Monotonic Abstraction for Programs with Multiply-Linked Structures*

Parosh Aziz Abdulla[1], Jonathan Cederberg[1], Tomáš Vojnar[2]

[1] Uppsala University, Sweden.
[2] FIT, Brno University of Technology, Czech Republic

Abstract. We investigate the use of monotonic abstraction and backward reachability analysis as means of performing shape analysis on programs with multiply pointed structures. By encoding the heap as a vertex- and edge-labeled graph, we can model the low level behaviour exhibited by programs written in the C programming language. Using the notion of *signatures*, which are predicates that define sets of heaps, we can check properties such as absence of null pointer dereference and shape invariants. We report on the results from running a prototype based on the method on several programs such as insertion into and merging of doubly-linked lists.

1 Introduction

Dealing with programs manipulating dynamic pointer-linked data structures is one of the most challenging tasks of automated verification since these data structures are of unbounded size and may have the form of complex graphs. As discussed below, various approaches to automated verification of dynamic pointer-linked data structures are currently studied in the literature. One of these approaches is based on using *monotonic abstraction* and *backward reachability* [4, 2]. This approach has been shown to be very successful in handling systems with complex graph-structured configurations when verifying parameterized systems [3]. However, in the area of verification of programs with dynamic linked data structures, it has so far been applied only to relatively simple singly-linked data structures.

In this paper, we investigate the use of monotonic abstraction and backward reachability for verification of programs dealing with dynamic linked data structures with multiple selectors. In particular, we consider verification of sequential programs written in a subset of the C language including its common control statements as well as its pointer manipulating statements (apart from pointer arithmetics and type casting). For simplicity, we restrict ourselves to data structures with two selectors. This restriction can, however, be easily lifted. We consider verification of safety properties in the form of absence of null and dangling pointer dereferences as well as preservation of shape invariants of the structures being handled.

* The first two authors were supported by the Swedish UPMARC project, the third author was supported by the COST OC10009 project of the Czech Ministry of Education.

We represent heaps in the form of simple vertex- and edge-labeled graphs. As is common in backward verification, our verification technique starts from sets of bad configurations and checks whether some initial configurations are backward reachable from them. For representing sets of bad configurations as well as the sets of configurations backward reachable from them, we use the so-called *signatures* which arise from heap graphs by deleting some of their nodes, edges, or labels. Each signature represents an *upward-closed set* of heaps wrt. a special *pre-order* on heaps and signatures. We show that the considered C pointer manipulating statements can be *approximated* such that one can compute predecessors of sets of heaps represented via signatures wrt. these statements.

We have implemented the proposed approach in a light-weight Java-based prototype and tested it on several programs manipulating doubly-linked lists and trees. The results show that monotonic abstraction and backward reachability can indeed be successfully used for verification of programs with multiply-linked dynamic data structures.

Related work. Several different approaches have been proposed for automated verification of programs with dynamic linked data structures. The most-known approaches include works based on monadic second-order logic on graph types [10], 3-valued predicate logic with transitive closure [14], separation logic [12, 11, 15, 6], other kinds of logics [16, 9], finite tree automata [5, 7], forest automata [8], graph grammars [13], upward-closed sets [4, 2], as well as other formalisms.

As we have already indicated above, our work extends the approach of [4, 2] from singly-linked to multiply-linked heaps. This extension has required a new notion of signatures, a new pre-order on them, as well as new operations manipulating them. Not counting [4, 2], the other existing works are based on other formalisms than the one used here, and they use a forward reachability computation whereas the present paper uses a backward reachability computation. Apart from that, when comparing the approach followed in this work with the other existing approaches, one of the most attractive features of our method is its simplicity. This includes, for instance, a simple specification of undesirable heap shapes in terms of signatures. Each such signature records some bad pattern that should not appear in the heaps, and it is typically quite small (usually with three or fewer nodes). Furthermore, our approach uses local and quite simple reasoning on the graphs in order to compute predecessors of symbolically represented infinite sets of heaps[1]. Moreover, the abstraction used in our approach is rather generic, not specialised for some fixed class of dynamic data structures.

Outline. In Section 2, we give some preliminaries and introduce our model for describing heaps. We present the class of programs we consider in Section 3. In Section 4, we introduce signatures as symbolic representations for infinite sets of configurations. We show how to use signatures for specifying bad heap patterns (that violate safety properties of the considered programs) in Section 5.

[1] Approaches based on separation logic and forest automata also use local updates, but the updates used here are still simpler.

In Section 6, we describe a symbolic backward reachability analysis algorithm for checking safety properties. Next, we report on experiments with the proposed method in Section 7. Finally, we give some conclusions and directions for future work in Section 8.

2 Heaps

Preliminaries. For a partial function $f : A \to B$ and $a \in A$, we write $f(a) = \bot$ to signify that f is undefined at a. We take $f[a \mapsto b]$ to be the function f' such that $f'(a) = b$ and $f'(x) = f(x)$ otherwise. We define the *restriction* of f to A', written $f|_{A'}$, as the function f' such that $f'(a) = f(a)$ if $a \in A'$, and $f'(a) = \bot$ if $a \notin A'$. Given $b \in B$, we write $f^{-1}(b)$ to denote the set $\{a \in A : f(a) = b\}$.

Heaps. We model the dynamically allocated memory, also known as the *heap*, as a labeled graph. The nodes of the graph represent memory cells, and the edges represent how these nodes are linked by their successor pointers. Each edge is labeled by a color, reflecting which of the possibly many successor pointers of its source cell the edge is representing. In this work, we—for simplicity—consider structures with two selectors, denoted as 1 and 2 (instead of, e.g., `next` and `prev` commonly used in doubly-linked lists or `left` and `right` used in trees) only. The results can, however, be generalized to any number of selectors.

To model *null pointers*, we introduce a special node called the *null node*, written #. Null successors are then modeled by making the corresponding edge point to this node. When allocated memory is relinquished by a program, any pointers previously pointing to that memory become *dangling*. Dangling pointers also arise when memory is freshly allocated and not yet initialized. This situation is reflected in our model by the introduction of another special node called the *dangling node*, denoted as $*$. In the same manner as for the null node, a pointer being dangling is modeled by having the corresponding edge point to the dangling node.

Furthermore, we model a program variable by labeling the node that a specific variable is pointing to with the variable in question.

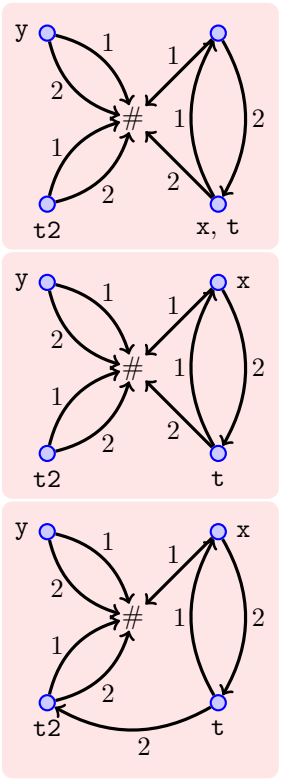


Fig. 1. Heaps

Three examples of heaps can be seen in Figure 1 (we will get back to what they represent in Section 3). To avoid unnecessarily cluttering the pictures, the special node $*$ has been left out. We will adopt the convention of omitting any

of the special nodes $*$ and $\#$ from pictures unless they are labeled or have edges pointing to them.

Assume a finite set of program variables X and a set $C = \{1, 2\}$ of edge colors. Formally, a *heap* is a tuple $(\overline{M}, E, s, t, \tau, \lambda)$ where

- $\overline{M} = M \cup \{\#, *\}$ represents the finite set of allocated memory cells, together with the two special nodes representing the `null` value and the dangling pointer, respectively.
- E is a finite set of edges.
- The *source function* $s : E \to M$ is a total function that gives the source of the edges.
- The *target function* $t : E \to \overline{M}$ is a total function that gives the target of the edges.
- The *type function* $\tau : E \to C$ is a total function that gives the color of the edges.
- $\lambda : X \to \overline{M}$ is a total function that defines the positions of the program variables.

We also require that the heaps obey the following invariant:

$$\forall c \in C \ \forall m \in M : |s^{-1}(m) \cap \tau^{-1}(c)| = 1.$$

The invariant states that among the edges going out from each cell there is exactly one with color 1 and one with color 2. Note that as a consequence of these invariants, each cell has exactly two outgoing edges. Therefore, each heap h induces a function $succ_{h,c} : M \to \overline{M}$ for each $c \in C$, which maps each cell to its c-successor. For $m \in M$, $succ_{h,c}(m)$ is formally defined as the $m' \in \overline{M}$ such that there is an edge $e \in E$ with $s(e) = m$, $t(e) = m'$, and $\tau(e) = c$. This is indeed a function due to the fact that there must be exactly one such edge, according to the specified invariants.

Auxiliary Operations on Heaps. We will now introduce some notation for operations on heaps to be used in the following.

Assume a heap $h = (\overline{M}, E, s, t, \tau, \lambda)$. For $m \in M$, we write $h \ominus m$ to describe the heap h' where m has been deleted together with its two outgoing edges, and any references to m are now dangling references. Formally, $h \ominus m$ is defined as the heap $h' = (\overline{M}', E', s', t', \tau', \lambda')$ where $\overline{M}' = \overline{M} \setminus \{m\}$, $E' = E \setminus s^{-1}(m)$, $s' = s|_{E'}$, $t' : E' \to \overline{M}'$ is a function such that $t'(e) = *$ if $e \in t^{-1}(m)$ and $t'(e) = t(e)$ otherwise, $\tau' = \tau|_{E'}$, and $\lambda'(x) = *$ if $x \in \lambda^{-1}(m)$ and $\lambda'(x) = \lambda(x)$ otherwise. In a similar manner, for $m' \notin M$, we write $h \oplus m'$ to mean the heap where we have added a new cell as well as two new dangling outgoing edges. Formally, $h \oplus m' = (\overline{M}', E', s', t', \tau', \lambda)$ where $\overline{M}' = \overline{M} \cup \{m'\}$, $E' = E \cup \{e_1, e_2\}$, $s' = s[e_1 \mapsto m', e_2 \mapsto m']$, $t' = t[e_1 \mapsto *, e_2 \mapsto *]$ and $\tau' = \tau[e_1 \mapsto 1, e_2 \mapsto 2]$ for some $e_1, e_2 \notin E$. By $h.s[e \mapsto m]$, we mean the heap identical to h, except that the source function now maps $e \in E$ to $m \in M$. This is formally defined as $h.s[e \mapsto m] = (\overline{M}, E, s[e \mapsto m], t, \tau, \lambda)$. The definitions of $h.t[e \mapsto m]$, $h.\tau[e \mapsto m]$, and $h.\lambda[x \mapsto m]$ are analogous.

3 Programming Language

In this section, we briefly present the class of programs which our analysis is designed for. We also formalize the transition systems which are induced by such programs.

In particular, our analysis and the prototype tool implementing it are designed for sequential programs written in a subset of the C language. The considered subset contains the common control flow statements (like `if`, `while`, `for`, etc.) and the C pointer manipulating statements, excluding pointer arithmetics and type casting. As for the structures describing nodes of dynamic data structures, we—for simplicity of the presentation as well as of the prototype implementation—allow one or two selectors to be used only. However, one can easily generalize the approach to more selectors. Statements manipulating data other than pointers (e.g., integers, arrays, etc.) are ignored—or, in case of tests, replaced by a non-deterministic choice. We allow non-recursive functions that can be inlined[2].

Figure 2 contains an example code snippet written in the considered C subset (up to the tests on integer data that will be replaced by a non-deterministic choice for the analysis). In this example, the data structure `DLL` represents nodes of a doubly-linked list with two successor pointers as well as a data value. The function `merge` takes as input two doubly-linked lists and combines them into one doubly-linked list[3]. In Figure 1, the result of executing two of the statements in the `merge` can be seen. From the top graph, the middle is generated by executing the statement at line `20`. By then executing the statement at line `25`, the bottom graph is generated. (Note that instead of

```
1  typedef struct DLL {
2    struct DLL *next;
3    struct DLL *prev;
4    int data;
5  } DLL;
6
7  DLL *merge(DLL *l1, DLL *l2) {
   ...
17   while(!(x==NULL)&&!(y==NULL)) {
18     if(x->data < y->data) {
19       t = x;
20       x = t->next;
21     } else {
22       t = y;
23       y = t->next;
24     }
25     t->prev = t2;
26     t2->next = t;
27     t2 = t;
28   }
   ...
```

Fig. 2. A program for merging doubly-linked lists

[2] Alternatively, one could use function summaries, which we, however, not consider here.

[3] In fact, if the input lists are sorted, the output list will be sorted too, but this is not of interest for our current analysis—let us, however, note that one can think of extending the analysis to track ordering relations between data in a similar way as in [2], which we consider as one of interesting possible directions for future work.

the `next` and `prev` selectors, the figure uses selectors 1 and 2, respectively.)

Operational Semantics and the Induced Transition System. From a C program, we can extract a *control flow graph* (PC, T) by standard techniques. Here PC is a finite set of *program counters*, and T is a finite set of transitions. A transition t is a tuple of the form (pc, op, pc') where $pc, pc' \in PC$, and op is an operation manipulating the heap. The operation op is of one of the following forms:

- `x == y` or `x != y`, which means that the program checks the stated condition.
- `x = y`, `x = y.next(i)`, or `x.next(i) = y`, which are assignments functioning in the same way as assignments in the C language[4].
- `x = malloc()` or `free(x)`, which are allocation and deallocation of dynamic memory, working in the same manner as in the C language.

When firing t, the program counter is updated from pc to pc', and the heap is modified according to op with the usual C semantics formalized below.

The induced transition system. We will now define the transition system $(S, \longrightarrow)$ induced by a control flow graph (PC, T). The states of the transition system are pairs (pc, h) where $pc \in PC$ is the current location in the program, and h is a heap. The transition relation $\longrightarrow$ reflects the way that the program manipulates the heap during program execution.

Given states $s = (pc, h)$ and $s' = (pc', h')$ there is a transition from s to s', written $s \longrightarrow s'$, if there is a transition $(pc, op, pc') \in T$ such that $h \xrightarrow{op} h'$. The condition $h \xrightarrow{op} h'$ holds if the operation op can be performed to change the heap h into the heap h'. The definition of $\xrightarrow{op}$ is found below.

Assume two heaps $h = (\overline{M}, E, s, t, \tau, \lambda)$ and $h' = (\overline{M}', E', s', t', \tau', \lambda')$. We say that $h \xrightarrow{op} h'$ if one of the following is fulfilled:

- op is of the form `x == y`, $\lambda(x) = \lambda(y) \neq *$, and $h = h'$.[5]
- op is of the form `x != y`, $\lambda(x) \neq \lambda(y)$, $\lambda(x) \neq *$, $\lambda(y) \neq *$, and $h = h'$.
- op is of the form `x = y`, $\lambda(y) \neq *$, and $h' = h.\lambda[x \mapsto \lambda(y)]$.
- op is of the form `x = y.next(i)`, $\lambda(y) \notin \{*, \#\}$, $succ_{h,i}(\lambda(y)) \neq *$, and $h' = h.\lambda[x \mapsto succ_{h,i}(\lambda(y))]$.
- op is of the form `x.next(i) = y`, $\lambda(x) \neq *$, $\lambda(y) \neq *$, and $h' = h.t[e \mapsto \lambda(y)]$ where e is the unique edge in E such that $s(e) = \lambda(x)$ and $\tau(e) = i$.
- op is of the form `x = malloc()` and there is a heap h_1 such that $h_1 = h \oplus m$ and $h' = h_1.\lambda[x \mapsto m]$ for some $m \notin \overline{M}$.[6]

[4] Here, `next(i)` refers to the i-th selector of the appropriate memory cell.

[5] Note that the requirement that $\lambda(x)$ and $\lambda(y)$ are not dangling pointers are not part of the standard C semantics. Comparing dangling pointers are, however, bad practice and our tool therefore warns the user

[6] Although the malloc operation may fail, we assume for simplicity of presentation that it always succeeds.

– *op* is of the form `free(x)`, $\lambda(x) \neq *$, and $h' = h \ominus \lambda(x)$.

4 Signatures

In this section, we introduce the notion of signatures which is a symbolic representation of infinite sets of heaps.

Intuitively, a signature is a predicate describing a set of minimal conditions that a heap has to fulfill to satisfy the predicate. It can be viewed as a heap with some parts "missing".

Formally, a signature is defined as a tuple $(\overline{M}, E, s, t, \tau, \lambda)$ in the same way as a heap, with the difference that we allow the τ and λ functions to be partial. For signatures, we also require some invariants to be obeyed, but they are not as strict as the invariants for heaps. More precisely, a signature has to obey the following invariants:

1. $\forall c \in C\ \forall m \in M : |s^{-1}(m) \cap \tau^{-1}(c)| \leq 1$,
2. $\forall m \in M : |s^{-1}(m)| \leq 2$.

These invariants say that a signature can have *at most* one outgoing edge of each color in the set $\{1, 2\}$, and at most two outgoing edges in total. Note that heaps are a special case of signatures, which means that each heap is also a signature.

Operations on signatures. We formalize the notion of a signature as a predicate by introducing an ordering on signatures. First, we introduce some additional notation for manipulating signatures. Recall that, for a heap $h = (\overline{M}, E, s, t, \tau, \lambda)$ and $m \in M$, $h \ominus m$ is a heap identical to h except that m has been deleted. As the formal definition of $\ominus$ carries over directly to signatures, we will use it also for signatures.

Given a signature $sig = (\overline{M}, E, s, t, \tau, \lambda)$, we define the removal of an edge $e \in E$, written $sig \boxminus e$, as the signature $(\overline{M}, E', s', t', \tau', \lambda)$ where $E' = E \setminus \{e\}$, $s' = s|_{E'}$, $t' = t|_{E'}$, and $\tau' = \tau|_{E'}$. Similarly, given $m_1 \in M$, $m_2 \in \overline{M}$, and $c \in C$, the addition of a c-edge from m_1 to m_2 is written $sig \boxplus (m_1 \xrightarrow{c} m_2)$. This is formalized as $sig \boxplus (m_1 \xrightarrow{c} m_2) = (\overline{M}, E', s', t', \tau', \lambda)$ where $E' = E \cup \{e'\}$ for some $e' \notin E$, $s' = s[e' \mapsto m_1]$, $t' = t[e' \mapsto m_2]$, and $\tau' = \tau[e' \mapsto c]$. Note that the addition of edges might make the result violate the invariants for signatures. However, we will always use it in such a way that the invariants are preserved. Finally, for $m \notin \overline{M}$, we define $sig.(\overline{M} := \overline{M} \cup \{m\})$ as the signature $(\overline{M} \cup \{m\}, E, s, t, \tau, \lambda)$.

Ordering on signatures. For a signature $sig = (\overline{M}, E, s, t, \tau, \lambda)$ and $m \in \overline{M}$, we say that m is *unlabeled* if $\lambda^{-1}(m) = \emptyset$. We say that m is *isolated* if m is unlabeled and also $s^{-1}(m) = \emptyset$ and $t^{-1}(m) = \emptyset$ both hold. We call m *simple* when m is unlabeled and $s^{-1}(m) = \{e_1\}$, $t^{-1}(m) = \{e_2\}$, $e_1 \neq e_2$, and $\tau(e_1) = \tau(e_2)$ all hold. Intuitively, an isolated cell has no touching edges, whereas a simple cell has exactly one incoming and one outgoing edge of the same color.

For $sig_1 = (\overline{M}_1, E_1, s_1, t_1, \tau_1, \lambda_1)$ and $sig_2 = (\overline{M}_2, E_2, s_2, t_2, \tau_2, \lambda_2)$, we write that $sig_1 \lhd sig_2$ if one of the following is true:

- *Isolated cell deletion.* There is an isolated $m \in M_2$ s.t. $sig_1 = sig_2 \ominus m$.
- *Edge deletion.* There is an edge $e \in E_2$ such that $sig_1 = sig_2 \boxminus e$.
- *Contraction.* There is a simple cell $m \in M_2$, edges $e_1, e_2 \in E_2$ with $t_2(e_1) = m$, $s_2(e_2) = m$, $\tau(e_1) = \tau(e_2)$, and a signature sig' such that $sig' = sig_2.t[e_1 \mapsto t(e_2)]$ and $sig_1 = sig' \ominus m$.
- *Edge decoloring.* There is an edge $e \in E_2$ such that $sig_1 = sig_2.\tau[e \mapsto \bot]$.
- *Label deletion.* There is a label $x \in X$ such that $sig_1 = sig_2.\lambda[x \mapsto \bot]$).

We call the above operations *ordering steps*, and we say that a signature sig_1 is smaller than a signature sig_2 if there is a sequence of ordering steps from sig_2 to sig_1, written $sig_1 \sqsubseteq sig_2$. Formally, $\sqsubseteq$ is the reflexive transitive closure of $\lhd$.

The semantics of signatures. Using the ordering relation $\sqsubseteq$ defined above, we can interpret each signature as a predicate. As previously noted, the intuition is that a heap h satisfies a predicate sig if h contains *at least* the structural information present in sig. We make this precise by saying that h satisfies sig, written $h \in [\![sig]\!]$, if $sig \sqsubseteq h$. In other words, $[\![sig]\!]$ is the set of all heaps in the *upward closure* of sig with respect to the ordering $\sqsubseteq$. For a set S of signatures, we define $[\![S]\!] = \bigcup_{s \in S} [\![s]\!]$.

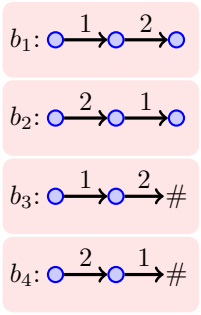
5 Bad Configurations

We will now show how to use the concept of signatures to specify *bad states*. The main idea is to define a finite set of signatures characterizing the set of all heaps that are *not* considered correct. Such a set of signatures is called the set of *bad patterns*.

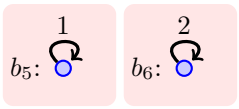
We present the notion on a concrete example, namely, the case of a program that should produce a single acyclic doubly-linked list pointed to by a variable x. In such a case, the following properties are required to hold at the end of the program:

1. Starting from any allocated memory cell, if we follow the `next(1)` pointer and then immediately the `next(2)` pointer, we should end up at the original memory cell.
2. Likewise, starting from any allocated cell, if we follow the `next(2)` pointer and then immediately the `next(1)` pointer, we should end up at the original cell.
3. If we repeatedly follow a pointer of the same type starting from any allocated cell, we should never end up where we started. In other words, no node is reachable from itself in one or more steps using only one type of pointer.
4. The variable x is not dangling, and there are no dangling next pointers.
5. The variable x points to the beginning of the list.
6. There are no unreachable memory cells.

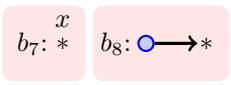
We call properties 1 and 2 *Doubly-Linkedness*, property 3 is called *Non-Cyclicity*, property 4 is called *Absence of Dangling Pointers*, property 5 is called *Pointing to the Beginning of the List*, and, finally, property 6 is called *Absence of Garbage*.



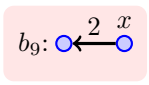
Doubly-Linkedness. As noted above, the set of bad states with respect to a property p is characterized by a set of signatures such that the union of their upward closure with respect to $\sqsubseteq$ contains all heaps not fulfilling p. The property we want to express is that following a pointer of one color and then immediately following a pointer of the other color gets you back to the same node. The bad patterns are then simply the set $\{b_1, b_2, b_3, b_4\}$, shown to the right, as they describe exactly the property of taking one step of each color and *not* ending up where we started.



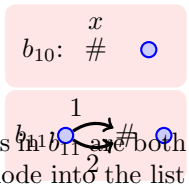
Non-Cyclicity. To describe all states that violate the property of not being cyclic, is to describe exactly those states that do have a cycle. Note that all the edges of the cycle has to be of the same color. Therefore, the bad patterns we get for non-cyclicity is the set $\{b_5, b_6\}$, depicted to the right.



Absence of Dangling Pointers. To describe dangling pointers, two bad patterns suffice—namely, the pattern b_7 depicted to the right stipulates that the variable x that should point to the resulting list is not dangling, and the pattern b_8 requires that there is no dangling next pointer.



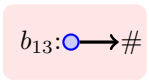
Pointing to the Beginning of the List. To describe that the pointer variable x should point to the beginning of a list, one bad pattern suffices—namely, the pattern b_9 depicted to the right (saying that the node pointed by x has a predecessor). Note that the pattern does not prevent the resulting list from being empty.



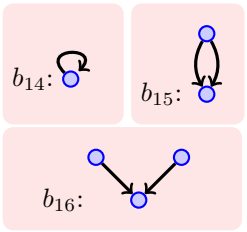
Absence of Garbage. To express that there should be no garbage, the patterns b_{10} and b_{11} are needed. The b_{10} pattern says that if the resulting list is empty, there should be no allocated cell. The b_{11} pattern designed for non-empty lists then builds on that we check the *Doubly-Linkedness* property too. When we assume it to hold, the isolated node can never be part of a well-formed list segment: Indeed, since the two edges in b_{11} are both pointing to the null cell, any possible inclusion of the isolated node into the list results in a pattern that is larger either than b_1 or than b_2.

Clearly, the above properties are common for many programs handling doubly-linked lists (the name of the variable pointing to the resulting list can easily be adjusted, and it is easy to cope with multiple resulting lists too). We now describe some more properties that can easily be expressed and checked in our framework.

Absence of Null Pointer Dereferences. The bad pattern used to prove absence of null pointer dereferences is b_{12}. A particular feature of this pattern is that it is duplicated many times. More precisely, for each program statement of the form `y = x.next(i)` or `x.next(i) = y`, the pattern is added to the starting set of bad states S_{bad} coupled with the program counter just before the operation. In other words, we construct a state that we know would result in a null pointer dereference if reached and try to prove that the configuration is unreachable. The construction for dangling pointer dereferences is analogous.



Cyclicity. To encode that a doubly-linked list is cyclic, we use b_{13} as a bad pattern. Given that we already have *Doubly-Linkedness*, we only need to enforce that the list is not terminated. This is achieved by the existence of a null pointer in the list since such a pointer will break the doubly-linkedness property. Note that this relies on the fact that the result actually is a doubly-linked list.



Treeness. To violate the property of being a tree, the data structure must have a cycle somewhere, two paths to the same node, or two incoming edges to some node. The bad patterns for trees are thus the set $\{b_{14}, b_{15}, b_{16}\}$ depicted to the right.

A Remark on Garbage. Note that the treatment of garbage presented above is not universal in the sense that it is valid for all data structures. In particular, if the data structure under consideration is a tree, garbage cannot expressed in our present framework. Intuitively, there is only one path in each direction that ends with null in a doubly-linked list, whereas a tree can have more paths to null. Thus a pattern like b_{11} is not sufficient since the isolated node can still be incorporated into the tree in a valid way. One way to solve this problem, which is a possible direction for future work, is to add some concept of *anti-edges* which would forbid certain paths in a structure from arising.

6 Reachability Analysis

In this section, we present the algorithm used for analysing the transition system defined in Section 3. We do this by first introducing an abstract transition system that has the property of being *monotonic*. Given this abstract system, we show how to perform *backward reachability analysis*. Such analysis requires the ability to compute the predecessors of a given set of states, all of which is described below.

Monotonic abstraction. Given a transition system $T = (S, \longrightarrow)$ and an ordering $\sqsubseteq$ on S, we say that T is monotonic if the following holds. For any states

s_1, s_2 and s_3 such that $s_1 \sqsubseteq s_2$ and $s_1 \longrightarrow s_3$, we can always find a state s_4 such that $s_2 \sqsubseteq s_4$ and $s_3 \longrightarrow s_4$.

The transition system defined in Section 3 does not exhibit this property. We can, however, construct an over-approximation of our transition relation in such a way that it becomes monotonic. This new transition system $\longrightarrow_A$ is constructed from $\longrightarrow$ by using the state s_3 above as our required s_4. Formally, $s \longrightarrow_A s'$ iff there is an s'' such that $s'' \sqsubseteq s$ and $s'' \longrightarrow s'$.

Since our abstraction generates an over-approximation of the original transition system, if it is shown that no bad pattern is reachable under this abstraction, the result holds for the original program too. The inverse does not hold, and so the analysis may generate false alarms, which, however, does not happen in our experiments. Further, the analysis is not guaranteed to terminate in general. However, it has terminated in all the experiments we have done with it (cf. Section 7).

Auxiliary Operations on Signatures. To perform backward reachability analysis, we need to compute the predecessor relation. We show how to compute the set of predecessors for a given signature with respect to the abstract transition relation $\longrightarrow_A$.

In order to compute pre, we define a number of auxiliary operations. These operations consist of *concretizations*; they add "missing" components to a given signature. The first operation adds a variable x. Intuitively, given a signature sig, in which x is missing, we add x to all places in which x may occur in heaps satisfying sig.

Let $M^{\#} = M \cup \{\#\}$ and $sig = (\overline{M}, E, s, t, \tau, \lambda)$. We define the set $sig\uparrow(\lambda(x) \notin \{\bot, *\})$ to be the set of all signatures $sig' = (\overline{M}', E', s', t', \tau', \lambda')$ s.t. one of the following is true:

- $\lambda(x) \in M^{\#}$ and $sig = sig'$. The variable is already present in sig, so no changes need to be made.
- $\lambda(x) = \bot$ and there is a cell $m \in M^{\#}$ such that $sig' = sig.\lambda[x \mapsto m]$. We add x to a cell that is explicitly represented in sig.
- $\lambda(x) = \bot$, and there is a cell $m \notin \overline{M}$ and a signature sig_1 such that $sig_1 = sig.(\overline{M} := \overline{M} \cup \{m\})$ and $sig' = sig_1.\lambda[x \mapsto m]$. We add x to a cell that is missing in sig. Note that according to the definition of $[\![sig]\!]$, there may exist cells in $h \in [\![sig]\!]$ that are not explicitly represented in sig.
- $\lambda(x) = \bot$, and there is a cell $m \notin \overline{M}$, edges $e_1 \in E$, $e_2 \notin E$ and signatures sig_1, sig_2 and sig_3 such that $sig_1 = sig.(\overline{M} := \overline{M} \cup \{m\})$, $sig_2 = sig_1 \boxplus (m \overset{\tau(e_1)}{\rightarrow} t(e_1))$, $sig_3 = sig_2.t[e_1 \mapsto m]$ and $sig' = sig_3.\lambda[x \mapsto m]$. We add x to a cell that is not explicit in sig. The difference to the previous case is that the missing cell lies *between* two explicit cells m_1, m_2 in sig, along an edge between them.

We now define an operation that adds a missing edge between to specific cells in a signature. Given cells $m_1 \in M, m_2 \in \overline{M}$, we say that a signature sig' is in the set $sig\uparrow(m_1 \overset{c}{\rightarrow} m_2)$ if one of the following is true:

– There is an $e \in E$ such that $s(e) = m_1$, $t(e) = m_2$, $\tau(e) = c$ and $sig' = sig$. The edge is already present, so no addition of edge is needed.
– There is an $e \in E$ such that $s(e) = m_1$, $t(e) = m_2$, $\tau(e) = \bot$, there is no $e' \in E$ such that $s(e') = m_1$ and $\tau(e') = c$, and we have and $sig' = sig.\tau[e \mapsto c]$. There is a decolored edge whose color we can update to c. To do this we need to ensure that there is no such edge already.
– There is no $e \in E$ such that $s(e) = m_1$ and $\tau(e) = c$, $|s^{-1}(m_1)| \leq 1$ and $sig' = sig \boxplus (m_1 \xrightarrow{c} m_2)$. The edge is not present, and m_1 does not already have an outgoing edge of color c. We add the edge to the graph.

The third operation adds labels x and y to the signature in such a way that they both label the same cell.

Formally, we say that a signature sig' is in the set $sig{\uparrow}(\lambda(x) = \lambda(y))$ if one of the following is true:

– $\lambda(x) \in M^{\#}$, $\lambda(x) = \lambda(y)$ and $sig' = sig$. Both labels are already present and labeling the same cell, so no changes are needed.
– $\lambda(y) = \bot$ and there is a $sig_1 \in sig{\uparrow}(\lambda(x) \notin \{\bot, *\})$ such that $sig' = sig_1.\lambda[y \mapsto \lambda_1(x)]$. The label y is not present, so we add it to a signature where x is guaranteed to be present.
– $\lambda(x) = \bot$, $\lambda_{sig}(y) \in M^{\#}$ and $sig' = sig.\lambda[x \mapsto \lambda(y)]$. The label x is not present, so we add it to the cell that is labeled by y.

Computing predecessors. We will now describe how to compute the predecessors of a signature sig and an operation op, written $\mathrm{pre}(op)(sig)$.

Assume a signature $sig = (\overline{M}, E, s, t, \tau, \lambda)$. We define $\mathrm{pre}(\texttt{x = malloc()})(sig)$ as the set sig' of signatures such that there are signatures $sig_1 = (\overline{M}_1, E_1, s_1, t_1, \tau_1, \lambda_1)$, sig_2, and sig_3 satisfying

– $sig_1 \in sig{\uparrow}(\lambda(x) \notin \{\bot, *\})$, there is no $y \in X$ such that $\lambda_1(y) = \lambda_1(x)$ and no $e \in E_1$ such that $t_1(e) = \lambda_1(x)$,
– $sig_2 \in sig_1{\uparrow}(\lambda_1(x) \xrightarrow{1} *)$,
– $sig_3 \in sig_2{\uparrow}(\lambda_1(x) \xrightarrow{2} *)$, and
– $sig' = sig_3 \ominus \lambda_1(x)$.

We let $\mathrm{pre}(\texttt{x = y})(sig)$ be the set sig' of signatures such that there is a signature sig_1 satisfying $sig_1 \in sig{\uparrow}(\lambda(x) = \lambda(y))$ and $sig' = sig_1.\lambda[x \mapsto \bot]$.

Next, we define $\mathrm{pre}(\texttt{x==y})(sig)$ to be the set of all sig' s.t. $sig' \in sig{\uparrow}(\lambda(x) = \lambda(y))$. On the other hand, we define $\mathrm{pre}(\texttt{x!=y})(sig)$ to be the set of all $sig' = (\overline{M}', E', s', t', \tau', \lambda')$ with $\lambda'(x) \neq \lambda'(y)$ and such that there is a signature $sig_1 \in sig{\uparrow}(\lambda(x) \notin \{\bot, *\})$ such that $sig' \in sig_1{\uparrow}(\lambda(y) \notin \{\bot, *\})$.

Further, $\mathrm{pre}(\texttt{x = y.next(i)})(sig)$ is defined as the set of all signatures $sig' = (\overline{M}', E', s', t', \tau', \lambda')$ such that there are sig_1, $sig_2 = (\overline{M}_2, E_2, s_2, t_2, \tau_2, \lambda_2)$, sig_3 with

– $sig_1 = sig{\uparrow}(\lambda(x) \notin \{\bot, *\})$,

- $sig_2 = sig_1{\uparrow}(\lambda(x) \notin \{\bot, *\})$,
- $sig_3 \in sig{\uparrow}(\lambda_2(y) \xrightarrow{i} \lambda_2(x))$, and
- $sig' = sig_3.\lambda[x \mapsto \bot]$.

We let pre(`x.next(i) = y`)(sig) be the set of all $sig' = (\overline{M}', E', s', t', \tau', \lambda')$ such that there are sig_1, sig_2, $sig_3 = (\overline{M}_3, E_3, s_3, t_3, \tau_3, \lambda_3)$, and $e \in E_3$ with

- $sig_1 = sig{\uparrow}(\lambda(x) \notin \{\bot, *\})$,
- $sig_2 = sig_1{\uparrow}(\lambda(x) \notin \{\bot, *\})$,
- $sig_3 \in sig{\uparrow}(\lambda_2(x) \xrightarrow{i} \lambda_2(y))$,
- $s_3(e) = \lambda_3(x)$, $t_3(e) = \lambda_3(y)$, $\tau(e) = i$, and
- $sig' = sig_3 \boxminus e$.

Finally, we define pre(`free(x)`)(sig) to be the set of all $sig' = (\overline{M}', E', s', t', \tau', \lambda')$ such that there are $sig_1 = (\overline{M}_1, E_1, s_1, t_1, \tau_1, \lambda_1)$, sig_2, and $m \notin \overline{M}_1$ with

- $\overline{M}_1 = \overline{M}$, $E_1 = E \setminus t^{-1}(*)$, $s_1 = s|_{E_1}$, $t_1 = t|_{E_1}$, $\tau_1 = \tau|_{E_1}$ and $\lambda_1(x) = \bot$ if $\lambda(x) = *$, $\lambda_1(x) = \lambda(x)$ otherwise,
- $sig_2 = sig_1 \oplus m$, and
- $sig' = sig_2.\lambda[x \mapsto m]$.

The Reachability Algorithm. We are now ready to describe the backward reachability algorithm used for checking safety properties. Given a set S_{bad} of bad patterns for the property under consideration, we compute the successive sets S_0, S_1, S_2, ..., where $S_0 = S_{bad}$ and $S_{i+1} = \bigcup_{s \in S_i} \mathrm{pre}(s)$. Whenever a signature s is generated such that there is a previously generated s' with $s' \sqsubseteq s$, we can safely discard s from the analysis. When all the newly generated signatures are discarded, the analysis is finished. The generated signatures at this point denote all the heaps that can reach a bad heap using the approximate transition relation $\longrightarrow_A$. If all the generated signatures characterize sets that are disjoint from the set of initial states, the safety property holds.

Remark. As the configurations of the transition system are pairs consisting of a heap and a control state, the set S_{bad} is a set of pairs where the control state is a given state, typically the exit state in the control flow graph. This extension is straightforward. For a more in depth discussion of monotonic abstraction and backwards reachability, see [1].

7 Implementation and Experimental Results

We have implemented the above proposed method in a Java prototype. To improve the analysis, we combined the backward reachability algorithm with a light-weight flow-based alias analysis to prune the state space. This analysis works by computing a set of necessary conditions on the program variables for each program counter. Whenever we compute a new signature, we check whether

it intersects with the conditions for the corresponding program counter, and if not, we discard the signature. Our experience with this was very positive, as the two analyses seem to be complementary. In particular, programs with limited branching seemed to benefit from the alias analysis.

We also used the result of the alias analysis to add additional information to the signatures. More precisely, suppose that, the alias analysis has given us that at a specific program counter pc, x and y must alias. Furthermore, suppose that we compute a signature sig that is missing at least one of x and y at pc. We can then safely replace sig with $sig{\uparrow}(\lambda(x) = \lambda(y)))$.

In Table 1, we show results obtained from experiments with our prototype. We considered programs traversing doubly-linked lists, inserting into them (at the beginning or according to the value of the element being inserted—since the value is abstracted away, this amounts to insertion to a random place), merging ordered doubly-linked lists (the ordering is ignored), and reversing them. We also considered algorithms for searching an element in a tree and for inserting new leaves into trees. We ran the experiments using a PC with Intel Core 2 Duo 2.2 GHz and 2GB RAM (using only one core as the implementation is completely serial). The table shows the time it took to run the analysis, and the number of signatures computed throughout the analysis. For each program manipulating doubly-linked lists, we used the set $\{b_1, b_2, \ldots, b_{11}\}$ as described in Section 5 as the set of bad states to start the analysis from. For the programs manipulating trees, we used the set $\{b_{14}, b_{15}, b_{16}\}$.

Table 1. Experimental results

Program	**Struct**	**Time**	**#Sig.**
`Traverse`	`DLL`	11.4 s	294
`Insert`	`DLL`	3.5 s	121
`Ordered Insert`	`DLL`	19.4 s	793
`Merge`	`DLL`	6 min 40 s	8171
`Reverse`	`DLL`	10.8 s	395
`Search`	`Tree`	1.2 s	51
`Insert`	`Tree`	6.8 s	241

The obtained results show that the proposed method can indeed successfully handle non-trivial properties of non-trivial programs. Despite the high running times for some of the examples, our experience gained from the prototype implementation indicates that there is a lot of space for further optimizations as discussed in the following section.

8 Conclusions and Future Work

We have proposed a method for using monotonic abstraction and backward analysis for verification of programs manipulating multiply-linked dynamic data structures. The most attractive feature of the method is its simplicity, concerning the way the shape properties to be checked are specified as well as the abstraction and predecessor computation used. Moreover, the abstraction used in the approach is rather generic, not specialised for some fixed class of dynamic data structures. The proposed approach has been implemented and successfully tested on several programs manipulating doubly-linked lists and trees.

An important direction for future work is to optimize the operations done within the reachability algorithm. This especially concerns checking of entailment on the heap signatures (e.g., using advanced hashing methods to decrease the number of signatures being compared) and/or minimization of the number of generated signatures (perhaps using a notion of a coarser ordering on signatures that could be gradually refined to reach the current precision only if a need be). It also seems interesting to parallelize the approach since there is a lot of space for parallelization in it. We believe that such improvements are worth the effort since the presented approach should—in principle—be applicable even for checking complex properties of complex data structures such as skip lists which are very hard to handle by other approaches without their significant modifications and/or help from the users. Finally, it is also interesting to think of extending the proposed approach with ways of handling non-pointer data, recursion, and/or concurrency.

References

1. P.A. Abdulla. Well (and Better) Quasi-Ordered Transition Systems. Bulletin of Symbolic Logic, 16:457–515, 2010.
2. P.A. Abdulla, M. Atto, J. Cederberg, and R. Ji: Automated Analysis of Data-Dependent Programs with Dynamic Memory. In *Proc. of ATVA'09*, *LNCS* 5799, Springer, 2009.
3. P.A. Abdulla, N. Ben Henda, G. Delzanno, and A. Rezine. Handling Parameterized Systems with Non-atomic Global Conditions. In *Proc. of VMCAI'08*, *LNCS* 4905, Springer, 2008.
4. P.A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, and A. Rezine. Monotonic Abstraction for Programs with Dynamic Memory Heaps. In *Proc. of CAV'08*, *LNCS* 5123, Springer, 2008.
5. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, *LNCS* 4134, Springer, 2006.
6. C. Calcagno, D. Distefano, P.W. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-abduction. In *Proc. of POPL'09*, ACM Press, 2009.
7. J.V. Deshmukh, E.A. Emerson, and P. Gupta. Automatic Verification of Parameterized Data Structures. In *Proc. of TACAS'06*, *LNCS* 3920, Springer, 2006.
8. P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest Automata for Verification of Heap Manipulation. Technical Report FIT-TR-2011-01, FIT BUT, Czech Republic, 2011. `http://www.fit.vutbr.cz/~isimacek/pub/FIT-TR-2011-01.pdf`
9. P. Madhusudan, G. Parlato, and X. Qiu. Decidable Logics Combining Heap Structures and Data. In *Proc. of POPL'11*, ACM Press, 2011.
10. A. Møller and M. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*, ACM Press, 2001.
11. H. H. Nguyen, C. David, S. Qin, and W. N. Chin Automated Verification of Shape and Size Properties via Separation Logic. In *Proc. of VMCAI'07*, *LNCS* 4349, Springer, 2007.
12. J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*, IEEE CS, 2002.

13. S. Rieger and T. Noll. Abstracting Complex Data Structures by Hyperedge Replacement. In *Proc. of ICGT'08*, *LNCS* 5214, Springer, 2008.
14. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.
15. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P.W. O'Hearn. Scalable Shape Analysis for Systems Code. In *Proc. of CAV'08*, *LNCS* 5123, Springer, 2008.
16. K. Zee, V. Kuncak, and M. Rinard. Full Functional Verification of Linked Data Structures. In *Proc. of PLDI'08*, ACM Press, 2008.