = g' \oplus_{=x} y$ with g' obtained by contracting g and eliminating v. Observe $g' \in (g_3 \oplus x)$.
  - If v is not simple in g_2, we define $g_4 = g' \oplus_{=x} y$ with $g' = g$. Observe $g' \in (g_3 \oplus x)$ by insertion of a simple vertex.

(b) a is of the form $x \neq y$. We Consider $\lambda_1(x) = \bot$, $\lambda_1(y) = \bot$ and $g_2 \in g \oplus_{\neq x} y$ for a $g \in (g_1 \oplus x)$. We distinguish two cases:
  - If v is simple in g_2; we define $g_4 \lhd g_2$ by contracting g_2 and eliminating v. Observe $g_4 \in g' \oplus_{\neq x} y$ and $g' \in (g_3 \oplus x)$ for g' obtained by contraction of g and elimination of v.
  - If v is not simple in g_2, we define $g_4 = g_2$. Observe $g_2 \in g \oplus_{\neq x} y$ and $g \in (g_3 \oplus x)$ by insertion of a simple vertex pointed by $x$for the same g. g is the same as in the definition of $g_1 \overset{a}{\rightsquigarrow} g_2$.

(c) a is of the form $x := y$. We look at the case where $\lambda_1(x) = \bot$, $\lambda_1(y) = \bot$ and $g_2 \in g_1 \oplus y$. We distinguish two cases:
  - If v is simple in g_2; we define $g_4 \lhd g_2$ by contracting g_2 and eliminating v. Observe $g_4 \in g_3 \oplus y$.
  - If v is not simple in g_2, we define $g_4 = g_2$. Observe $g_4 \in (g_3 \oplus x)$ by the insertion of a simple vertex.

(d) a is of the form $x := y.next$. We consider $\lambda_1(x) \neq \bot$, $\lambda_1(y) = \bot$ and there are graphs g, g' such that $g_2 = g' \ominus x$, $g' = g \boxplus (y \to x)$ and $g \in g_1 \oplus_{x\leftarrow} y$. We distinguish two cases:
  - If v is simple in g_2; we define $g_4 \lhd g_2$ by contracting g_2 and eliminating v. Observe $g_4 = \tilde{g}' \ominus x$, $\tilde{g}' = \tilde{g} \boxplus (y \to x)$ and $\tilde{g} \in g_3 \oplus_{x\leftarrow} y$ for $\tilde{g}$, resp. $\tilde{g}'$, obtained by contraction and elimination of v in g, resp. g'.
  - If v is not simple in g_2, we define $g_4 = g_2$. Observe $g_4 = g \ominus x$, $g = g' \boxplus (y \to x)$ and $g' \in g_3 \oplus_{x\leftarrow} y$ (by insertion of a simple vertex before x and making y point to it). Hence $g_3 \overset{a}{\rightsquigarrow} g_4$.

(e) a is of the form $x.next := y$ We consider $\lambda_1(x) = \bot$,$\lambda_1(y) \neq \bot$ and $g_2 = g \boxminus (x \to)$, where $g \in g_1 \oplus_{y\leftarrow} x$. We distinguish two cases:
  - If v is simple in g_2; we define $g_4 \lhd g_2$ by contracting g_2 and eliminating v. For g' obtained by contraction and elimination of v from g, we observe $g_2 = g' \boxminus (x \to)$ with $g' \in g_1 \oplus_{y\leftarrow} x$.
  - If v is not simple in g_2, we define $g_4 = g_2$. Observe that $g_2 = g \boxminus (x \to)$, with $g \in g_1 \oplus_{y\leftarrow} x$ (insertion of a simple vertex before x and making y point to it).

Lemma 2

Suppose $g_1 \overset{a}{\rightsquigarrow} g_2$, we show there is a g_3 such that $g_1 \preceq g_3$ and $g_2 \xrightarrow{a}_A g_3$ (in fact $g_2 \xrightarrow{a} g_3$). Let $g_i = (V_i, succ_i, \lambda_i)$ for $i \geq 1$. We consider five cases depending on the type of a:

1. a is of the form $x = y$. We define $g_3 = g_2$ and show $g_1 \preceq g_3$ by considering the following four subcases. For each one of these subcases, $g_2 \xrightarrow{a} g_3$ follows directly from the definitions:
   (a) $\lambda_1(x) \neq \bot$, $\lambda_1(y) \neq \bot$, $\lambda_1(x) = \lambda_1(y)$ and $g_2 = g_1$. $g_1 \preceq g_3$ follows from $g_3 = g_2 = g_1$.
   (b) $\lambda_1(x) \neq \bot$, $\lambda_1(y) = \bot$, and $g_2 = g_1 \oplus_{=x} y$. $g_1 \preceq g_3$ follows by deletion of the variable y.
   (c) $\lambda_1(x) = \bot$ and $\lambda_1(y) \neq \bot$ is similar to case (1b).
   (d) $\lambda_1(x) = \bot$, $\lambda_1(y) = \bot$ and there is a g_4 in $g_1 \oplus x$ such that $g_2 = g_4 \oplus_{=x} y$. Observe that $g_4 \preceq g_3$ holds by deletion of the variable y. We show $g_1 \preceq g_4$ by considering the three subcases resulting from $g_1 \oplus x$:
      i. There is a $v \notin V_1$ such that $g_4 = ((g_1 \oplus v).\lambda)\,[x \leftarrow v]$. $g_1 \preceq g_4$ follows by deletion of the variable x and the vertex v.
      ii. There is a $v \in V_1$ such that $g_4 = (g_1.\lambda)\,[x \leftarrow v]$. $g_1 \preceq g_4$ follows by deletion of the variable x.
      iii. There are $v_1 \in V_1$ and $v_2 \notin V_1$ with $succ_1(v_1) \neq \bot$, and graphs g_5, g_6 and g_7 such that $g_7 = g_1 \oplus v_2$, $g_6 = (g_7.succ)\,[v_2 \leftarrow succ_7(v_1)]$, $g_5 = (g_6.succ)\,[v_1 \leftarrow v_2]$, and $g_4 = (g_5.\lambda)\,[x \leftarrow v_2]$. $g_1 \preceq g_4$ follows by deletion of the variable x, and elimination of the vertex v_2 by contraction.
2. a is of the form $x \neq y$. We define $g_3 = g_2$. We show $g_1 \preceq g_3$ by considering the following four subcases. For each one of these subcases, $g_2 \xrightarrow{a} g_3$ follows directly from the definitions:

(a) $\lambda_1(x) \neq \lambda_1(y)$ and $g_2 = g_1$. $g_1 \preceq g_3$ follows from $g_3 = g_2 = g_1$.

(b) $\lambda_1(x) \neq \bot$, $\lambda_1(y) = \bot$ and $g_2 \in g_1 \oplus_{\neq x} y$. Define $g_3 = g_2$. g_3 is in $g_1 \oplus y$ and $\lambda_3(x) \neq \lambda_3(y)$. We show $g_1 \preceq g_3$ by the arguments in (1(d)i), (1(d)ii) and (1(d)iii).

(c) $\lambda_1(x) = \bot$ and $\lambda_1(y) \neq \bot$ is similar to (2b) where x and y are permuted.

(d) $\lambda_1(x) = \bot$, $\lambda_1(y) = \bot$ and there is a g_4 in $g_1 \oplus x$ such that $g_2 \in g_4 \oplus_{\neq x} y$. We get $g_1 \preceq g_4$ by the arguments in (1(d)i), (1(d)ii) and (1(d)iii). $g_4 \preceq g_2$ by the arguments of (2b) where g_1 is replaced by g_4.

3. a is of the form $x := y$. We consider the four subcases of the definition of $g_1 \overset{a}{\rightsquigarrow} g_2$.

(a) $\lambda_1(x) \neq \bot$, $\lambda_1(y) \neq \bot$ and $\lambda_1(x) = \lambda_1(y)$ and $g_2 = g_1 \ominus x$. Observe $\lambda_2(y) \neq \bot$, $succ_1 = succ_2$ and $\lambda_1 = \lambda_2\,[x \leftarrow \lambda_2(y)]$. Hence $g_2 \xrightarrow{a} g_1$. We define $g_3 = g_1$.

(b) $\lambda_1(x) \neq \bot$, $\lambda_1(y) = \bot$ and there exists a $g_3 = g_1 \oplus_{=x} y$ such that $g_2 = g_3 \ominus x$. Observe $\lambda_3(x) \neq \bot$, $\lambda_3(y) \neq \bot$ and $\lambda_3(x) = \lambda_3(y)$. Using arguments similar to (3a), we deduce $g_2 \xrightarrow{a} g_3$. Furthermore, we have $g_1 \preceq g_3$ by deletion of the variable x.

(c) $\lambda_1(x) = \bot$, $\lambda_1(y) \neq \bot$ and $g_2 = g_1$. We have $\lambda_2(y) \neq \bot$, and define $g_3 = (g_2.\lambda)\,[x \leftarrow \lambda_2(y)]$. We have by construction $g_2 \xrightarrow{a} g_3$, and $g_1 \preceq g_3$ by deletion the variable x.

(d) $\lambda_1(x) = \bot$, $\lambda_1(y) = \bot$ and $g_2 \in g_1 \oplus y$. We have $\lambda_2(y) \neq \bot$, and define $g_3 = (g_2.\lambda)\,[x \leftarrow \lambda_2(y)]$. We have by construction $g_2 \xrightarrow{a} g_3$, and $g_1 \preceq g_3$ by deletion of both variables x and y if $\lambda_2(y) = \bot$.

4. a is of the form $x := y.next$. For each one of the six subcases corresponding to the definition of $g_1 \overset{a}{\rightsquigarrow} g_2$, we define g_3 such that $g_2 \xrightarrow{a} g_3$ and $g_1 \preceq g_3$.

(a) $\lambda_1(x) \neq \bot$, $\lambda_1(y) \neq \bot$, $succ_1(\lambda_1(y)) \neq \bot$, $succ(\lambda_1(y)) = \lambda_1(x)$ and $g_2 = g_1 \ominus x$. We define $g_3 = g_1$.

(b) $\lambda_1(x) \neq \bot$, $\lambda_1(y) \neq \bot$, $\lambda_1(y) \neq \lambda_1(null)$, $succ_1(\lambda_1(y)) = \bot$, and there is a $g_3 = g_1 \boxplus (y \rightarrow x)$ such that $g_2 = g_3 \ominus x$. We deduce $g_1 \preceq g_3$ holds by edge deletion.

(c) $\lambda_1(x) \neq \bot$, $\lambda_1(y) = \bot$, and there are graphs g_3, g_4 such that $g_4 \in g_1 \oplus_{x\leftarrow} y$, $g_3 = g_4 \boxplus (y \rightarrow x)$, and $g_2 = g_3 \ominus x$. Observe that $\lambda_3(x) \neq \bot$, $\lambda_3(y) \neq \bot$, $succ_3(\lambda_3(y))$, $succ_3(\lambda_3(y)) = \lambda_3(x)$ and $g_2 = g_3 \ominus x$, meaning $g_2 \xrightarrow{a} g_3$. We show $g_1 \preceq g_4$ and $g_4 \preceq g_3$ as follows. We start with $g_1 \preceq g_4$, and consider the three subcases corresponding to the definition of $g_4 \in g_1 \oplus_{x\leftarrow} y$.

i. There is a $v \notin V_1$ such that $g_4 = ((g_1 \oplus v).\lambda)\,[y \leftarrow v]$. $g_1 \preceq g_4$ by deleting the variable y and the vertex v.

ii. There is a $v \in V_1$ such that $\lambda_1(v) \neq \lambda_1(null)$ with $succ_1(v) = \bot$ or $succ_1(v) = \lambda_1(x)$ and $g_4 = (g_1.\lambda)\,[y \leftarrow v]$. $g_1 \preceq g_4$ by deleting the variable y.

iii. There are $v_1 \in V_1$, $v_2 \notin V_1$ and graphs g_5, g_6 such that $succ_1(v_1) = \lambda_1(x)$, $g_6 = ((g_1 \oplus v_2).succ)\,[v_2 \leftarrow succ_1(v_1)]$, $g_5 = (g_6.succ)\,[v_1 \leftarrow v_2]$, and $g_4 = (g_5.\lambda)\,[y \leftarrow v_2]$. $g_1 \preceq g_4$ by deleting the variable y and eliminating v_2 by contraction.

We show $g_4 \preceq g_3$ by considering the two cases $succ_4(\lambda_4(y)) = \lambda_4(x)$ and $succ_4(\lambda_4(y)) = \perp$ when defining g_3 in $g_3 = g_4 \boxplus (y \to x)$. Observe that these are the only possible values for $succ_4(\lambda_4(y))$ according to the definition of g_4. In the first case, we get $g_4 = g_3$. The second case gives $g_3 = (g_4.succ)\,[\lambda_4(y) \leftarrow \lambda_4(x)]$, and hence $g_4 \preceq g_3$ by edge deletion.

(d) $\lambda_1(x) = \perp$, $\lambda_2(y) \neq \perp$, $succ_1(\lambda_1(y)) \neq \perp$, and $g_2 = g_1$. We define $g_3 = (g_2.\lambda)\,[x \leftarrow succ_2(\lambda_2(y))]$. We have $g_1 \preceq g_3$ by deleting variable x.

(e) $\lambda_1(x) = \perp$, $\lambda_1(y) \neq \perp$, $succ_1(\lambda_1(y)) = \perp$, $\lambda_1(y) \neq \lambda_1(null)$ and g_2 in the set $g_1 \boxplus (y \to)$. Notice that $succ_2(\lambda_2(y)) \neq \perp$. We define $g_3 = g_2 \oplus_{y\to} x$. $g_2 \preceq g_3$ holds by deletion of variable x. We show $g_1 \preceq g_2$ by considering the three subcases corresponding to the definition of g_2 in the set $g_1 \boxplus (y \to)$:

   i. There is a $v \notin V_1$ and a graph g_2 such that $g_2 = (g_4.succ)\,[\lambda_4(y) \leftarrow v]$ and $g_4 = g_1 \oplus v$. We have $g_1 \preceq g_4$ by deleting v, while $g_4 \preceq g_2$ by edge deletion.

   ii. There is a $v \in V$ such that $g_2 = (g_1.succ)\,[\lambda_1(y) \leftarrow v]$. $g_1 \preceq g_2$ holds by edge deletion.

   iii. There are $v_1 \in V_1$, $v_2 \notin V_1$ and graphs g_4, g_5 such that $g_5 = ((g_1 \oplus v_2).succ)\,[v_2 \leftarrow succ_1(v_1)]$, $g_4 = (g_5.succ)\,[v_1 \leftarrow v_2]$, and $g_3 = (g_4.succ)\,[\lambda(y) \leftarrow v_2]$. $g_1 \preceq g_5$ holds by deleting an edge and by eliminating v_2 with contraction.

(f) $\lambda_1(x) = \perp$, $\lambda_1(y) = \perp$ and there are g, g', g'' such that $g \in g_1 \oplus x$, $g' \in g' \oplus_{x\leftarrow} y$, $g'' = g' \boxplus (y \to x)$ and $g_2 = g'' \ominus x$. We define $g_3 = g''$. $g_1 \preceq g_3$ by similar arguments to 1(d)i-1(d)ii-1(d)iii (in $g' \in g_1 \oplus x$) and to 4c (in $g' \in g' \oplus_{x\leftarrow} y$ and $g'' = g' \boxplus (y \to x)$). Also, $g_2 \xrightarrow{a} g_3$ because $\lambda_3(x) \neq \perp$, $\lambda_3(y) \neq \perp$, $succ_3(\lambda_3(y)) \neq \perp$, $succ(\lambda_3(y)) = \lambda_3(x)$ and $g_2 = g_3 \ominus x$.

5. a is of the form $x.next := y$. For each one of the six subcases corresponding to the definition of $g_1 \overset{a}{\rightsquigarrow} g_2$, we find a graph g_3 such that $g_2 \xrightarrow{a} g_3$ and $g_1 \preceq g_3$:

(a) $\lambda_1(x) \neq \perp$, $\lambda_1(y) \neq \perp$, $succ_1(\lambda_1(x)) \neq \perp$, $succ_1(\lambda_1(x)) = \lambda_1(y)$ and $g_2 = g_1 \boxminus (x \to)$. Define $g_3 = g_1$. Observe $g_3 = (g_2.succ)\,[\lambda_2(x) \leftarrow \lambda_2(y)]$, meaning $g_2 \xrightarrow{a} g_3$.

(b) $\lambda_1(x) \neq \perp$, $succ_1(\lambda_1(x)) \neq \perp$, $\lambda_1(y) = \perp$ and there is a $g_3 = g_1 \oplus_{x\to} y$ such that $g_2 = g_3 \boxminus (x \to)$. Observe that $g_1 \preceq g_3$ by deletion of variable y.

(c) $\lambda_1(x) \neq \perp$, $succ_1(\lambda_1(x)) = \perp$, $\lambda_1(y) \neq \perp$, $\lambda_1(x) \neq \lambda_1(null)$ and $g_2 = g_1$. We define $g_3 = (g_2.succ)\,[\lambda_2(x) \leftarrow \lambda_2(y)]$. Also, observe $g_1 \preceq g_3$ by edge deletion.

(d) $\lambda_1(x) \neq \perp$, $succ_1(\lambda_1(x)) = \perp$, $\lambda_1(y) = \perp$, $\lambda_1(x) \neq \lambda_1(null)$, and g_2 in the set $g_1 \oplus y$. Observe $\lambda_2(y) \neq \perp$. We define $g_3 = (g_2.succ)\,[\lambda_2(x) \leftarrow \lambda_2(y)]$. Observe $g_2 \preceq g_3$ holds by edge deletion, and $g_1 \preceq g_2$ by deletion of variable y.

(e) $\lambda_1(x) = \perp$, $\lambda_1(y) \neq \perp$, and there is a g_3 such that $g_2 = g_3 \boxminus (x \to)$ where $g_3 \in (g_1 \oplus_{y\leftarrow} x)$. We have $g_3 = (g_2.succ)\,[\lambda_2(x) \leftarrow \lambda_2(y)]$, meaning

$g_2 \xrightarrow{a} g_3$. Furthermore, $g_1 \preceq g_3$ by arguments similar to those in (4(c)i), (4(c)ii) and (4(c)iii).

(f) $\lambda_1(x) = \bot$, $\lambda_1(y) = \bot$, and there graphs g_3, g_4 such that $g_4 \in g_1 \oplus y$, $g_3 \in g_4 \oplus_{y\leftarrow} x$, and $g_2 = g_3 \boxminus (x \rightarrow)$. We use similar arguments to those in (5e) to deduce $g_2 \xrightarrow{a} g_3$ and $g_1 \preceq g_3$.

Assume $g_1 \xrightarrow{a}_A g_2$. This implies the existence of a g_3 such that $g_3 \xrightarrow{a} g_2$ and $g_3 \preceq g_1$. Showing that $g_2 \overset{a}{\rightsquigarrow} g_3$ is sufficient for proving $g_2 \rightsquigarrow g_1\!\downarrow$. We let in the following $g_i = (V_i, succ_i, \lambda_i)$ for $i \geq 0$; and consider five cases depending on the type of a:

1. a is of the form $x = y$. By definition: $\lambda_3(x) \neq \bot$, $\lambda_3(y) \neq \bot$, $\lambda_3(x) = \lambda_3(y)$ and $g_3 = g_2$. We have $g_2 \overset{a}{\rightsquigarrow} g_3$ by case (1a) in Section 5.
2. a is of the form $x \neq y$. By definition: $\lambda_3(x) \neq \bot$, $\lambda_3(y) \neq \bot$, $\lambda_3(x) \neq \lambda_3(y)$ and $g_3 = g_2$. We have $g_2 \overset{a}{\rightsquigarrow} g_3$ by case (2a) in Section 5.
3. a is of the form $x := y$. By definition: $\lambda_3(x) \neq \bot$, $\lambda_3(y) \neq \bot$, $\lambda_3(x) = \lambda_3(y)$ and $g_3 = g_2 \ominus x$. We have $g_2 \overset{a}{\rightsquigarrow} g_3$ by case (3a) in Section 5.
4. a is of the form $x := y.next$. By definition: $\lambda_3(x) \neq \bot$, $\lambda_3(y) \neq \bot$, $succ_3(\lambda_3(y)) \neq \bot$, $succ_3(\lambda_3(y)) = \lambda_3(x)$ and $g_3 = g_2 \ominus x$. We have $g_2 \overset{a}{\rightsquigarrow} g_3$ by case (4a) in Section 5.
5. a is of the form $x.next := y$. By definition: $\lambda_3(x) \neq \bot$, $succ_3(\lambda_3(x)) \neq \bot$, $\lambda_3(y) \neq \bot$, $succ_3(\lambda_3(x)) = \lambda_3(y)$ and $g_3 = g_3 \boxminus (x \rightarrow)$. We have $g_2 \overset{a}{\rightsquigarrow} g_3$ by case (5a) in Section 5.

B Proofs of Lemmas - Section 7

We show termination of the reachability algorithm in several steps. First, we recall basic concepts from the theory of well quasi-orderings. Then, we identify particular properties of our graphs which are implied by the fact that each node has at most one successor. More precisely, we will define a certain measure on graphs, which we call the *degree* of the graph, and show that the graphs which are generated during the reachability algorithm have degrees which are bounded by some natural number. This will lead to a new entailment ordering on graphs, defined in terms of particular encodings of the graph structures. We show that the new ordering is a WQO, and that it is stronger than the ordering $\preceq$ used in the reachability algorithm; leading to the termination proof. Let $\mathbb{N}^{>0}$ denote the set of positive integers.

Well Quasi-Orderings 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: $a_1 a_2 \cdots a_m \preceq^* b_1 b_2 \cdots b_n$ if there is an injection h from the set from $\{1, 2, \ldots, m\}$ to the set $\{1, 2, \ldots, n\}$ such that $i < j$ implies $h(i) < h(j)$ and $a_i \preceq b_{h(i)}$ for all $i, j : 1 \leq i, j \leq m$. In our proofs, we will use the following property of WQOs (see e.g.[2]).

Lemma 3. *If $\preceq$ is a WQO on A then $\preceq^*$ is a WQO on A^*.*

In other words, WQOs extend to finite words under the mentioned ordering. As special cases, WQOs also extend to multisets, vectors of length k for a given $k \in \mathbb{N}^{>0}$, etc. For instance, the lemma implies that vectors of multisets of natural numbers are WQO.

Blocks In this paragraph, we identify some particular patterns in the class of graphs we consider in this paper. Consider a graph $g = (V, \mathit{succ}, \lambda)$. A vertex v is said to be a *leaf* if $\mathit{succ}^{-1}(v) = \emptyset$; and a *root* if $\mathit{succ}(v) = \bot$. A graph is said to be a *tree* if it contains exactly one root, and it is called a *star* if it contains no root.

A graph $g' = (V', \mathit{succ}', \lambda')$ is said to be a *block* in g if V' is a maximal subset of V containing a vertex v such that, for all $v' \in V'$, it is the case that $(v', v) \in \mathit{succ}^*$. In other words, there is path along the edges of g from v' to v. Furthermore, succ' and λ' are the restrictions of succ and λ to V', respectively. In other words, $\mathit{succ}'(v) = v'$ iff $v' \in V'$ and $\mathit{succ}(v) = v'$; and $\lambda(x) = v$ iff $v \in V'$ and $\lambda(x) = v$. Each graph can obviously be uniquely partitioned into a finite set of blocks. Furthermore, in our graphs, since each vertex has at most one successor, it follows that each block in g is either a tree or a star.

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 as $\mathit{deg}\,(v) := d_2 - d_1$ where d_2 is the number of unguarded vertices and d_1 is the number of roots 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.

The proof of the following lemma exploits the fact that each block is either a tree or a star.

Lemma 4. *For any $k \in \mathbb{N}^{>0}$, there are only finitely many compact graphs g such that $\mathit{deg}\,(g) \leq k$ and such that g does not contain any blocks with degree 0.*

Proof. Suppose that there is a $k \in \mathbb{N}^{>0}$ such that $\mathit{deg}\,(g) \leq k$ for each $g \in G$. First we show the claim for graphs which consist of a single block. Consider such a graph g. Since there is at most one root in g, the number of leafs is bounded by $\mathit{deg}\,(g) + |X|$. Furthermore, there is at most $|X|$ internal nodes (non-leafs) which have only one child (the nodes labeled by some variable). The claim follows immediately.

Now, suppose that g contains several blocks. Since each block has a positive degree, it follows that the number of blocks is bounded by k, and each of them has a degree which is less than k. This means that the size of the set is bounded by at most k multiplied by the number of blocks of degree at most k (which is bounded as explained above).

In particular, Lemma 4 implies that if we take any (infinite) set of compact graphs and delete from each graph the blocks which are of degree 0, then we will be left with only finitely many graphs. This will be reflected in our termination proof, where we will distinguish between blocks of degree 0 and blocks with positive degrees. We define $g_{=0}$ to be the graph we get from g by keeping only the blocks with degree 0. Analogously, we define $g_{>0}$ to be the graph we get from g by keeping only the blocks with positive degrees.

Encodings and Signatures An *encoding* is a tuple $e = (V, \mathit{succ}, \lambda, \#)$ where $g = (V, \mathit{succ}, \lambda)$ is a compact graph, and $\# : V \times V \to \mathbb{N}^{>0}$ is a partial mapping such that $\#(v_1, v_2) \neq \bot$ iff $v_2 = \mathit{succ}(v_1)$. In other words, $\#$ associates a positive integer to each edge in g. We call g the *signature* of e and denote it by $\mathit{sig}\,(e)$.

Fix a graph $g = (V, \mathit{succ}, \lambda)$. For non-simple vertices $v, v' \in V$, we say that v' is the *target* of v if there are vertices $v_0, v_1, \ldots, v_n$ with $n \geq 1$ such that $v_0 = v$, $v_n = v'$, v_i is simple for all $i : 1 \leq i < n$, and $v_i = \mathit{succ}(v_{i-1})$ for all $i : 1 \leq i \leq n$. In other words, there is a path of simple nodes leading from v to v'. In such a case we define the *distance* between v and v' by $\mathit{dist}(v, v') := n$. Notice that the target v' of vertex v is unique if it exists. We define $\mathit{enc}\,(g)$ to be the encoding $(V', \mathit{succ}', \lambda', \#)$, where V' is the set of non-simple nodes in V, $\mathit{succ}(v) = v'$ if v' is the target of v in g, $\lambda' = \lambda$, and $\#(v, v') = \mathit{dist}(v, v')$. We define $\mathit{sig}\,(g) := \mathit{sig}\,(\mathit{enc}\,(g))$. We will extend graph concepts to encodings by interpreting them on their signature. For instance, when we say that encoding e has degree k then we mean that g (i.e., $\mathit{sig}\,(e)$) has degree k; and write $e_{=0}$ to denote $\mathit{enc}\,(g_{=0})$, etc.

We will define a WQO $\sqsubseteq$ on encodings. The definition of $\sqsubseteq$ will be derived from two relations, namely $\sqsubseteq^1$ and $\sqsubseteq^2$. The relation $\sqsubseteq^1$ is based on graph inclusion and will be applied on blocks with degree 0; while the relation $\sqsubseteq^2$ is based on graph isomorphism and will be applied to blocks with positive degrees.

Consider compact graphs $g = (V, \mathit{succ}, \lambda)$ and $g' = (V', \mathit{succ}', \lambda')$. For an injection $h : V \to V'$, we say that g is *included* in g' with respect to h, denoted $g \subseteq_h g'$ if, for all vertices $v, v' \in V$ and variables $x \in X$, it is the case that (i) if $v' = \mathit{succ}(v)$ then $h(v') = \mathit{succ}'(h(v))$; and (ii) if $\lambda(x) = v$ then $\lambda'(x) = h(v)$. For encodings e, e', we write $e \sqsubseteq_h^1 e'$ if $\mathit{sig}\,(e) \subseteq_h \mathit{sig}\,(e')$ and for all vertices v, v', if $\#(v, v') \neq \bot$ then $\#(v, v') \leq \#(h(v), h(v'))$.

For a bijection $h : V \to V'$, we say that g_1 and g_2 are *isomorphic* with respect to h, denoted $g \sim_h g'$ if, for all vertices $v, v' \in V$ and variables $x \in X$, it is the case that (i) $v' = \mathit{succ}(v)$ iff $h(v') = \mathit{succ}'(h(v))$; and (ii) $\lambda(x) = v$ iff $\lambda'(x) = h(v)$. We say that g and g' are isomorphic, denoted $g \sim g'$ if $g \sim_h g'$ for some h; and For encodings e, e', we write $e \sqsubseteq_h^2 e'$ if $\mathit{sig}\,(e) \sim_h \mathit{sig}\,(e')$ and for all vertices v, v', if $\#(v, v') \neq \bot$ then $\#(v, v') \leq \#(h(v), h(v'))$.

We use $e \sqsubseteq_h e'$ to denote that both $e_{=0} \sqsubseteq_{h_1}^1 e'_{=0}$, and $e_{>0} \sqsubseteq_{h_2}^2 e'_{>0}$ where h_1 and h_2 are the restrictions of h to the sets of vertices in $e_{=0}$ and $e_{>0}$ respectively. We write $e \sqsubseteq e'$ to denote that $e \sqsubseteq_h e'$ for some h. For graphs g, g', we write $g \sqsubseteq g'$ to denote that $\mathit{enc}\,(g) \sqsubseteq \mathit{enc}\,(g')$.

Lemma 5. *The ordering $\sqsubseteq$ is a WQO on any set G of graphs with degrees bounded by some $k \in \mathbb{N}^{>0}$.*

Proof. First, we show that $\sqsubseteq^1$ is a WQO on encodings of degree 0. A block in a compact graph g (underlying such an encoding) with degree 0 is of one of four forms (in the first three cases, the block does not contain any vertex labeled by a variable): (i) a single isolated vertex; or (ii) a single vertex with a self-loop (an edge starting from and ending at the vertex); or (iii) two vertices with an edge between them; or (iv) the block contains at least one vertex labeled by a variable. Since we have only finitely many variables, the set of possible blocks which can occur in g is finite (although there is no bound on the number of blocks which may occur in g). This means that each encoding with degree 0 can be considered as a vector of multisets of natural numbers: each entry of the vector corresponds to the natural numbers which appear on the edges of blocks of a certain type. For instance, if we consider blocks which are self-loops and without variables, then the corresponding entry will record the natural numbers which appear on the loop. Furthermore, we need multisets since there is no bound on the number of blocks of that particular type which may occur in g, and therefore there is no bound either on the number of times the same natural number may occur in the different blocks of that type. The well quasi-ordering of $\sqsubseteq^1$ on encodings of degree 0, then follows from Lemma 3.

Next, we show that $\sqsubseteq^2$ is a WQO on encodings of positive degrees which are bounded by some k. Consider such a set of encodings, and the corresponding set of underlying compact graphs. From Lemma 4, we know that there are only finitely many such compact graphs. Therefore, the set of encodings can be viewed as vectors of natural numbers: each entry of a vector is indexed by one edge in one of the (finitely many) compact graphs. The value of an entry is given by the natural number on the corresponding edge. The WQO of $\sqsubseteq^2$ then follows from Lemma 3.

Finally, we show that $\sqsubseteq$ is a WQO on the set of encodings with degrees bounded by k. Each underlying compact graph can be considered as consisting of two parts; namely the blocks with degree 0, and those with positive degrees (but with degrees which are bounded by k). The encoding of such a graph can be described a pair, where the first element is a vector of multisets of natural numbers, and the second part is a multiset of natural numbers (as described above). Since both parts are WQOs, it follows by Lemma 3 that the set of pairs of such elements is also a WQO.

The next lemma shows the relation between the orderings $\sqsubseteq$ and $\preceq$. For graphs g, g', suppose that $g \sqsubseteq g'$. Then, we can derive g from g' by applying a finite sequence of variable, vertex, and edge deletions on blocks with degree 0, and applying a finite sequence of contraction operations on blocks with positive degrees. This gives the following lemma.

Lemma 6. *For graphs g_1, g_2, if $g_1 \sqsubseteq g_1$ then $g_1 \preceq g_2$.*

The following lemma implies that the degree of the configurations generated during the reachability algorithm is bounded by the degree of the configurations in the set C_F. The proof of the lemma can be carried out by the fact that no unguarded vertices are introduced in the computation of $\rightsquigarrow$.

Lemma 7. *For graphs g_1, g_2, if $g_1 \rightsquigarrow g_2$ then $deg\,(g_2) \leq deg\,(g_1)$.*

Proof. Suppose that $g_1 \overset{a}{\rightsquigarrow} g_2$. We inspect the cases in the definition of $\rightsquigarrow$ one by one, and verify that no unguarded vertices are introduced in the derivation of g_2 from g_1. We observe that the degree can be increased by removing a variable from a leaf (which is not pointed by any other variable), or by adding an edge leading to a node which is not unguarded.

1. a is of the form $x = y$. In all cases, no variables or edges are deleted.
2. a is of the form $x := y$. We inspect the different subcases:
   (a) x is removed; however y still points to the vertex.
   (b) x is removed; however y is made to point to the vertex.
   (c) No variables or edges are removed.
   (d) No variables or edges are removed.
3. a is of the form $x \neq y$. In all the case, no variables or edges are deleted.
4. a is of the form $x := y.next$. We inspect the different subcases:
   (a) x is removed; however the vertex is still the successor of the vertex pointed to by y.
   (b) x is removed; however the vertex is now pointed to by the successor of vertex pointed to by y.
   (c) x is removed; however the vertex is now pointed to by the successor of vertex pointed to by y.
   (d) No variables or edges are removed.
   (e) No variables or edges are removed.
   (f) No variables or edges are removed.
5. a is of the form $x.next := y$ We inspect the different subcases:
   (a) The edge from the vertex pointed by $x$to the vertex pointed by $y$is removed; however both vertices are still pointed to by $x$and y.
   (b) The edge from the vertex pointed by $x$to its successor is removed; however, the successor is now pointed to by y.
   (c) No variables or edges are removed.
   (d) No variables or edges are removed.
   (e) No variables or edges are removed.
   (f) No variables or edges are removed.

Now, we are ready to prove Theorem 2. Suppose that that algorithm does not terminate. Then, during the course of the algorithm, we add to the set `ToExplore` an infinite sequence $c_0, c_1, c_2, \ldots$ of configurations, where c_i is of the form (q, g_i), and and where $c_i \not\preceq c_j$ for all i, j with $i < j$. From Lemma 7, we know that that the set $\{g_0, g_1, g_2, \ldots\}$ has bounded degree, and therefore by Lemma 5, it follows that there are i, j such that $i < j$ $e(g_i) \sqsubseteq^2 e(g_j)$. From Lemma 6 it follows that $g_i \preceq g_j$, and hence $c_i \preceq c_j$ which is a contradiction. This gives Theorem 2.