From pointer systems to counter systems using shape analysis

Sébastien Bardin\textsuperscript{2} Alain Finkel\textsuperscript{2} Étienne Lozes\textsuperscript{2}

\textit{LSV, ENS Cachan & CNRS UMR 8643}\newline
\textit{61 av. Pdt Wilson, 94235 Cachan Cedex France}

Arnaud Sangnier\textsuperscript{2}

\textit{EDF R\&D, LSV, ENS Cachan & CNRS UMR 8643}\newline
\textit{61 av. Pdt Wilson, 94235 Cachan Cedex France}

\textbf{Abstract}

We aim at checking safety properties on systems manipulating dynamic linked lists. First we prove that every pointer system is bisimilar to an effectively constructible counter system. We then deduce a two-step analysis procedure. We first build an over-approximation of the reachability set of the pointer system. If this over-approximation is too coarse to conclude, we then extract from it a bisimilar counter system which is analyzed via efficient symbolic techniques developed for general counter systems.

\textit{Key words:} dynamic allocation, automatic verification, counter system, pointer system, shape analysis

\section{Introduction}

\textbf{Context.} The model checking techniques for infinite-state systems are now an active research area. These techniques allow to verify different kinds of models like pushdown systems, channel systems, counter systems, pointer systems and many other models like rewriting systems. For some of these models, there exist today tools for verifying such systems: Moped (for pushdown systems), TREX\textsuperscript{[1]} (for channel systems), FAST\textsuperscript{[6]}, LASH\textsuperscript{[16]}, TREX\textsuperscript{[1]}, BRAIN\textsuperscript{[17]} and ALV (for counter systems), TVLA\textsuperscript{[13]} and PALE\textsuperscript{[14]} (for pointer systems).

\textsuperscript{1} This work has been partially supported by contract 4300038040 between EDF/LSV and by the AVERILES project.
\textsuperscript{2} \texttt{[bardin|finkel|lozes|sangnier]@lsv.ens-cachan.fr}

\textit{This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science URL: www.elsevier.nl/locate/entcs}
The model of counter systems is very expressive, it is well-known and it has been used (alone or with some extensions) for modelling a lot of case studies (see for instance the recent verification of the embedded protocol TTP by using the tool FAST[6]).

The problem. Explicit memory management is a common source of errors in programs. Mechanisms of dynamic allocation are naturally available in imperative languages such as C, and functional languages often include such facilities for efficiency purpose. Explicit memory management is typically exploited in industrial, real-time or embedded systems. This efficiency has naturally a price: in such a programming style, the safety of the operations of allocation, deallocation, and dereferencing relies on a sound conception of the program. As so, many unsafe behaviors can happen, such as memory violation, memory leak, unwilled aliasing, etc.

There is currently a great interest for defining automatic methods for the verification of these systems with pointers. However, there exists different models of programs with memory which are not compared and, in general, they are not supported by model checkers: as a matter of fact, TLVA is an analyser using abstract interpretation and PALE is a prover which needs the help of the user.

We would like to be able to automatically verify the qualitative and quantitative properties. Shape analysis, first introduced by Sagiv and Lev-Ami [12,13,15], opened a way to a qualitative analysis of the memory manipulated by the program. Typical questions answered by this method are list-shape preservation, aliasing detection, etc.

The quantitative analysis [5,11] based on Presburger arithmetics, recently extended by more powerful decidable arithmetics [9] tries to answer about the length equality of two lists, the preservation of the memory size, etc.

In this paper we address the problem of verification of both types of properties for a class of (non-interprocedural) programs manipulating list structures, possibly with circularity and sharing, called here pointer systems [5]. We abstract from standard data structures: pointers are the only data type we consider, and we manipulate abstract heap addresses without pointer arithmetics.

Our contribution is to propose an automatic translation of any pointer system \( S_P \) into a bisimilar counter system \( S_C \). This translation is based on a notion of memory shape. This translation is more intuitive than the other translations in logic formula. Another advantage of our translation is that the input formalism can be easily extended to pointer systems with counters. We define a general framework for both quantitative and qualitative analysis. We develop a two-step analysis procedure of pointer systems based on the counter system generated. (1) A static analysis of the counter system: this analysis gives an over-approximation of the memory shapes the system may reach, allowing to derive safety properties efficiently in some cases. (2) If the first analysis is not tight enough, then a reachability analysis is performed at
the counter system level to refine the previous approximation. This analysis relies on efficient acceleration techniques developed for counters [3,4], and can discover automatically complex arithmetic relations in the memory heap.

**Related Works.** Some tools and techniques checking reachability properties on pointer programs have been already proposed. We list here the main works we are aware of:

- **PALE**[14] translates both the program and the memory heap in a decidable monadic logic. The tool can be used for lists as well as trees, but it requires annotating the program with loop invariants.

- **TVLA** [12,13,15] is based on a finite abstraction w.r.t. predicates typically stating that a memory cell points to another one. The approach works for lists and trees, but the user has to provide predicates controlling the preciseness of the abstraction.

- In [8] the authors propose technics based on abstract regular model checking: the structure of the memory heap is represented by regular expressions, and an *abstract, check and refine* procedure is designed to check properties. The authors present encouraging benchmarks.

- **Smallfoot**[7] is an automatic verification tool that checks separation logic specifications of sequential and concurrent programs that manipulate recursive dynamically-allocated (linked) data structures.

- In [9,10] the authors introduce a logic geared towards quantitative shape analysis.

- In [5] we introduced *pointer systems* to model the class of programs we are interested in. We propose a semi-algorithmic method based on symbolic representations of infinite sets of configurations of the pointer system, called *symbolic memory states* (SMS). However such an iterative fixpoint computation must be equipped with some adequate widening (or acceleration) techniques in order to help convergence. This adequate widening was missing.

- In [11] (not published), Finkel and Nowak proposed a translation of pointer systems into counter systems. However this translation leads to counter systems labeled by relations instead of functions. As a matter of fact, it was not feasible to re-use verification techniques developed for traditional counter systems.

## 2 Preliminaries

We present in this section our model of programs (pointer system), our model of the memory heap (memory graph) (see [5]) and counter systems. In the following we consider given two finite, disjoint sets, the set $V$ of pointer variables and the set $K$ of counter variables. To avoid ambiguity, we range over pointer variables with $x, y, ...$ and over counter variables with $k_1, k_2, ...$
2.1 Pointer and counter systems

A systems model is a tuple $\mathcal{M} = (\mathcal{D}, \mathcal{G}, \mathcal{A})$ where $\mathcal{D}$ is an infinite set of data, $\mathcal{G} \subseteq 2^\mathcal{D}$ is set of guards, and $\mathcal{A} \subseteq \mathcal{D}^2$ set of actions. We usually note $d \models g$ for $d \in g$ and $\text{post}(a, d)$ for $a(d)$. In the frame of a systems model, we may consider a particular system:

**Definition 2.1 (System).** A system $S$ in $\mathcal{M} = (\mathcal{D}, \mathcal{G}, \mathcal{A})$ is a pair $S = (Q, \delta)$, where $Q$ is a finite set whose elements are called control states, and $\delta$ is a finite subset of $Q \times \mathcal{G} \times \mathcal{A} \times Q$, called the transition relation.

As usual, we often note $q \xrightarrow{a} q'$ a given transition $(q, g, a, q')$. The semantics of a system $S$ is given by means of its associated transition system $TS(S) = (Q \times \mathcal{D}, \rightarrow)$ where $\rightarrow \subseteq (Q \times \mathcal{D})^2$ is defined as:

$$(q, d) \rightarrow (q', d') \iff \exists g, a \text{ such that } (q, g, a, q') \in \delta, d \models g \text{ and } \text{post}(a, d) = d'$$

We note $\text{Reach}_S(q, d)$ the set $\{(q', d') | (q, d) \rightarrow^* (q', d')\}$ where $\rightarrow^*$ denotes the reflexive and transitive closure of $\rightarrow$.

A standard model of systems is the counter systems model $\mathcal{M}_c = (\mathbb{N}^K, \Phi, \mathcal{F})$ where $\mathbb{N}^K$ is the set of counter valuations, $\Phi$ is the set of Presburger formulas over $K$, and $\mathcal{F}$ is the set of linear functions in $\mathbb{N}^K$. We range over with $\text{val}$ for valuations, $\phi$ for a Presburger formula, and $f$ for a linear function.

**Pointer systems** are systems accessing a heap (in [5] they were called pointer automata). We denote the pointer model $\mathcal{M}_p = (\mathcal{D}_p, \mathcal{G}_p, \mathcal{A}_p)$ where $\mathcal{D}_p$ is the set of memory graphs we will define in the next section, and $\mathcal{G}_p$ and $\mathcal{A}_p$ are the set of guards $g$ and actions $a$, respectively defined by the following grammars:

$$g ::= \text{True} | \text{IsNull}(x) | \neg\text{IsNull}(x)$$
$$a ::= x := E | x.s := E | x := \text{new} | \text{free}(x) | \text{skip}$$

where $E$ is either $\text{null}$, or $x$ or $x.s$.

The formal definition of the semantics of guards and actions is sketched in the next section. Intuitively, $\text{null}$ is the null pointer, $x$ denotes the memory cell $n_x$ pointed to by $x$ and $x.s$ denotes the memory cell pointed to by $n_x$. The actions $\text{free}$ and $\text{new}$ respectively deallocate and allocate memory cells.

**Example.** Figure 1 presents a reverse function written in C and the corresponding pointer system in both textual and graphical representation. This example is taken from [13].

2.2 Memory model and semantics

With assumptions made on the class of programs we want to analyze, the memory heap can be modeled as a finite oriented graph whose nodes are
/* reverse.c */
#include "list.h"

List reverse(List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x->n;
        y->n = t;
        t = NULL;
    }
    return y;
}

Figure 1. A C program reversing a list and an equivalent pointer system.

allocated memory cells. An edge from node \( n_1 \) to node \( n_2 \) indicates that the memory cell \( n_1 \) contains the address of memory cell \( n_2 \) (\( n_1 \) points to \( n_2 \)). Each node is also labeled by the finite set of pointer variables pointing to the cell. Finally the graph contains three special nodes: \( z \) models the null pointer, \( p \) models all the memory cells with illegal address and \( r \) represents an illegal computation (typically, the successor of null is \( r \)). The definition of memory graph directly follows.

**Definition 2.2 (Memory graph).**[5] A memory graph \( MG \) is a tuple \( (N, next, var) \) such that

- \( N \) is a finite set of nodes, containing three distinguished elements \( p, r \) and \( z \).
- \( next \) is a total function from \( N \) to \( N \), called the successor function, such that \( next^{-1}(r) = \{z, p, r\} \).
- \( var : N \rightarrow 2^\mathbb{V} \) is a function such that \( \{var(n)\}_{n \in N} \) forms a partition of \( \mathbb{V} \).

Each node has exactly one successor since a memory cell points either to a valid memory cell, or to null or to an invalid memory cell. Edges are defined by the pairs \( (n, next(n)) \) for \( n \in N \). Figure 2 shows an example of memory graph (empty sets of variables are not written). We will denote \( \mathcal{MG} \) the sets of all memory graphs.

We say that a memory graph has a memory violation if \( var(r) \neq \emptyset \). It has a memory leakage if \( N \) contains at least one node that cannot be accessed from any node labeled with at least one variable. A memory graph is said to be unsafe if it has a memory violation or a memory leakage, otherwise it is said to be safe. As announced in the previous section, it is possible to give a semantic to \( MG \models g \) and \( \text{post}(a, MG) \). However, we skip here the formal definition of these notions. Intuitively, \( \text{post} \) is defined in terms of adding and deleting edges, nodes and labels of memory graphs. For example the instruction \( x := \text{new} \) creates a new node \( n' \), moves the \( x \) label to it, and set \( next(n') \) to \( p \).
Remark 2.3 This behaviour is very closed to the one of C. It is easy to adapt it to various languages and specifications. For example, in Java, the node created by `new` would be linked to `z` rather than `p`.

We precise that we define our semantics in order to detect on-the-fly memory violations and memory leaks: when an unsafe memory graph occurs, the computation stops, because no guard is satisfied by an unsafe memory graph (even `True`).

3 Computing with memory shapes

In this section, we present an abstract view of memory states we call memory shapes. The first section gives a formal definition of this notion, whereas the second section explains how we may define a symbolic computation on these objects.

3.1 Memory shapes

We now introduce memory shapes. Before giving our formal definition, we collect some useful notions on memory graphs. We say that a node $n$ of a memory graph $M G$ is a core node if either the input degree of $n$ (that is the number of incoming edges) is different of 1, or $n$ is labeled by at least one pointer variable, or $n$ is one of the three special nodes. A memory graph is said to be minimal [5] if it contains only core nodes.

Definition 3.1 (Memory shape). A memory shape is:

- either `SegF` or `MemLeak`;
- or a tuple $(N, next, var, K, c)$ where $(N, next, var)$ is a safe, minimal memory graph, $K \subseteq \mathbb{K}$ is a finite set of counter variables and $c : N \setminus \{p, z, r\} \rightarrow K$ is a bijection.

We will denote $\mathcal{MS}$ the set of all memory shapes. We call valued memory shape a pair $(MS, val)$ where $MS$ is a memory shape and $val : K \rightarrow \mathbb{N}^*$ maps each counter in $MS$ to a strictly positive integer. Intuitively, a valued memory shape represents, in a more compact way, a memory graph without memory leak: each $c(n)$ labeling the edge $(n, n')$ represents the succession of $val(c(n))$ edges in the original memory graph. Conversely, any safe memory graph can be represented by an adequate valued memory shape. This gives us a function $\langle \cdot \rangle : \mathcal{MS} \times \mathbb{N}^K \rightarrow \mathcal{MG}$ that associates to a valued memory
shape \((MS, val)\) the corresponding memory graph \(\langle MS, val \rangle\). This function is surjective on safe memory graphs, and in practice we turn it into a bijection adding a counter labeling discipline in memory shapes (but this point is not relevant for this presentation). We recall below two important properties of memory shapes [5,9,2].

**Theorem 3.2** The two following properties hold for memory shapes with a set \(\mathcal{V}\) of pointers variables:

(i) \(|MS| \leq (2 \cdot |\mathcal{V}|)^{3 \cdot |\mathcal{V}|}.

(ii) the number of counters in a memory shape is bounded by \(2 \cdot |\mathcal{V}|\):

**Remark 3.3** For the sake of simplicity, we do not present here a discipline on counter labeling. This point is somehow irrelevant in this presentation, except to gain a bijection as we will mention later on. But it can be noted that the isomorphism of memory shapes up to counter labeling is decidable in linear time, which we hardly use in practice.

### 3.2 Symbolic computation

In this section, we introduce the symbolic computation on memory shapes. It relies on two functions \(\text{TEST}\) and \(\text{POST}\) that lift at the abstract level the concrete computation. We will now give the intuition on how we can take advantage of our notion of memory shape to compute these functions.

The \(\text{TEST}\) function has to decide whether a given memory shape satisfies a guard, that is all underlying memory graphs satisfy it. Definition 3.1 ensures that only the \(z\) node of the memory shape can be mapped to the \(z\) node of the memory graph. So checking a guard \(\text{IsNull}(x)\) simply boils down to check if \(x\) labels the \(z\) node in the memory shape.

The \(\text{POST}\) function has to produce, from a given memory shape and a pointer action, the set \(\{(\phi_i, f_i, MS_i)\}\) of all possible issues: here the \(\phi_i\)'s define mutual excluding conditions on counters that ensure that a unique memory shape will be reached in each case. For every guard, the corresponding memory shape \(MS_i\) is computed, as the linear function \(f_i\) updating counters accordingly.

We skip here the formal definition of this function, and rely on a particular example to clarify our point. The interested reader will find in appendix a more detailed presentation.

**Example 3.4** Consider the memory shape \(MS_1\) represented in figure 3 and the pointer action \(x.s := y\). The \(\text{POST}\) function will return the set \(\{(k_3 = 1, f : (k_1, k_2, k_3) \rightarrow (k_1 + k_2, 0, k_3), MS_2), (k_3 > 1, id, MemLeak)\}\) in which \(MS_2\) denotes the memory shape represented in figure 3. Intuitively, if the counter associated to the pointer variable \(x\) is strictly greater than 1, the action \(x.s := y\) will lead to a memory leak. Otherwise, the shape \(MS_2\) is reached, the edges labeled with \(k_1\) and \(k_2\) are collapsed, and one generates the counter action \(k_1 := k_1 + k_2\).
More formally, the symbolic test function \( \text{TEST} : \mathcal{MS} \times \mathcal{G} \rightarrow \{0, 1\} \) and the symbolic computation function \( \text{POST} : \mathcal{A} \times \mathcal{MS} \rightarrow 2^{\Phi \times F \times \mathcal{MS}} \), are defined such that they enjoy the following properties:

**Proposition 3.5**

(i) the memory shape holds enough information for deciding the guard, that is \( \text{TEST}(\mathcal{MS}, g) = 0 \) iff for all counter valuation \( \text{val} \), \( \langle \mathcal{MS}, \text{val} \rangle \not\models g \), and \( \text{TEST}(\mathcal{MS}, g) = 1 \) iff for all counter valuation \( \text{val} \), \( \langle \mathcal{MS}, \text{val} \rangle \models g \),

(ii) the symbolic computation mimics at the abstract level the concrete computation: if \( \text{POST}(a, \mathcal{MS}) = \{(\phi_i, f_i, \mathcal{MS}_i)\}_{i \in I} \), then the \( g_i \)'s form a partition of \( \mathbb{N}^k \), and for all valuation, if \( \text{val} \models \phi_i \), \( \text{post}(a, \langle \mathcal{MS}, \text{val} \rangle) = \langle \mathcal{MS}_i, f_i(\text{val}) \rangle \).

### 4 Translation of pointer systems into counter systems

In this section, we present a translation from pointer systems to counter systems using the notions of memory shape and symbolic computation we have just defined. We first state the definition of the counter system and its soundness with respect to the pointer system it is built from. In a second time, we present an algorithm to construct effectively this counter system.

#### 4.1 Principle

Equipped with the functions \( \text{TEST} \) and \( \text{POST} \) previously introduced, the translation of a pointer system \( (Q_p, \delta_p) \) into a counter system \( (Q_c, \delta_c) \) is defined by \( Q_c = Q_p \times \mathcal{MS} \) and \( \delta_c \) is defined as follows:

\[
\bigcup_{q_p \in Q_p} \bigcup_{q' \in \delta_p, \mathcal{MS} : \text{TEST}(\mathcal{MS}, g) = 1} \left\{ \langle g, \mathcal{MS} \rangle \xrightarrow{g_i a_i} \langle q', \mathcal{MS}_i \rangle \right\} \quad \text{with } \text{POST}(a, \mathcal{MS}) = \{(g_i, a_i, \mathcal{MS}_i)\}_{i \in I}
\]

Note that \( \delta_c \) is a finite part of \( Q_c \times \Phi \times F \times Q_c \), hence \( (Q_c, \delta) \) actually defines a counter system.
Theorem 4.1 (Soundness of translation) Given a pointer system \((Q_p, \delta_p)\), and \((Q_c, \delta_c)\) the counter system defined above, there is a bisimulation between both underlying transitions systems.

According to Proposition 3.5, the relation

\[ R = \left\{ ((q, (MS, val)), ((q, MS), val)) \mid q \in Q_p, MS \in MS, val \in \mathbb{N}^K \right\} \]

is a bisimulation between both transition systems.

4.2 Translation algorithm

The algorithm works with a set \(\text{Current}\) of the control states \((q, MS)\) known to be reachable and for which outgoing transitions have not been computed yet, and a set \(\text{Treated}\) of reachable control states that have already been treated ensuring single-pass translation.

Algorithm 1 From pointer system to counter system

**Input**: \((Q_p, \delta_p)\) a pointer system;
**Output**: \((Q_c, \delta_c)\) a counter system;

for all \(\forall (q_0, MS_0) \in Q_p \times MS\) do

\(\text{Current} \leftarrow \{(q_0, MS_0)\}\);

\(Q_c \leftarrow \{(q_0, MS_0)\}\);

\(\text{Treated} \leftarrow \emptyset\);

while \(\text{Current} \neq \emptyset\) do

pick \((q, MS)\) in \(\text{Current} \setminus \text{Treated}\);

for all \(q \xrightarrow{g} q' \in \delta_p\) do

if \(MS \models g\) then

for all \((\phi, f, MS') \in \text{POST}(a, MS)\) do

\(\delta_c \leftarrow \delta_c \cup \{(q, MS) \xrightarrow{\phi f} (q', MS')\}\);

\(Q_c \leftarrow Q_c \cup \{(q', MS')\}\);

\(\text{Current} \leftarrow \text{Current} \cup \{(q', MS')\}\);

end for

end if

end for

\(\text{Treated} \leftarrow \text{Treated} \cup \{(q, MS)\}\);

\(\text{Current} \leftarrow \text{Current} \setminus \{(q, MS)\}\)

end while

end for

Theorem 4.2 (Soundness of the algorithm) The algorithm 1 terminates and it computes the counter system defined in section 4.1.

Remark 4.3 In practice, we use this algorithm for a fixed initial control state \(q_0\) and a fixed initial memory shape \(MS_0\). This gives a much smaller counter system \((Q_0, \delta_0)\) such that \(Q_0 \subseteq Q_c\) and \(\delta_0 \subseteq \delta_c\), \((Q_0, \delta_0)\) is the strongly connected component of the control graph of \((Q_c, \delta_c)\) containing \((q_0, MS_0)\). We explain below why this is sufficient for our analysis.
The complexity results of Theorem 3.2 argues for termination and efficiency of the algorithm. Moreover, Theorem 4.1 ensures the soundness of the algorithm.

**Remark 4.4** A first partial translation from pointer systems to counter systems has been made in [11].

## 5 Analysis of pointer systems

### 5.1 Analysis of the counter system

As a consequence of the bisimulation result, the set of memory shapes appearing in the control states of \((Q_0, \delta_0)\) is an over-approximation of the memory shapes actually reachable in the original pointer system. More formally, we define the function \(\text{Abs} : Q_p \times MG \rightarrow Q_p \times MS\) mapping \((q, (MS, val))\) onto \((q, MS)\). Then our over-approximation result can be stated as follows:

**Theorem 5.1** (Over-approximation) For all \(q_0 \in Q_p\), for all \(MS_0 \in MS\), and for all \(val \in \mathbb{N}^K\), \(\text{Abs}(\text{Reach}_{S_p}((q_0, (MS_0, val)))) \subseteq Q_0\).

This result allows us to perform a verification of the pointer system in two passes. The first step consists in the static analysis of the counter system. We directly check on \(Q_0\) if we produced states \((q, \text{SegF})\) or \((q, \text{MemLeak})\). If not, we may directly conclude that the pointer system is safe. Otherwise, we have to check if the unsafe states are actually reachable in the transition system. For this, we rely on the tool FAST[3] and the techniques of acceleration that are implemented in it.

### 5.2 The reverse function example

We illustrate our analysis on the program reversing a list introduced in figure 1 applied on a non empty single-linked list. So we set \(MS_0\) to be the corresponding memory shape and generate the counter system \((Q_0, \delta_0)\). We then observe that no control state contains the memory shape \(\text{SegF}\), but one state contains \(\text{MemLeak}\). Hence after this first step we can conclude that there will not be a memory violation. In order to know if a memory leak might happen, we analyze the counter system using FAST. This second step tells us that the control state containing the memory shape \(\text{MemLeak}\) will not be reached in the pointer system (for any valuation).

## 6 Perspectives

We defined a translation from pointer systems to counter systems and used it to give both a qualitative and quantitative analysis of the pointer system. While doing this, we believe we defined a general framework for both types of analysis, which was not so clearly stated in other works. In order to tackle
more complex examples, we are currently working on a better integration of the quantitative analysis at the translation stage.

Acknowledgements We thank David Nowak and Philippe Schnoebelen for enlighting discussions.

References


A Translation Technical Points: successors of memory shape

We describe here the function denoted POST which produces from a given memory shape and a pointer action a set of triples \{ (φ_i, f_i, M S_i) \} describing all the possible issues; the φ_i’s define mutual excluding conditions on counters that ensure that a unique memory shape will be reached in each case. For every guard, the corresponding memory shape M S_i is computed, as the linear function f_i updating counters accordingly. The eight algorithms that we give correspond to the eight kind of actions which can appear in a pointer automaton. For each function, we suppose that the given memory shape is denoted M S = ⟨ N, next, var, K, c ⟩.

Algorithm 2 Algorithm of POST_x:=null

Let n ∈ N such that x ∈ var(n);
if n = z then
Return \{ (True, id, M S) \}
else
if n = p or var(n) ≠ \{ x \} or deg(n) > 1 then
Return \{ (True, id, M S’) \} where M S’ = ⟨ N, next, var’, K, c ⟩ with var’(z) = var(z) ∪ \{ x \}, var’(n) = var(n) \{ x \} and ∀ m ∈ N \{ z, n ⟩, var’(m) = var(m);
else
if deg(n) = 0 then
Return \{ (True, id, MemLeak) \};
else
Let n’ be the node such that next(n’) = n, k and k’ the counter variables such that c(n) = k and c(n’) = k’;
Return \{ (True, [k’ := k+k’; k = 0], M S”) \} where M S” = ⟨ N\{n⟩, next’, var’, K\{k⟩, c’⟩ with :
• var’(z) = var(z) ∪ \{ x \}, ∀ m ∈ N \{ n, z ⟩, var’(m) = var(m),
• next’(n’) = next(n) and ∀ m ∈ N \{ n, n’ ⟩, next’(m) = next(m),
• ∀ m ∈ N \{ n, z, p, r ⟩, c’(m) = c(m);
end if
end if
end if
Algorithm 3 Algorithm of $\text{POST}_{x=y}$

Let $n, m \in N$ such that $x \in \text{var}(n)$ and $y \in \text{var}(m)$;
if $n = m$ then
  \hspace{1em} \textbf{Return} \{(\text{True}, \text{id}, MS)\};
else
  if $n = z$ or $n = p$ or $\text{var}(n) \neq \{x\}$ or $\text{deg}(n) > 1$ then
    \hspace{1em} \textbf{Return} \{(\text{True}, \text{id}, MS')\} where $MS' = \langle N, \text{next}, \text{var}', K, c \rangle$ with $\text{var}'(m) = \text{var}(m) \cup \{x\}$, $\text{var}'(n) = \text{var}(n) \setminus \{x\}$ and $\forall n' \in N \setminus \{n, m\}$, $\text{var}'(n') = \text{var}(n')$;
  else
    if $\text{deg}(n) = 0$ then
      \hspace{1em} \textbf{Return} \{(\text{True}, \text{id}, \text{MemLeak})\};
    else
      \hspace{2em} Let $n'$ be the node such that $\text{next}(n') = n$, $k$ and $k'$ the counter variables such that $c(n) = k$ and $c(n') = k'$;
      \hspace{1em} \textbf{Return} \{(\text{True}, [k' := k' + k; k := 0], MS')\} where $MS' = \langle N \setminus \{n\}, s', \text{var}', K \setminus \{k\}, c' \rangle$ with :
        \hspace{3em} $\text{var}'(m) = \text{var}(m) \cup \{x\}$, $\forall p \in N \setminus \{n, m\}$, $\text{var}'(p) = \text{var}(p)$,
        \hspace{3em} $\text{next}'(n') = \text{next}(n)$ and $\forall p \in N \setminus \{n, n'\}$, $\text{next}'(p) = \text{next}(p)$,
        \hspace{3em} $\forall p \in N \setminus \{n, z, p, r\}$, $c'(p) = c(p)$;
    end if
  end if
end if

Algorithm 4 Algorithm of $\text{POST}_{x,s=\text{null}}$

Let $n, n' \in N$ such that $x \in \text{var}(n)$ and $n' = \text{next}(n)$;
if $n = p$ or $n = z$ then
  \hspace{1em} \textbf{Return} \{(\text{True}, \text{id}, \text{SegF})\};
else
  if $\text{deg}(n') = 2$ and $\text{var}(n') = \emptyset$ and $n' \notin \{z, p, r\}$ then
    \hspace{2em} Let $n''$ be the node such that $\text{next}(n'') = n'$ and $k$, $k'$, $k''$ be the counter variables such that $c(n) = k$, $c(n') = k'$ and $c(n'') = k''$;
    \hspace{1em} \textbf{Return} \{(k = 1, [k'' := k' + k''; k'' := 0], MS'); (k > 1, \text{id}, \text{MemLeak})\} where $MS' = \langle N \setminus \{n''\}, \text{next}', \text{var}', K \setminus \{k''\}, c' \rangle$ with :
      \hspace{3em} $\forall p \in N \setminus \{n''\}$, $\text{var}'(p) = \text{var}(p)$,
      \hspace{3em} $\text{next}'(n'') = \text{next}(n'')$, $\text{next}'(n) = z$, $\forall p \in N \setminus \{n'', n, n'\}$, $\text{next}'(p) = \text{next}(p)$,
      \hspace{3em} $\forall p \in N \setminus \{n', z, p, r\}$, $c'(p) = c(p)$;
  else
    \hspace{2em} Let $k$ be the counter variable such that $c(n) = k$;
    \hspace{1em} \textbf{Return} \{(k = 1, \text{id}, MS'); (k > 1, \text{id}, \text{MemLeak})\} where $MS' = \langle N, \text{next}', \text{var}, K, c \rangle$ with :
      \hspace{3em} $\text{next}'(n) = z$ and $\forall p \in N \setminus \{n\}$, $\text{next}'(p) = \text{next}(p)$;
  end if
end if
Algorithm 5 Algorithm of POST\(x.s=y\)

Let \(n, m, n' \in N\) such that \(x \in \text{var}(n), y \in \text{var}(m)\) and \(n' = \text{next}(n)\);
if \(n = p\) or \(n = z\) then
   Return \(\{\text{True}, \text{id}, \text{SegF}\}\);
else
   if \(\text{deg}(n') = 2\) and \(\text{var}(n') = \emptyset\) and \(n' \notin \{z, p, r\}\) then
      Let \(n''\) be the node such that \(\text{next}(n'') = n'\) and \(k, k', k''\) be the counter variables such that \(c(n) = k, c(n') = k'\) and \(c(n'') = k''\);
      Return \(\{(k = 1, [k'' = k' + k''; k' = 0], MS'); (k > 1,\text{id}, MemLeak)\}\) where \(MS' = (N \setminus \{n'\}, \text{next}', \text{var}', K \setminus \{k'\}, c')\) with :
      \(\forall p \in N \setminus \{n'\}, \text{var}'(p) = \text{var}(p)\),
      \(\text{next}'(n'') = \text{next}(n'), \text{next}'(n) = m, \forall p \in N \setminus \{n'', n, n'\}, \text{next}'(p) = \text{next}(p)\),
      \(\forall p \in N \setminus \{n', z, p, r\}, c'(p) = c(p)\);
   else
      Let \(k\) be the counter variable such that \(c(n) = k\);
      Return \(\{(k = 1, \text{id}, MS'); (k > 1, \text{id}, MemLeak)\}\) where \(MS' = (N, \text{next}', \text{var}, K, c)\) with :
      \(\text{next}'(n) = m\) and \(\forall p \in N \setminus \{n\}, \text{next}'(p) = \text{next}(p)\);
   end if
end if
Algorithm 6 Algorithm of $\text{POST}_x,s=y,s$

Let $n, m, n', m' \in N$ such that $x \in \text{var}(n)$, $y \in \text{var}(m)$, $n' = \text{next}(n)$ and $m' = \text{next}(m)$;

if $n = p$ or $n = z$ or $m = p$ or $m = z$ then
    Return $\{\text{True}, \text{id}, \text{SegF}\}$;
else
    if $n' = m'$ then
        Return $\{\text{True}, \text{id}, MS\}$
    else
        if $\text{deg}(n') = 2$ and $\text{var}(n') = \emptyset$ and $n' \notin \{z, p, r\}$ then
            Let $n''$ be the node such that $\text{next}(n'') = n'$ and $k', k'', l$ be the counter variables such that $c(n) = k$, $c(n') = k', c(n'') = k''$ and $c(m) = l$;
            Return $\{(k = 1 \land l = 1, [k'' = k' + k'' ; k' = 0], MS') ; (k > 1, \text{id}, \text{MemLeak}) ; (k = 1 \land l > 1, [k'' = k' + k'' ; k' = 1 ; l := l - 1], MS'')\}$ where $MS' = \langle N \setminus \{n'\}, \text{next}', \text{var}', K \setminus \{k'\}, c'\rangle$ with:
            \begin{itemize}
                \item $\forall p \in N \setminus \{n'\}$, $\text{var}'(p) = \text{var}(p)$,
                \item $\text{next}'(n'') = \text{next}(n')$, $\text{next}'(n) = m'$, $\forall p \in N \setminus \{n'', n, n'\}$, $\text{next}'(p) = \text{next}(p)$,
                \item $\forall p \in N \setminus \{n', z, p, r\}$, $c'(p) = c(p)$;
            \end{itemize}
            and $MS'' = \langle N, \text{next}'', \text{var}, K, c \rangle$ with:
            \begin{itemize}
                \item $\text{next}''(m) = n'$, $\text{next}''(n') = m'$ and $\forall p \in N \setminus \{m, n', n''\}$, $\text{next}''(p) = \text{next}(p)$;
            \end{itemize}
        else
            Let $k, l$ be the counter variables such that $c(n) = k$ and $c(m) = l$;
            Return $\{(k = 1 \land l = 1, \text{id}, MS') ; (k > 1, \text{id}, \text{MemLeak}) ; (k = 1 \land l > 1, [\text{new}_k := 1 ; l := l - 1], MS'')\}$ where $MS' = \langle N, \text{next}', \text{var}, K, c \rangle$ with:
            \begin{itemize}
                \item $\text{next}'(n) = m'$ and $\forall p \in N \setminus \{n\}$, $\text{next}'(p) = \text{next}(p)$;
            \end{itemize}
            and $MS'' = \langle N \cup \{\text{new}_n\}, \text{next}'', \text{var}', K \cup \{\text{new}_k\}, c'' \rangle$ with:
            \begin{itemize}
                \item $\text{new}_n \notin N$,
                \item $\text{new}_k \notin K$,
                \item $\text{var}'(\text{new}_n) = \emptyset$ and $\forall p \in N, \text{var}'(p) = \text{var}(p)$,
                \item $\text{next}''(n) = \text{new}_n$, $\text{next}''(m) = \text{new}_n$, $\text{next}''(\text{new}_n) = \text{next}(m)$ and $\forall p \in N \setminus \{n, m\}$, $\text{next}''(p) = \text{next}(p)$,
                \item $c''(\text{new}_n) = l$, $c''(m) = \text{new}_k$ and $\forall p \in N \setminus \{m, z, p, r\}$, $c''(p) = c(p)$.
            \end{itemize}
        end if
    end if
end if
Algorithm 7 Algorithm of $\text{POST}_{x=y.s}$

Let $n, m, m' \in N$ such that $x \in \text{var}(n)$, $y \in \text{var}(m)$ and $m' = \text{next}(m)$;
if $m = p$ or $m = z$ then
  Return $\{(\text{True}, \text{id}, \text{SegF})\}$
else
  Let $l$ be the counter variable such that $c(m) = l$;
  if $n = z$ or $n = p$ or $\text{var}(n) > 1$ or $\text{deg}(n) > 1$ then
    Return $\{(l = 1, \text{id}, MS') ; (l > 1, \text{new}_k := 1 ; l := l - 1, MS'')\}$ where $MS' = (N, \text{next}, \text{var}', K, c)$ with :
    • $\text{var}'(m') := \{x\}$, $\text{var}'(n) = \text{var}(n) \setminus \{x\}$ and $\forall p \in N \setminus \{n, m\}$, $\text{var}'(p) = \text{var}(p)$;
    and $MS'' = (N \cup \{\text{new}_n\}, \text{next}'', \text{var}'', K \cup \{\text{new}_k\}, c'')$ with :
      • $\text{new}_n \notin N$,
      • $\text{new}_k \notin K$,
      • $\text{next}''(\text{new}_n) = m'$, $\text{next}''(m) = \text{new}_n$, $\forall p \in N \setminus \{m\}$, $\text{next}''(p) = \text{next}(p)$,
      • $\text{var}''(\text{new}_n) = \{x\}$, $\text{var}''(n) = \text{var}(n) \setminus \{x\}$ and $\forall p \in N \setminus \{n\}$, $\text{var}''(p) = \text{var}(p)$,
      • $c''(\text{new}_n) = l$, $c''(m) = \text{new}_k$ and $\forall p \in N \setminus \{m, z, p, r\}$, $c''(p) = \text{count}(p)$
  else
    if $\text{deg}(n) = 0$ then
      Return $\{(\text{True}, \text{id}, \text{MemLeak})\}$
    else
      Let $n'$ be the node such that $\text{next}(n') = n$ and let $k'$ be the counter variable such that $c(n') = k'$;
      if $x = y$ then
        if $n = m'$ then
          Return $\{(\text{True}, \text{id}, MS)\}$
        else
          Return $\{(l = 1, [k' := k' + 1 ; l := 0], MS') ; (l > 1, \text{new}_k := k' + 1 ; l := l - 1, MS'')\}$ where $MS' = (N \setminus \{n\}, \text{next}'', \text{var}'', K \setminus \{\text{new}_k\}, c'')$ with :
            • $\text{next}''(n') = m'$ and $\forall p \in N \setminus \{n, n'\}$, $\text{next}''(p) = \text{next}(p)$,
            • $\text{var}''(m') = \text{var}''(n') \cup \{x\}$ and $\forall p \in N \setminus \{n, m'\}$, $\text{var}''(p) = \text{var}(p)$,
            • $\forall p \in N \setminus \{n, z, p, r\}$, $c''(p) = c(p)$
          end if
        else
          Let $k$ be the counter variable such that $c(n) = k$;
          if $n = m'$ then
            Return $\{(\text{True}, [l := 1 ; k := k + l - 1], MS)\}$
          else
            Return $\{(l = 1, [k' := k' + k ; k := 0], MS') ; (l > 1, [k' := k' + k ; k := 1 ; l := l - 1], MS'')\}$ where $MS' = (N \setminus \{n\}, \text{next}'', \text{var}'', K \setminus \{k\}, c'')$ with :
              • $\text{next}''(n') = \text{next}(n)$ and $\forall p \in N \setminus \{n, n'\}$, $\text{next}''(p) = \text{next}(p)$,
              • $\text{var}''(m') = \text{var}''(m) \cup \{x\}$ and $\forall p \in N \setminus \{m', n\}$, $\text{var}''(p) = \text{var}(p)$,
              • $\forall p \in N \setminus \{n, c'(p) = c(p)$;
            and $MS'' = (N, \text{next}'', \text{var}, K, c'')$ with :
              • $\text{next}''(n') = \text{next}(n)$, $\text{next}''(m) = n$, $\text{next}''(n) = m'$ and $\forall p \in N \setminus \{n', m', n\}$, $\text{next}''(p) = \text{next}(p)$,
              • $c''(m) = k$, $c''(n) = l$ and $\forall p \in N \setminus \{n, m, z, p, r\}$, $c''(p) = \text{count}(p)$
            end if
          end if
        end if
      end if
    end if
  end if
end if
Algorithm 8 Algorithm of POST\(_{\text{New}(x)}\)

Let \( n \in N \) such that \( x \in \text{var}(n) \);
if \( n = z \) or \( n = p \) or \( \text{var}(n) \neq \{x\} \) or \( \text{deg}(n) > 1 \) then
   Return \((\{\text{True}, \{\text{new}_k := 1\}, \text{MS}'\})\) where \( \text{MS}' = \langle N \cup \{\text{new}_n\}, \text{next}', \text{var}', K \cup \{\text{new}_k\}, c' \rangle \) with :
   • \( \text{new}_n \notin N \),
   • \( \text{new}_k \notin C \),
   • \( \text{next}'(\text{new}_n) = p \) and \( \forall p \in N, \text{next}'(p) = \text{next}(p) \),
   • \( \text{var}'(\text{new}_n) = \{x\}, \text{var}'(n) = \text{var}(n) \setminus \{x\} \) and \( \forall p \in N \setminus \{n\}, \text{var}'(p) = \text{var}(p) \),
   • \( c'(\text{new}_n) = \text{new}_k \) and \( \forall p \in N \setminus \{z, p, r\}, c'(p) = c(p) \);
else
   if \( \text{deg}(n) = 0 \) then
      Return \((\{\text{True}, \text{id}, \text{MemLeak}\})\);
   else
      Let \( n' \) be the node such that \( \text{next}(n') = n, k \) and \( k' \) the counter variables such that \( c(n) = k \) and \( c(n') = k' \);
      Return \((\{\text{True}, \{k' := k + k; k := 1\}, \text{MS}'\})\) where \( \text{MS}' = \langle N, \text{next}', \text{var}, K, c \rangle \) with :
      • \( \text{next}'(n') = \text{next}(n), \text{next}'(n) = z \) and \( \forall p \in N \setminus \{n, n'\}, \text{next}'(p) = \text{next}(p) \),
   end if
end if

Algorithm 9 Algorithm of POST\(_{\text{Free}(x)}\)

Let \( n \in N \) such that \( x \in \text{var}(n) \);
if \( n = z \) then
   Return \((\{\text{True}, \text{id}, \text{MS}\})\);
else
   if \( n = p \) then
      Return \((\{\text{True}, \text{id}, \text{SegF}\})\);
   else
      Let \( k \) be the counter variable such that \( c(n) = k \);
      Return \((\{k = 1, \{k := 0\}, \text{MS}', (k > 1, \text{id}, \text{MemLeak})\})\) where \( \text{MS}' = \langle N \setminus \{n\}, \text{next}', \text{var}', K \setminus \{k\}, c' \rangle \) with :
      • \( \forall p \in N \setminus \{n\} \) such that \( \text{next}(p) = n, \text{next}'(p) = p \) and \( \forall p \in N \setminus \{n\} \) such that \( \text{next}(p) \neq n, \text{next}'(p) = \text{next}(p) \),
      • \( \text{var}'(p) = \text{var}(p) \cup \text{var}(n) \) and \( \forall p \in N \setminus \{p, n\}, \text{var}'(p) = \text{var}(p) \),
      • \( \forall p \in N \setminus \{n, z, p, r\}, c'(p) = c(p) \);
   end if
end if