Monotonic Abstraction for Programs with Muliply-Linked Structures

Timotheus Jochum
Supervision: David Korzeniewski

WS 2016

Abstract

Automated verification of programs is an important part of software engineering. This work describes the techniques of monotonic abstraction and backwards reachability analysis which are used to perform a shape analysis on programs operating on multiply linked data-structures, as introduced by [3]. We use vertex- and edge-labeled graphs to represent heaps and introduce heap configurations called signatures. Signatures are predicates which we use to check whether a heap satisfies certain properties, such as the absence of null pointer dereferences and non-cyclicity. For the backwards reachability analysis we explain the predecessor function as described by [3]. We conclude with a discussion of the results from a prototype implementation also introduced by [3].

1 Introduction

The verification of correctness of programs is a complex task, especially when it comes to verification of multiply linked data-structures. Such data-structures for example can be *structs* from common programming languages like the C programming-language. The problem with these structures is that due to their usage of pointers, these structures can be of unbounded size and therefore very complex. Previous work has shown, that with the use of *monotonic abstraction* and *backwards reachability analysis*, this problem can be handled well, when applied to single linked data structures [1, 2, 4].
This seminar paper will discuss the results of the use of monotonic abstraction and backwards reachability analysis on programs dealing with multiply linked data structures as described in [3].
We will introduce methods to represent the current heap configuration of the program as a vertex- and edge-labeled graph, in order to perform a backwards reachability analysis. The backwards reachability analysis will start from a set of incorrect heap configurations and tries to calculate whether our initial heap configuration is reachable or not.
To represent incorrect heap configurations, we use *signatures*. A signature is basically a part of a heap with a specific configuration.
We work with sequential programs written in a subset of the C programing-language which includes pointer manipulation and common control statements. To keep the approach simple, we restrict this method on data structures with only two pointers. This approach however can be extended to data structures with more pointers.
This method has been implemented in a Java prototype and tested with several programs manipulating doubly-linked lists and trees. The Results show that monotonic abstraction and backwards reachability analysis can indeed be successfully used for verification of programs operating on multiply linked data-structures [3].

Since this approach uses relatively simple reasoning on the graphs in order to compute predecessors heap configurations, it is highly generic and can be used for other dynamic data-structures as well.
In section 2 we start with several definitions and conventions used through out this paper. We introduce and define heaps as a vertex- and edge-labeled graph in section 3.
In section 4 we introduce signatures and an ordering relation on signatures. Monotonic abstraction is defined in section 5 whereas section 6 covers the reachability algorithm. In section 7 and 8, we discuss the results of the prototype and give a conclusion. This seminar paper is based on the work done by Parosh Aziz Abdulla [3].

2 Definitions and Conventions

This Section introduces definitions and conventions used through out this paper. We define the undefinedness of an element $a \in A$ of a partial function $f\colon A \to B$ as $f(a) = \perp$. We take $f[a \mapsto b]$ to be the function f' such that $f'(a) = b$ and $f'(x) = f(x)$ otherwise. Equally we define $f[A' \mapsto b]$ for all $a \in A'$ where $A' \subseteq A$. We define $f|_{A'}$ as the *restriction* of f to A' as $f|_{A'} = f(a)$ if $a \in A'$, and $f|_{A'} = \perp$ for $a \notin A'$. The inverse image $\{a \in A : f(a) = b\}$ for $b \in B$ will be written as $f^{-1}(b)$.

3 Heap

To model the heap memory used by the program we want to verify, we use a graph with labeled nodes and labeled edges. Each node represents a memory cell and each edge represents a pointer to the next part of the data-structure. In the following we will refer to the pointers as selectors. The edges are labeled with a color to distinguish between these selectors. For simplicity, only structures with two selectors will be considered. The selectors will be labeled 1 and 2 (instead of commonly known selectors *next* and *prev*). The approach we use though can be generalized for structures with any number of selectors.
We now need to define two special nodes, one for the *null pointer* and one for *dangling pointers*. The *null node*, written as #, will be used to model null successors whereas the *dangling node*, written as $*$, will be used to model pointers which used to point to memory that has been either freed, or not yet been allocated. In addition, we need to model a program variable. We model a variable by labeling the node, the variable points to, with the variable's name. The above leads to the following formal definition of a heap.

Definition 3.1 *Let $C = \{1, 2\}$ be a set of colors and X be a set of program variables. We define a heap as 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 pointer and the dangling pointer, respectively.*
- *E is a finite set of edges.*
- *The source function $s : E \to M$ is a total function that returns the source of an edge.*
- *The target function $t : E \to M$ is a total function that returns the target of an edge.*
- *The type function $\tau : E \to C$ is a total function that returns the color of an edge.*
- *$\lambda : X \to M$ is a total function that defines the positions of the program variables.*

To verify that each memory cell has exactly one outgoing edge labeled 1 and one outgoing edge labeled 2, the heap must satisfy the following invariant:

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

Any graph that does not satisfy the above invariant is not a correct heap.

3.1 Operations on Heaps

We define two operations on heaps. The first operation adds a memory cell to the heap and the second operation removes one.
Let $h = (\overline{M}, E, s, t, \tau, \lambda)$ be a heap. $h \oplus m$ for a memory cell $m \in M$, is equal to a heap h', where h' is the heap resulting from the addition of the cell m to the set of memory cells $\overline{M}$ and the addition of two outgoing dangling edges to the set of edges E.
Respectively $h \ominus m$ is defined as the heap h' with the cell m and it's outgoing edges removed, with the addition, that all edges that previously pointed to m are now dangling edges.

4 Signatures

This section introduces signatures. A signature is defined in the same way as a heap, but the semantic is different. We begin with the definition for signatures and then explain it's semantic. A signature is defined as a tuple $(\overline{M}, E, s, t, \tau, \lambda)$ identically to heaps with the exception, that the functions τ and λ can be partial. As for heaps, we also define two invariants for signatures:

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

The first invariant states, that each signature's memory cell can not have more than one outgoing edge of each color. The second invariant states that each memory cell can have at most two outgoing edges in total. A signature is only a signature, if it complies to these invariants.
We use signatures to represent a certain configuration a heap can be in. Intuitively a signature can be viewed as a heap with some parts "missing" [3]. Since a signature is defined in the same way as heaps, we can check whether a heap contains this signature. If it does contain the signature, we say that the signature is satisfied by the heap. Since infinite heaps can satisfy the same signature, we can view a signature as a infinite set of all these heaps that satisfy it. This also means that every heap is a signature but not vice versa.

4.1 Operations on Signatures

As we did for the heaps, we define operations for signatures. Since signatures are basically heaps, we can carry over the definition for $\ominus$ and $\oplus$.
Let $sig = (\overline{M}, E, s, t, \tau, \lambda)$ be a signature.
We define $sig \boxminus e$ as the signature sig', where $e \in E$ has been removed from sig and it's functions s, t and τ are restricted to $E' = E \setminus \{e\}$. We define the addition of an edge in a similar way. Given $m_1 \in M$, $m_2 \in \overline{M}$ and $c \in C$, $sig \boxplus (m_1 \xrightarrow{c} m_2)$ denotes the addition of a transition of color c between m_1 and m_2 to the signature sig. We define the addition of a memory cell $m \notin \overline{M}$ to a signature as $sig' = sig \boxplus m$ where $sig' = (\overline{M} \cup m, E, s, t, \tau, \lambda)$.

4.2 Ordering on Signatures

To define an ordering on signatures, we need to take a look at memory cells first. We begin with the definition of *unlabeled*, *isolated* and *simple* memory cells. Given a signature $sig = (\overline{M}, E, s, t, \tau, \lambda)$ and a cell $m \in \overline{M}$, m is unlabeled if $\lambda^{-1}(m) = \emptyset$. We say the cell is isolated, if it is unlabeled and also $s^{-1}(m) = \emptyset$ and $t^{-1}(m) = \emptyset$ hold. The cell m is called simple, when m is unlabeled and the following constraints hold: $s^{-1}(m) = \{e_1\}$, $t^{-1}(m) = \{e_2\}$, $e_1 \neq e_2$ and $\tau(e_1) = \tau(e_2)$.
Given signatures $sig_1 = (\overline{M}_1, E_1, s_1, t_1, \tau_1, \lambda_1)$ and $sig_2 = (\overline{M}_2, E_2, s_2, t_2, \tau_2, \lambda_2)$, we can now define an *ordering step* on signatures, written as $sig_1 \lhd sig_2$. There are five ordering steps that we will discuss. We begin with the *isolated cell deletion*. If there is an isolated memory cell $m \in M_2$ such that $sig_1 = sig_2 \ominus m$, we can apply the ordering step to sig_2 to get sig_1. The second ordering step is called *edge deletion*. If there is an edge $e \in E_2$, such that $sig_1 = sig_2 \boxminus e$, $sig_1 \lhd sig_2$. *Contraction* is the third ordering step. If 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 $sig_1 = sig_2.t[e_1 \mapsto t(e_2)] \ominus m$, the application of contraction to sig_2 results in sig_1. *Edge decoloring* is the fourth ordering step. If there is an edge $e \in E_2$, such that $sig_1 = sig_2.\tau[e \mapsto \bot]$, applying edge decoloring to sig_2 will result in sig_1. The last ordering step is called *label deletion*. If there is a label $x \in X$, such that $sig_1 = sig_2.\lambda[x \mapsto \bot]$, $sig_1 \lhd sig_2$.
With the ordering steps defined, we can now introduce the *ordering relation* on signatures. We write $sig_1 \lhd\lhd sig_2$, to denote that a signature sig' exists, such that $sig_1 \lhd sig'$ and $sig' \lhd sig_2$. A signature sig_1 is now called *smaller* than a signature sig_2, written as $sig_1 \sqsubseteq sig_2$, if there is a sequence of *ordering steps* from sig_2 to sig_1. The $\sqsubseteq$-relation used here, is the reflexive transitive closure of $\lhd$ [3].
With the ordering relation on signatures defined, we can now take a closer look at the semantics of signatures and define them properly. To denote that a Heap h satisfies a signature sig, we write $h \in [\![sig]\!]$, if $sig \sqsubseteq h$, where $[\![sig]\!]$ represents the set of all heaps in the *upward closure* of sig with respect to our ordering relation. For a set of signatures S, we define $[\![S]\!] = \bigcup_{s \in S} [\![s]\!]$ [3].

4.3 Bad Configurations

With the signatures defined, we can now specify configurations of *heaps* which are not correct. Our intention hereby is to specify a finite set of signatures describing all configurations we consider as *bad states*. This allows us to perform the backwards reachability analysis from this set of states.
To keep this paper as short as possible, we only take a look at one example of such a bad state and only refer to the complete list of bad states [3].

Example of a bad state:

- Non-Cyclicity: This configuration refers to all structures which have a selector pointing to the same memory cell m. We can easily give a set of signatures for this with $S = \{s_1, s_2\}$ where s_1 and s_2 are defined as shown in Figure 1.

Figure 1: Signatures for Non-Cyclicity

4.4 Transition System

Now that we have heaps, signatures and bad states defined, we need to define a system which allows us to change the heap from one state into another, according to the instructions from the program we are verifying. We call this system a *transition system*. We define the transition system as a tuple $(S, \longrightarrow)$, where S is a set of states and $\longrightarrow$ is the transition relation. A state of such a transition system is a pair (pc, h) where h is a heap and pc is the current program counter in the program running. Given states $s = (pc, h)$ and $s' = (pc', h')$, we say a transition from s to s' exists, written as $s \longrightarrow s'$, if there is a transition, such that $h \xrightarrow{op} h'$.
In other words, there is a heap operation defined which changes the heap h into h'. The following operations are defined in our subset of the C programing language:

- op is of the form $x == y$
- op is of the form $x\ != y$
- op is of the form $x = y$
- op is of the form $x = y.next(i)$
- op is of the form $x.next(i) = y$
- op is of the form $x = malloc()$
- op is of the form $free(x)$

5 Monotonic Abstraction

This section will introduce the concept of monotonic abstraction. We call a transition system $(S, \longrightarrow)$, with the ordering relation $\sqsubseteq$ we defined earlier monotonic, if the following holds. For any given states sig_1, sig_2 and sig_3 such that $sig_1 \sqsubseteq sig_2$ and $sig_1 \longrightarrow sig_3$, we can always find a state sig_4 such that $sig_2 \longrightarrow sig_4$ and $sig_3 \sqsubseteq sig_4$. The problem is that the transition system we defined is not monotonic. We therefore have to construct an over-approximation for this system in such a way that if the over-approximation does not contain any reachable bad states, we can conclude that the original system does also not contain any reachable bad states. We do this by redefining the transition relation. Let $\longrightarrow$ be the original transition relation, then $\longrightarrow_A$ is the transition relation $\longrightarrow$ in such a way that it becomes monotonic. We can construct $\longrightarrow_A$ by using state sig_3 from the monotonic definition as the sig_4 required by the definition. Or more formally, $s \longrightarrow s'$ iff there is an s'' such that $s'' \sqsubseteq s$ and $s'' \longrightarrow s'$. There is a problem with this approach though, it only holds in one direction. This may generate false alarms during the analysis which however did not occur in the experiments given.

5.1 Computing Predecessors

The following section is about the relation *pre* used to determine all predecessors of a given signature *sig*, with respect to our abstraction of the transition relation $\longrightarrow_A$. In order to compute this relation, we have to define auxiliary operations on signatures. These auxiliary operations, namely *add variable*, *add edge* and *add label*, are procedures which add the given component (variable, edge, label) to all places where the component might occur in a heap h, such that h still satisfies this signature.
The following definitions are entirely based on definitions used in the main reference [3], since they are essential for the calculation of predecessors and can hardly be rewritten.
We begin with the *add variable* operation.
Given $M^{\#} = M \cup \{\#\}$ and $sig = (\overline{M}, E, s, t, \tau, \lambda)$. We define the set $sig \uparrow (\lambda(x) \in M^{\#})$ to be the set of all signatures $sig' = (\overline{M'}, E', s', t', \tau', \lambda')$ so that one of the following is true:

- $\lambda(x) \in M^{\#}$ and $sig = sig'$. In this case, the variable is already present.
- $\lambda(x) = \bot$ and $sig \lhd_x sig'$. We add x to an undefined cell in sig.
- $\lambda(x) = \bot$ and $sig \lhd_{\lambda'(x)} \lhd_x sig'$. We add x to a cell that is not yet present in sig

The second operation we want to define is *add edge*, which adds an edge between two cells in a signature *sig*. Let $m_1 \in M$ and $m_2 \in \overline{M}$, a signature sig' is now in the set $sig \uparrow (m_1 \xrightarrow{e} m_2)$ if one of the following holds:

- There is already an edge $e \in E$ such that $s(e) = m_1$, $t(e) = m_2$, $\tau(e) = c$ and $sig' = sig$.
- There is already an edge $e \in E$ such that $s(e) = m_1$, $t(e) = m_2$, $\tau(e) = \bot$ and there is no $e' \in E$ such that $s(e') = m_1$ and $\tau(e') = c$, and we have a $sig' = sig.\tau[e \mapsto c]$.
- 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 third and last operation that we need to define is the *add label* operation which will add two labels x and y to the given signature sig so that they both label the same cell. We say that a signature sig' is in the set of signatures $sig \uparrow (\lambda(x) = \lambda(y))$, if one of the following holds:

- $\lambda(x) \in M^{\#}$, $\lambda(x) = \lambda(y)$ and $sig' = sig$. Again, this is the simple case where both labels are already present.
- $\lambda(x) = \bot$, $\lambda(y) \in M^{\#}$ and $sig' = sig.\lambda[x \mapsto \lambda(y)]$. We add the given label x since it is not yet present.
- $\lambda(y) = \bot$ and there is a $sig_1 \in sig \uparrow (\lambda(x) \in M^{\#})$ such that $sig' = sig_1.\lambda[y \mapsto \lambda_1(x)]$. Since the label y is not yet present, we add it to a signature where x is guaranteed to be present.

With these auxiliary operations defined, we can now define the actual predecessors function. Given a signature sig and an operation op of our transition system we defined earlier, the set of all predecessors is written as $pre(op)(sig)$. In the following, we will define pre for each operation, by giving a set of constraints that each signature, that is in the set of predecessors, has to satisfy.
Let $sig = (\overline{M}, E, s, t, \tau, \lambda)$ be a signature. We say that the set of signatures
$sig' = pre(x = y)(sig)$ is the set of signatures such that there exists a signatures sig_1 satisfying the following constraint.

- $sig_1 \in sig \uparrow (\lambda(x) = \lambda(y))$ and $sig_1 \lhd_x sig'$.

We define $pre(x == y)(sig)$ in the same way as $sig \uparrow (\lambda(x) = \lambda(y))$.
To define $pre(x \mathrel{!}= y)(sig)$ we take a similar approach. We say that $pre(x \mathrel{!}= y)(sig)$ is the set of all signatures $sig' = (\overline{M'}, E', s', t', \tau', \lambda')$ with $\lambda'(x) \neq \lambda'(y)$ satisfying:

- There is no signature $sig_1 \in sig \uparrow (\lambda(x) \in M^{\#})$ such that $sig' \in sig_1 \uparrow (\lambda(y) \in M^{\#})$.

For $pre(x = y.next(i))(sig)$ we need a slightly different approach.
We say that $pre(x = y.next(i))(sig)$ is defined as the set of signatures $sig' = (\overline{M'}, E', s', t', \tau', \lambda')$ such there exist signatures sig_1, sig_2, and sig_3 which satisfy the following constraints.

- $sig_1 = sig \uparrow (\lambda(x) \in M^{\#})$
- $sig_2 = sig_1 \uparrow (\lambda(y) \in M^{\#}$
- $sig_3 \in sig \uparrow (\lambda_2(y) \xrightarrow{i} \lambda_2(x))$
- $sig' = sig_3.\lambda[x \mapsto \bot]$

In the same manner as for $pre(x = y.next(i))(sig)$ we will now define
$pre(x.next(i) = y))(sig)$ as the set of all signatures $sig' = (\overline{M'}, E', s', t', \tau', \lambda')$ where signatures sig_1, sig_2, sig_3 and an edge $e \in E_3$ exist such that these constraints hold:

- $sig_1 = sig \uparrow (\lambda(x) \in M^{\#})$
- $sig_2 = sig_1 \uparrow (\lambda(x) \in M^{\#})$
- $sig_3 \in sig \uparrow (\lambda_2(x) \xrightarrow{i} \lambda_2(y))$
- $s_3(e) = \lambda_3(x)$, $t_3(e) = \lambda_3(y)$, $\tau(e) = i$
- $sig' = sig_3 \boxminus e$

We define the functions for memory allocation and deallocation as follows. Let sig be a signature. We say $pre(x = malloc())(sig)$ is the set sig' of signatures, such that there exist signatures sig_1, sig_2 and sig_3 satisfying:

- $sig_1 \in sig \uparrow (\lambda(x) \in M^{\#})$, $\lambda_1(x) \neq \#$, and there exists no $y \in X \setminus \{x\}$ such that $\lambda_1(y) = \lambda_1(x)$, $t^{-1}(\lambda(x)) = \emptyset$, and for every edge $e \in E_1$ such that $s_1(e) = \lambda_1(x)$ the following holds: $t_1(e) = *$.
- $sig_2 = sig_1 \ominus \lambda_1(x)$
- $sig' = sig_2.\lambda[x \mapsto \bot]$

We end the definition of pre with the definition for memory deallocation. Given a signature sig, we say that $pre(free(x))(sig)$ is the set sig' of all signatures where signatures sig_1 and sig_2 with $m \notin \overline{M_1}$ exist, which satisfy these constraints:

- $\lambda(x) = *$
- $\overline{M_1} = \overline{M}$, $E_1 = E \setminus t^{-1}(*)$, $s_1 = s|_{E_1}$, $t_1 = t|_{E_1}$, $\tau_1 = \tau|_{E_1}$ and for every $y \in X$, $\lambda_1(y) = \bot \; if \; \lambda(y) = *$, $\lambda_1(y) = \lambda(y)$ otherwise
- $sig_2 = sig_1.(\overline{M} := \overline{M} \cup \{m\})$
- $sig' = sig_2.\lambda[x \mapsto m]$

With the predecessor function now completely defined, we can take a look at the reachability algorithm which uses pre. The prove for the correctness with respect to the abstract transition relation $\longrightarrow_A$, can be found in the main reference of this seminar-paper [3].

6 The Reachability Algorithm

In this section, we will describe how the backwards reachability algorithm works. The algorithm is given a set of bad states S_{bad} as defined earlier and computes all successive sets $S_0, S_1, S_2...$, where $S_0 = S_{bad}$ and $S_{i+1} = \bigcup_{s \in S_i} pre(s)$. If during this process, a signature s is being generated such that there exists a signature s' which has been generated before, with $s' \sqsubseteq s$, the signature s can be discarded from further analysis. The analysis finishes, when all new generated signatures are discarded. At this point, all generated signatures left denote exactly these heaps, which can reach a bad state using our approximated transition relation $\longrightarrow_A$. If both these sets are disjoint, the safety property holds, otherwise it fails.

7 Results

The described verification method for multiply linked data structures has been implemented in a Java prototype by [3]. This implementation though has been extended with a light-weight flow-based alias analysis to prune state space [3]. Results from several experiments with this prototype can be seen in table 1. The table shows the time it took to run the analysis as well as the number of signatures computed throughout the analysis.

Table 1 Experimental Results from [3]

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

These results show that simple programs such as traversing, inserting and searching on multiply linked data structures can indeed be verified in a short amount of time. More complex operations, like merging two lists take several minutes to complete and require way more signatures to be calculated.

8 Conclusion

We have described a method to verify the correctness of programs, operating on multiply linked data-structures, using monotonic abstraction and backwards reachability analysis. The results of the prototype implemented by [3] have shown that the proposed method does indeed work, if implemented correctly.
These results have to be taken with a small amount of concern though, since the sample size is really small. Further tests with a bigger sample size have to be done, to give these results more meaning.
The method also offers room for improvements. Such improvements could for example be parallelization to reduce the time needed for the analysis or optimization of the reachability algorithm itself. This could be done by using more advanced hashing methods to decrease the number of signatures being compared [3].
On the other hand has this approach a fundamental strength which should be mentioned. The method is very generic and can therefore be used for the analysis of other more complex data structures as well. An example for this would be data-structures with more than two selectors or other complex structures like skip lists.
This work also offers opportunities for future work. The method is constrained to the described subset of the C programing language. The method though could be extended to even support recursion and concurrency as well as pointer arithmetics.

References

[1] Parosh Aziz Abdulla, Muhsin Atto, Jonathan Cederberg, and Ran Ji. Automated analysis of data-dependent programs with dynamic memory. In *International Symposium on Automated Technology for Verification and Analysis*, pages 197–212. Springer, 2009.

[2] Parosh Aziz Abdulla, Ahmed Bouajjani, Jonathan Cederberg, Frédéric Haziza, and Ahmed Rezine. Monotonic abstraction for programs with dynamic memory heaps. In *International Conference on Computer Aided Verification*, pages 341–354. Springer, 2008.

[3] Parosh Aziz Abdulla, Jonathan Cederberg, and Tomáš Vojnar. Monotonic abstraction for programs with multiply-linked structures. *International Journal of Foundations of Computer Science*, 24(02):187–210, 2013.

[4] Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno, and Ahmed Rezine. Handling parameterized systems with non-atomic global conditions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 22–36. Springer, 2008.