BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

Towards Shape Analysis in 2LS

Viktor Malík*

Abstract
Many low-level programs work with dynamic data structures on the heap (such as linked lists), which are often a source of bugs. Formal analysis of the shape of these structures can help finding errors or, on the contrary, prove the correctness of a program. We present the main principles behind 2LS, a program analysis framework for C programs, which is based on automatic invariant inference using an SMT solver, and a sketch of a solution that we proposed to the integration of shape analysis into this framework. The proposed solution includes a way how the shape of a program heap can be described using logical formulae and how a first-order SMT solver can be used to infer loop invariants and function summaries for each function of the analysed program. Our approach is based on pointer access paths that describe the shape of the heap by expressing the reachability of heap objects from pointer-typed program variables. The information obtained from the analysis can be used to prove various properties of programs manipulating dynamic data structures, such as the fact that the shape of a linked list does not change after performing an operation that traverses the list and writes data into each node.

Keywords: formal verification – 2LS – template-based analysis – shape analysis – linked lists – pointer access paths – abstract interpretation – SSA form – invariant inference

Supplementary Material: Git repository

*xmalik11@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Currently, there is a large number of tools for program analysis and verification available, each of them designed to verify various properties of programs. However, most of the tools are very narrowly focused in a single area of analysis and fail to handle complex analysis tasks requiring one to deal with multiple features of programs (e.g. pointers and non-pointer data) at the same time. One of the tools trying to combine analysis of different program features is 2LS [1]. It is currently well usable to verify safety as well as termination of programs using various numerical domains (integers, floats or bit-vectors). Moreover, since it combines multiple analysis techniques together, it is able to both prove correctness and find errors in programs. What 2LS lacks, is an ability to analyse programs that manipulate dynamic data structures.

In this work, we give a solution to the integration of shape analysis into 2LS, which is generally aimed to analyse the shape of dynamic data structures (such as lists, trees, etc.). Since the core algorithm of 2LS is based on abstract interpretation, the main task is to design an abstract domain that is able to (approximately) describe the shape of the heap. The domain must be usable in the solver-based analysis approach of 2LS, which among other issues requires one to create an algorithm for synthesising an abstract value from models of satisfiability obtained from the solver.

Most of the current tools for shape analysis use an intermediate representation of the analysed programs in the form of control-flow graphs (CFGs). They compute an abstract state of the heap (representing a set of reachable concrete states) at each program location. The abstract domains used in these tools build on a store-based semantics and describe the shape of the heap using various logics [2, 3, 4], automata [5] or graphs [6], which closely correspond to the real state of the heap.

However, this approach is not usable in 2LS since the programs are represented in an acyclic single static

assignment form (SSA), omitting parts of the control-flow. Moreover, a solver for quantifier-free first-order logic is used, and so we have to use an abstract domain that allows us to reason about the shape of a heap using quantifier-free formulae only.

To this end, we use an approach based on a so-called *storeless semantics*. Contrary to the store-based approaches described earlier, storeless abstract domains represent the heap as a set of pointer access paths [7]. An *access path* does not concretely express the state of the heap, it only describes which dynamic objects are reachable from a pointer. We show that using the access paths, we are able to describe the shape of (the reachable part of) the heap using logical formulae.

The notion of access paths is used in various other tools [8, 7, 9], however, these also represent the source programs as CFGs and compute the sets of reachable program states iteratively using the abstract interpretation approach. On the other hand, the incremental solver-based approach of 2LS allows a simpler creation and combination of abstract domains. This feature could allow us to combine our shape analysis with other analyses already present in 2LS, which might bring new options of analysing interesting and complex program properties that other tools are not able to handle well (e.g. properties combining the shape of a list with values of data stored in its nodes).

Our current solution is oriented towards singly and doubly linked lists and we are able to analyse various shapes of the lists and operations manipulating them (such as reversal of a singly linked list). We compute a summary of each function separately, which allows us to reuse the already computed information in case of multiple calls of the function.

2. Program Verification in 2LS

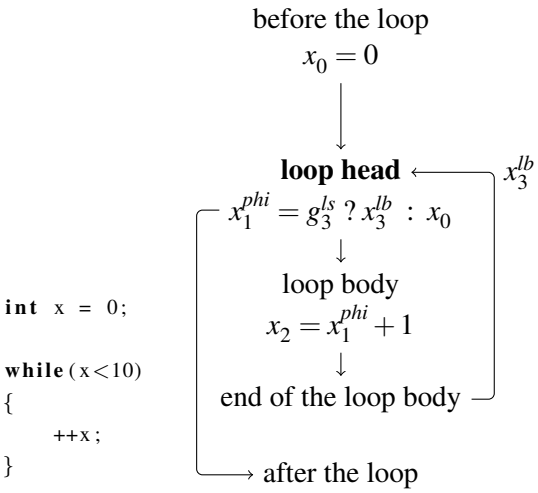
2LS is a program analysis framework built upon the CPROVER [10] verification framework. It is oriented towards analysis of sequential C programs. The core algorithm, called *k*I*k*I, efficiently combines bounded model checking (BMC), *k*-induction and abstract interpretation [11]. Although all these techniques can be used together, we only use *abstract interpretation* for the shape analysis extension of 2LS proposed in this paper.

The *k*I*k*I approach to abstract interpretation adopted in the 2LS framework is based on *inferring of inductive invariants*. This problem, which can be expressed in the (existential fragment of the) second-order logic, is reduced to a problem expressible in quantifier-free first-order logic using so-called *templates*. This reduction enables 2LS to use an SMT solver for an automated inference of loop invariants and function summaries. These are then used to prove properties of the analysed program.

2.1 SSA Encoding of Source Programs

In order to be able to represent the source program as a formula and use it in an SMT solver, 2LS converts it into the *static single assignment form* (SSA). It is a well-known concept of an intermediate program representation satisfying the property that each variable is assigned at most once.

For an acyclic code, its SSA form is a formula that represents exactly the strongest postcondition of running the code. 2LS extends the standard SSA form by an over-approximation of the loops, so that it allows one to reason about program properties using a solver. The SSA representation is made acyclic by cutting loops at the end of the loop body. An example of this conversion is illustrated by Figure 1.



(a) A loop in C **(b)** SSA encoding of the loop

Figure 1. Conversion of loops into the SSA form used in 2LS

The loop has been cut at the end of its body: instead of passing the version of x from the end of the loop body (x_2) back to the Φ node in the loop head, a free "loop-back" variable x_3^{lb} is passed. The choice of the value of x in the Φ node is made non-deterministically using the free boolean "loop-select" variable g_3^{ls}. This way, the SSA form is made acyclic, and thus it always holds when passed to the solver.

Since x_3^{lb} and g_3^{ls} are free variables, this representation is an over-approximation of the actual program traces. The precision can be improved by constraining the value of x_3^{lb} by means of a *loop invariant*, which will be inferred during the analysis. A loop invari-

ant describes a property in the given abstract domain that holds at the loop entry (x_0) and at the end of the loop body (x_2) and hence also on the loop-back edge (x_3^{lb}) [11].

2.2 Template-based Verification

For simplicity, the following formalisation will view source programs as symbolic transition systems. The state of a program is described by a logical interpretation of logical variables corresponding to each program variable. A set of states can be described using a formula—the states in the set are defined by models of the formula. Given a vector of variables x, a predicate $Init(x)$ is the predicate describing the initial states. A transition relation is described as a formula $Trans(x,x')$. From these, it is possible to determine the set of all reachable states as the least fixed-point of the transition relation starting from the states described by $Init(x)$. This is, however, difficult to compute, so instead an *inductive invariant* is used. Inv is an inductive invariant if it has the property:

$$\forall x,x'.(Inv(x) \wedge Trans(x,x') \Longrightarrow Inv(x')) \quad (1)$$

An inductive invariant defined as above is a description of a fixed-point of the transition relation.

A verification task does often require showing that the set of all reachable states does not intersect with the set of error states denoted $Err(x)$. Using the concept of inductive invariants and existential second-order quantification, we can formalise it as:

$$\begin{aligned}\exists_2 Inv.\ \forall x,x'.\ &(Init(x) \Longrightarrow Inv(x)) \wedge \\ &(Inv(x) \wedge Trans(x,x') \Longrightarrow Inv(x')) \wedge \\ &(Inv(x) \Rightarrow \neg Err(x))\end{aligned} \quad (2)$$

To directly handle Formula 2 by a solver, it would be needed to handle second-order logic quantification. Since a suitably general and efficient second-order solver is not currently available, the problem is reduced to one that can be solved by an iterative application of a first-order solver. This reduction is done by restricting the form of the inductive invariant Inv to $\mathscr{T}(x,\delta)$ where $\mathscr{T}$ is a fixed expression (a so-called *template*) over program variables x and template parameters δ. This restriction corresponds to the choice of an abstract domain in abstract interpretation—a template only captures the properties of the program state space that are relevant for the analysis. This reduces the second-order search for an invariant to a first-order search for template parameters:

$$\begin{aligned}\exists\delta.\ \forall x,x'.\ &(Init(x) \Longrightarrow \mathscr{T}(x,\delta)) \wedge \\ &(\mathscr{T}(x,\delta) \wedge Trans(x,x') \Longrightarrow \mathscr{T}(x',\delta))\end{aligned} \quad (3)$$

Although the problem is now expressible in first-order logic, the formula contains quantifier alternation, which poses a problem for current SMT solvers. This is solved by iteratively checking the negated formula (to turn $\forall$ into $\exists$) for different choices of constants d for the template parameters δ. Then the second conjunct in Formula 3 will have the form:

$$\exists x,x'.\ \neg(\mathscr{T}(x,d) \wedge Trans(x,x') \Longrightarrow \mathscr{T}(x',d)) \quad (4)$$

From the abstract interpretation point of view, d is an abstract value, i.e. it represents (*concretises to*) the set of all program states x that satisfy the formula $\mathscr{T}(x,d)$. The abstract values representing the infimum $\bot$ and supremum $\top$ of the abstract domain denote the empty set and the whole state space, respectively: $\mathscr{T}(x,\bot) \equiv false$ and $\mathscr{T}(x,\top) \equiv true$ [11].

Formally, the concretisation function γ is the same for each abstract domain:

$$\gamma(d) = \{x \mid \mathscr{T}(x,d) \equiv true\} \quad (5)$$

As for the abstraction function, it is essential to find the most precise abstract value representing a concrete program state. Thus:

$$\alpha(x) = \min(d) \text{ such that } \mathscr{T}(x,d) \equiv true \quad (6)$$

Since the abstract domain forms a partially ordered set and $\mathscr{T}(x,\top) \equiv true$, existence of such minimal value d is guaranteed.

In order to be more effective, 2LS uses so-called *incremental solving*. This technique uses one solver instance for multiple solving iterations, where clauses added in the last iteration are re-solved only.

In particular, during the analysis, the SSA representation of the source program is passed first to the solver. It describes the formula $Init(x) \wedge Trans(x,x')$ and since it is acyclic, it is guaranteed to always hold. Moreover, the SSA form explicitly expresses the control flow and thus removes the need to (directly) implement abstract transformers.

The algorithm for the invariant inference takes an initial value of $d = \bot$ and iteratively passes the below quantifier-free formula to the incremental solver, which conjoins it to the SSA formula.

$$\mathscr{T}(x,d) \wedge \neg(\mathscr{T}(x',d)) \quad (7)$$

This formula corresponds to the negation in Formula 4 where $Trans(x,x')$ is omitted since it was already passed

to the solver in the form of the SSA. If Formula 7 is unsatisfiable, then an invariant has been found, otherwise a model of satisfiability is returned by the solver. The model represents a counterexample to the current instance of the template being an invariant. The current value of the template parameter d is then refined by joining it with the obtained model of satisfiability using a domain-specific join operator [11].

3. Proposal of a Shape Analysis for 2LS

Integration of a new type of analysis into 2LS requires designing an abstract domain that can describe the desired properties of the programs to be analysed, in our case the shape of the program heap. This involves a proposal of a suitable form of the template and its parameters, and creating the corresponding join algorithm. Apart from this, it turns out that, for the domain we propose, the process of the generation of the SSA form needs to be improved.

3.1 Representing Heap Operations in SSA

2LS uses an SMT solver based on the bit-vector and the array theory, which identifies variables through their symbolic names only. The problem is that dynamically allocated objects do not have any names, and therefore we must introduce new symbols to represent them. In order to achieve this, we replace each call of the *malloc* function in the source program by an instantiation of a new abstract dynamic object and return its symbolic address as the result of the call:

$$\texttt{malloc(sizeof}(t)) \longrightarrow \&do_i \tag{8}$$

Here the type of do_i is t, and i is the program location in which the call occurs. The created dynamic object is an abstraction since it represents all objects allocated at the given location. We denote the set of all abstract dynamic objects that appear in a given program, extended by `null`, as *Obj*.

Another problem comes with dereferences. A dereference of a pointer may result into different objects at different execution points. To ensure correct dereferencing, we perform a simple static *points-to* analysis prior to the conversion into the SSA. This analysis determines for each pointer variable a set of (abstract) memory objects it can be dereferenced into at each program location where it is used. In case the pointer can be `null` or its value can be unknown (e.g. because it has not been initialized), this information is also determined. The information obtained from the points-to analysis is then used to correctly represent pointer dereferences, where each dereference is replaced by a case split containing a case for each potential target object.

3.2 Template Shape Domain

We restrict our template to use only those memory objects that describe the shape of the heap—namely pointer-typed variables and structure-typed heap objects. For the following formulae, we define the sets *Ptr* and *Fld* to contain all pointers and structure fields in the source program, respectively. The formula represented by a template is then a conjunction of basic formulae, so-called *template rows*, where each row corresponds to one of these memory objects. Since two types of objects are considered (pointers and heap objects), we have to use two types of template rows equivalent to two parts of the memory:

- The *stack* part describes the points-to relation between the static variables (pointers) and the heap. The formula of the stack part is defined as a conjunction of *stack rows*:

$$\mathscr{T}^S \equiv \bigwedge_{p \in Ptr} \mathscr{T}^S_i(p, d^S_i) \tag{9}$$

  A stack row $\mathscr{T}^S_i$ is a parametrized formula which specifies a set of abstract objects given by the parameter $d^S_i \in Obj$ that the row pointer p may point to:

$$\mathscr{T}^S_i(p, d^S_i) \equiv \bigvee_{o \in d^S_i} p = \&o \tag{10}$$

- The *heap* part describes the shape of the heap. We define a *heap row* for each pair of an abstract dynamic object and its (pointer-typed) field, thus the formula for the heap part is:

$$\mathscr{T}^H \equiv \bigwedge_{(o,f) \in Obj \times Fld} \mathscr{T}^H_i((o,f), d^H_i) \tag{11}$$

  Here, $\mathscr{T}^H_i$ is a heap row which is a formula that characterizes a set of access paths (given by the parameter d^H_i) leading from the abstract row object o via the row pointer field f:

$$\mathscr{T}^H((o,f), d^H_i) \equiv \bigwedge_{(d,O) \in d^H_i} path(o,f,d)[O] \tag{12}$$

  The predicate $path(o,f,d)[O]$ reflects that there is a path in the heap leading from an object o via a field f into a destination object d and the path passes through a set of heap objects O.

Finally, using the formulae for the heap and the stack part, we define the template for our domain as:

$$\mathscr{T} \equiv \mathscr{T}^S \wedge \mathscr{T}^H \tag{13}$$

By determining the points-to relation between the stack and the heap and the set of all access paths, we

can efficiently describe the shape of linked lists in the heap. For example, a `null` terminated singly-linked list that is allocated at location i within one loop and whose head is pointed by a pointer variable h will be represented by the following value in our shape domain:

$$h = \&do_i \ \wedge \\ path(do_i, next, \texttt{null})[do_i] \tag{14}$$

For simplicity, set braces will be skipped for the set of objects that the path passes through.

3.3 Abstract Value Synthesis Algorithm

The join algorithm for our shape domain is performed row-wise. The update of the row parameter is different for every type of the row. It depends on the value assigned to the pointer (for a *stack* row) or to the field of the dynamic object (for a *heap* row) in the obtained model of satisfiability.

The update of a *stack* row parameter value is simple. The object whose address was assigned to the corresponding pointer in the model of satisfiability is added to the set representing the row value. This way, we collect exactly all objects that the pointer can reference.

The update of a *heap* row parameter value is more complicated since the access paths of different template rows depend on each other. Thus we create a transitive closure over the set of all paths in the heap, according to the following formula:

$$path(o', f, d)[O] \ \wedge \ o.f = \&o' \\ \Rightarrow path(o, f, d)[O \cup \{o'\}] \tag{15}$$

4. Implementation and Results

We have integrated the shape domain described in the previous section into the current version of 2LS. Running the core algorithm of 2LS together with our domain allows us to obtain invariants describing various linked lists in the heap. We now show two cases where we used our domain to analyse programs working with dynamic data structures (Sections 4.1 and 4.2).

4.1 List Reversal Analysis

Consider the following piece of code that creates the singly linked list shown in Figure 2:

```
1 struct node *head = malloc(sizeof(*head));
2 struct node *mid = malloc(sizeof(*mid));
3 struct node *tail = malloc(sizeof(*tail));
4 head->next = mid;
5 mid->next = tail;
```

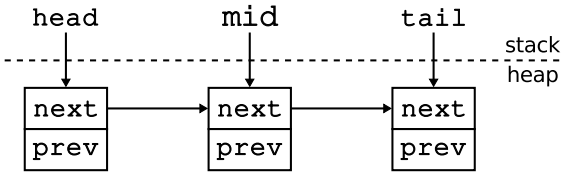


Figure 2. Example of a singly linked list

Running our analysis on this program will infer the following invariants:

$$head = \&do_1 \ \wedge \ mid = \&do_2 \ \wedge \ tail = \&do_3 \ \wedge \\ path(do_1, next, do_2)[\,] \ \wedge \\ path(do_1, next, do_3)[do_2] \ \wedge \\ path(do_2, next, do_3)[\,]$$

Next, let us have a loop that traverses a singly linked list and reverses it:

```
1 struct node *list = head;
2 while (list != tail) {
3     struct node *next = list->next;
4     next->prev = list;
5     list->next = NULL;
6     list = next;
7 }
```

The result of such operation is a list shown in Figure 3.

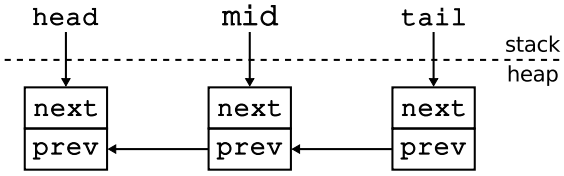


Figure 3. A reversed singly-linked list from Figure 2.

After analysing the given loop, we obtain new invariants:

$$path(do_2, prev, do_1)[\,] \\ path(do_3, prev, do_2)[\,] \\ path(do_3, prev, do_1)[do_2]$$

Using these invariants, it is possible to prove that the loop for list reversal does not change the ordering of the nodes of the given list.

4.2 Function Summaries

One of the advantages of 2LS is its support for interprocedural analysis where each function is analysed separately. The result of such analysis is a function summary describing how a function transforms its inputs into outputs. A summary has a form of inductive

invariant (hence a template with computed parameters) restricted on function inputs and outputs.

Combining the interprocedural analysis with the shape domain, we are able to analyse how a single function changes the shape of the heap. For instance, the following function creates a new node and appends it to the head of a linked list referenced by the function parameter:

```
void chain_node(struct node **head){
    struct node *node = malloc(sizeof(*node));
    node->next = *head;
    *head = node;
}
```

2LS with our shape domain computes the summary of the given function as follows:

Inputs : $head, head^{obj}$

Outputs : $head_2^{obj}$

Summary :

$head_2^{obj} = \&do_0$

$path(do_0, next, head^{obj})[]$

Since the function is analysed independently of other functions, we introduce a symbol $head^{obj}$ to be an abstraction of the object pointed by *head* at the function input. The summary expresses that this object is written within the function since the output SSA variable $head_2^{obj}$ contains a location specification, which means that the object appeared on the left side of an assignment in the function. This new version $head_2^{obj}$ points to a newly allocated object, which is linked to the original list head via the *next* field. Whenever the function `chain_node` is called in the source program, the call is replaced by the summary, and $head^{obj}$ and $head_2^{obj}$ are bound to the corresponding objects from the caller function.

Moreover, since we introduced the abstraction of objects pointed by function parameters, this allows us to analyse functions passing inputs by reference in other domains besides the shape domain, too, which was previously not possible in 2LS.

4.3 Experiments

We performed a number of experiments to show the utility of our extension of 2LS. One of the relevant benchmarks in the community of software analysis and verification are benchmarks from the Software Verification Competition (SV-COMP). These are also well-suitable for us since 2LS participated in SV-COMP'17 and so we are able to compare our results with scores obtained without our shape analysis.

Since our extension is aimed at shape analysis, the most important category is the Heap Reachability category. We re-ran the benchmark for this category at SV-COMP'17 with and without our extension. The comparison can be seen in Table 1.

Table 1. A comparison between 2LS without and with our extension on the SV-COMP'17 Heap Reachability category

	Shape analysis	
	Without	**With**
Number of tasks	173	173
Correct results	76	82
Incorrect results	18	4
Inconclusive	79	87
Score	**-240**	**32**

We can see that our analysis increased the number of correctly verified tasks and decreased the number of incorrect results, which led to a significant increase of the score. However, the number of inconclusive tasks increased, too, since a part of the previously incorrect results are now indecisive. A large part of the tasks are inconclusive because they depend on analysis of non-pointer data, which we are not able to handle, yet. The combination of our shape domain with numerical domains of 2LS could thus bring another improvement in the score.

Another set of relevant examples can be found in already existing tools for shape analysis. Currently, one of the best tools in this area is the Predator tool [12]. We extracted the regression tests from this tool that work with singly (SLL) and doubly linked lists (DLL). We ran 2LS on each of these tasks without and then with our extension. Table 2 shows that our analysis notably increased the number of successfully verified programs.

Table 2. A comparison between the number of successfully verified tasks from the Predator benchmark with and without our extension of 2LS

	SLL	DLL
Number of tasks	17	8
Without shape analysis	6	2
With shape analysis	**14**	**7**

In both experiments, there is a number of tasks that is successfully verified even without the shape analysis. This is caused by the fact that the programs do not contain any loops, thus no invariant is to be

computed, and the SSA form with the SMT solver is enough to prove the program correctness or find an error.

5. Conclusion

2LS is a program verification framework efficiently combining multiple analysis techniques together. It is able to verify a large scale of programs and their properties, mainly related to data-flow among numerical variables (which is acknowledged by the gold medal in the Float category of the Software Verification Competition SV-COMP 2016).

One of the main drawbacks of 2LS is its incapability to analyse programs containing dynamic data structures. Wry try to eliminate this weakness by implementing a new abstract domain for the description of the shape of the program heap. Since the core algorithm is based on a template-based invariant inference using an SMT solver, our domain describes the heap using logical formulae.

The solution we propose is based on pointer access paths that express the reachability of heap objects from pointer variables. Moreover, we improve the static points-to analysis that is used in 2LS for a correct representation of pointer dereferences in the SSA form.

We performed a series of experiments from the Heap Reachability category of SVCOMP'17 and from the Predator tool. The results prove that out extension to 2LS brought improvement in the analysis of programs working with dynamic data structures.From the point of view of the complexity of the involved shape updates, the most complex example that we are able to handle is the reversal of a singly linked list, in which we are able to analyse each function separately.

In the future, we intend to improve the domain to be able to verify more complicated lists. The main goal is to combine our shape analysis with the value analysis already present in 2LS, which would open new possibilities of analysing interesting properties of programs that most of the other tools cannot handle.

Acknowledgements

I would like to thank my supervisor Tomáš Vojnar for consultations on the theoretical aspects of this work. I would also like to thank Peter Schrammel of the University of Sussex, who maintains the project of 2LS, for his continuous support in both theoretical and implementation parts of the project.

References

[1] Peter Schrammel and Daniel Kroening. 2LS for Program Analysis - (Competition Contribution). In *TACAS*, volume 9636 of *LNCS*, pages 905–907. Springer, 2016.

[2] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[3] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, pages 221–231. ACM, 2001.

[4] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118. ACM, 1999.

[5] Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. *Forest Automata for Verification of Heap Manipulation*, pages 424–440. Springer, 2011.

[6] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. *Byte-Precise Verification of Low-Level List Manipulation*, pages 215–237. Springer, 2013.

[7] Noam Rinetzky, Jörg Bauer, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, pages 296–309, 2005.

[8] Stephen Chong, , and Radu Rugina. Static analysis of accessed regions in recursive data structures. In *SAS*, pages 463–482. Springer, 2003.

[9] Ivan Matosevic and Tarek S. Abdelrahman. Efficient bottom-up heap analysis for symbolic path-based data access summaries. In *CGO*, pages 252–263. ACM, 2012.

[10] Cprover. http://www.cprover.org/.

[11] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. Safety Verification and Refutation by *k*-Invariants and *k*-Induction. In *SAS*, volume 9291 of *LNCS*, pages 145–161. Springer, 2015.

[12] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Predator: A shape analyzer based on symbolic memory graphs. In *TACAS*, pages 412–414. Springer, 2014.