EPiC Series in Computing
Volume 73, 2020, Pages 151–168
LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Learning Data Structure Shapes from Memory Graphs[*]

Jan H. Boockmann and Gerald Lüttgen

Software Technologies Research Group, University of Bamberg, Germany
{jan.boockmann, gerald.luettgen}@swt-bamberg.de

Abstract

This paper presents a novel algorithm for automatically learning recursive shape predicates from memory graphs, so as to formally describe the pointer-based data structures contained in a program. These predicates are expressed in separation logic and can be used, e.g., to construct efficient secure wrappers that validate the shape of data structures exchanged between trust boundaries at runtime. Our approach first decomposes memory graph(s) into sub-graphs, each of which exhibits a single data structure, and generates candidate shape predicates of increasing complexity, which are expressed as rule sets in Prolog. Under separation logic semantics, a meta-interpreter then performs a systematic search for a subset of rules that form a shape predicate that non-trivially and concisely captures the data structure. Our algorithm is implemented in the prototype tool `ShaPE` and evaluated on examples from the real-world and the literature. It is shown that our approach indeed learns concise predicates for many standard data structures and their implementation variations, and thus alleviates software engineers from what has been a time-consuming manual task.

1 Introduction

The verification and analysis of computer programs containing pointer-based data structures such as doubly-linked lists and trees is a particular challenge and often involves establishing invariant properties concerning the structure's *shape*. Data structure shapes are typically specified by recursive predicates, e.g., expressed in *separation logic* [6, 29, 33] which allows for compositionally modeling and reasoning about the heap. Shapes consider structural information only and ignore semantic properties such as orderdness or balancedness.

As an example application, consider trusted execution environments such as *Intel SGX* [25] or *Sancus* [28] that provide secure enclaves in which code is executed in isolation from a potentially malicious host system. When data structures with pointers are accessed from within a secure enclave, these pointers may be abused to manipulate the enclave's execution flow. Recent work has demonstrated how shape predicates written in a constructive fragment of the separation logic dialect *VeriFast* [19] can be used to generate *secure wrappers* that can then efficiently check at run-time whether a pointer passed into the trusted environment does indeed

[*]This research is partially supported by the German Research Foundation (DFG) under project DSI2 (grant no. LU 1748/4-2).

point to a data structure of the expected shape [4, 35]. Currently, such *shape predicates* have to be specified manually by software developers, which is a time-consuming task [31].

Shape analysis tools such as *Forester* [15] and *Infer* [10, 11] are static verifiers for reasoning about the memory safety of programs and internally extract required shape information automatically from C programs. Although shape analysis tools start to adopt machine learning techniques [9], they are limited by their use of predefined shape predicates for matching pointer-based data structures in memory graphs. The same applies to dynamic analysis tools such as *HeapDbg* [24] or *DSI* [38], which also excavate shape information. Therefore, existing automatic shape inference techniques are unsuitable to synthesize secure wrappers for system-level applications that use enclaves, as these applications often involve custom data structure implementations. Consequently, the research challenge is how shape predicates can be learned automatically from one or multiple memory graphs, i.e., heap snapshots taken at a desired program location, without relying on predefined shape libraries.

In this paper, we present a novel approach that addresses this challenge by learning a shape predicate from memory graphs captured during the execution of a C program that contains potentially non-standard implementations of list-based data structures. The considered learning problem is a special one since the model, i.e., a shape predicate, is learned from examples, i.e., one or more memory graphs, *and* additional constraints, i.e., our specific search algorithm that is the key contribution of this paper. The learning problem is also non-trivial because no negative but only positive examples are available, possibly just a single memory graph. This raises the question how one can prevent the learning of trivial predicates that simply enumerate the input memory graph(s) or accept any memory graph. We restrict ourselves to shape predicates that are expressible in separation logic without existential quantification or, to be precise, in the VeriFast dialect mentioned above. This restriction is not severe: while it renders data structures such as a lasso-shaped linked lists out of scope, most popular dynamic data structure can be expressed, including (cyclic) singly and doubly-linked lists with and without head or tail pointers and trees with and without root or parent pointers, as well as nested combinations thereof.

Our learning approach is based on a systematic search for shape predicates matching the given memory graph(s). It uses Prolog as its main inference engine, because our search benefits from Prolog's efficient backtracking capabilities which current separation logic tools lack. In a preliminary step, we decompose the provided memory graph(s) into sub-graphs, each exhibiting a single data structure, by identifying separate memory regions and nested data structures. Then, we use a rule schema to iteratively generate candidate Prolog rules of increasing complexity, where heuristics filter out rules that (*i*) are irrelevant for the given memory graphs and would thus hamper the efficiency of our search, or (*ii*) would yield trivial shape descriptions, e.g., rules ignoring the value of particular pointer fields. A meta-interpreter that respects the "resource-aware" semantics of separation logic then searches through the candidate rules with the aim of identifying a subset of the rules, i.e., a predicate, that accurately captures the exhibited data structure; a triviality check excludes poor rules. Recomposing the individually learned predicates yields an overall predicate that matches the input memory graph(s) and that is, in a final step, translated to a shape predicate in the separation logic fragment of VeriFast.

We have implemented our approach in the prototype tool `ShaPE` (*Shape Predicate Extractor*) and evaluated it on a benchmark consisting of real-world, textbook, and synthetic examples, as well as examples taken from the research literature. The results demonstrate that `ShaPE` does indeed learn concise predicates for standard data structures and their implementation variations. An exception are very complex data structures such as cyclic doubly-linked lists, which require shape predicates with four or more parameters, or those data structures that

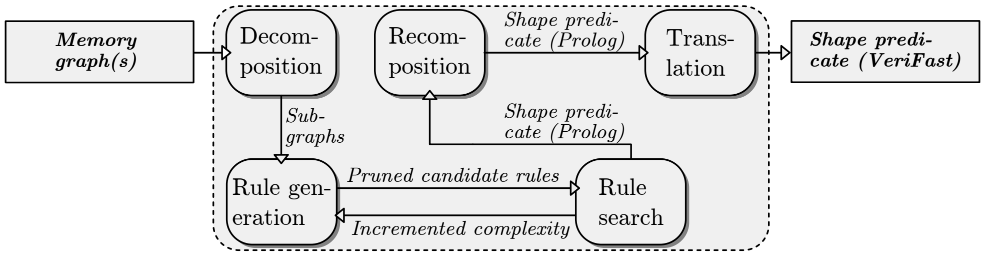


Figure 1. Architecture of our prototype tool ShaPE.

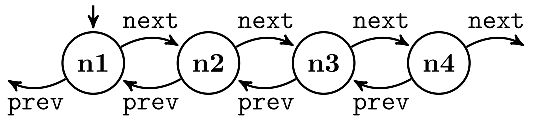
cannot be expressed in a constructive fragment of separation logic. This also applies to general skip-lists; however, ShaPE can in principle detect skip-lists with a given height. In summary, our approach improves upon related work by removing the limitations of considering predefined shape predicates only. It frequently succeeds in relieving software developers from manually phrasing shape predicates, e.g., for synthesizing secure wrappers. We believe that our approach can also benefit techniques for static program verification and program comprehension.

Organization. Sect. 2 describes our algorithm for learning shape predicates, and Sect. 3 introduces our memory graph decomposition. Sect. 4 evaluates our approach, while Sect. 5 discusses related work. Sect. 6 presents our conclusions and suggestions for future work.

2 Core Shape Learning Algorithm

Our approach requires a list of program states (*memory graphs*), which model the stack and heap captured at a particular location during a program's execution. Function arguments and local/global variables on the stack point to heap-allocated chunks of memory (*nodes*) that are interconnected by pointers (*edges*). Nodes are associated with a C struct definition (*type information*), where pointer fields constrain the edges that may originate from and point to a node. Observe that the type system of the C programming language is not strong enough to characterize a data structure shape exhibited at runtime, e.g., binary trees and doubly-linked lists might use the same struct definition, but exhibit different shapes. Because our approach focuses on learning shape predicates, which ignore relations among payload data, we only consider pointer arguments and variables (*entry pointers*) and ignore non-pointer fields. The latter would be necessary information when extending our approach to considering also arrays.

The architecture of our prototype tool ShaPE implementing our approach is depicted in Fig. 1. Initially, component "Decomposition" splits input memory graph(s) into sub-graphs, each representing a single data structure. This allows us to handle programs containing multiple or nested dynamic data structures. Our core learning algorithm then assumes that input memory graph(s) contain a single data structure where all nodes are of the same type. The algorithm is encapsulated in component "Rule generation" that emits a list of pruned candidate rules, and component "Rule search" that employs a Prolog meta-interpreter to search for a group of rules, i.e., a predicate, that characterizes the sub-graphs in a non-trivial way. If no predicate is found, the rule generation is invoked again to produce rules with increased complexity. Each decomposition technique is associated with a corresponding recomposition technique in component "Recomposition" to construct an overall shape predicate from the individually learned shape predicates. Subsequently, component "Translation" phrases the obtained shape predicate as predicates in the constructive separation logic dialect of VeriFast [19].



(a) Observed memory graph

```
struct dll {
  struct dll *next;
  struct dll *prev;
};
```

(b) Available type information

```
node(n1).                 node(n2).             node(n3).               node(n4).
next(n1, n2).             next(n2, n3).         next(n3, n4).           next(n4, null).
prev(n1, null).           prev(n2, n1).         prev(n3, n2).           prev(n4, n3).
entrypointer(ep1, n1).                          entrypointers([ep1]).
```

(c) Prolog facts encoding the observed memory graph

```
entry(This) :- next(This, Next), prev(This, null), p(Next, This).
p(This, P1) :- next(This, null), prev(This,   P1).
p(This, P1) :- next(This, Next), prev(This,   P1), p(Next, This).
```

(d) Learned rules in Prolog syntax, encoding the data structure's shape

```
predicate entry(struct dll* This;) =
  malloc_block_dll(This) &*&
  This->next |-> ?Next &*& This->prev |-> ?Prev &*&
  Prev == NULL && Next != NULL ? p(Next, This) : false;

predicate p(struct dll* This, struct dll* P1;) =
  malloc_block_dll(This) &*&
  This->next |-> ?Next &*& This->prev |-> ?Prev &*&
  Prev == P1 &*& P1 != NULL &*& P1 != This &*&
  Next != NULL && Next != This && Next != P1 ? p(Next, This)
   : Next == NULL ? emp
   : false;
```

(e) Shape predicate translated to the separation logic dialect of VeriFast

Figure 2. Running example of a doubly-linked list (DLL) shape.

We postpone the explanation of our de-/recomposition approach to Sect. 3 and illustrate the workings of our core shape learning algorithm in the following section, using a doubly-linked list (DLL) as running example. Input data, i.e., a single memory graph and its associated type information, and produced output data, i.e., the learned Prolog rules and their corresponding separation logic shape predicate, are shown in Fig. 2. The four nodes of the memory graph (see Fig. 2(a)) exhibit a typical DLL shape. Access to the data structure is granted by the entry pointer, i.e., an unlabeled pointer without an originating source node, which indicates node `n1` as the *entry node*. The accompanying type information (see Fig. 2(b)) contains the single C struct definition `struct dll`. The remainder of this section covers the components "Rule generation", "Rule search", and "Translation" of our tool (cf. Fig. 1), and explains their workings.

2.1 Memory Graph Translation to Prolog Facts

In Prolog, *facts* are used to encode known information whereas *rules* describe means to derive new information from given facts. Each element of a memory graph, i.e., nodes, edges, and entry pointers, is mapped to a corresponding fact type. A `node` fact requires the node id as input, an `edge` fact requires the source and target node id as input, and an `entrypointer` fact requires the name and the entry node id as input. Additionally, a list of all entry pointers is stored in the `entrypointers` fact. Null pointers are encoded by the special atom `null`. The facts representing the memory graph of our running example are shown in Fig. 2(c).

2.2 Generation of Candidate Rules

We employ a rule schema to derive candidate rules that describe each node and its relation to other nodes in the memory graph. In this schema, each rule (*i*) operates exactly on one node, called `This`; (*ii*) dereferences and assigns a value to each field of the node on which it operates; (*iii*) contains zero or multiple recursive calls; and (*iv*) may receive/share information about other nodes via parameters/arguments.

A single predicate rarely suffices to characterize a data structure, because the entry node must often be dealt with differently to the remaining nodes. Hence, we encode a shape predicate by two distinct groups of rules: `entry` and `p` rules. An `entry` rule is invoked once in the beginning, with the list of entry pointers passed as arguments. In contrast, a `p` rule characterizes each of the other nodes in the memory graph, is invoked by an `entry` rule, and also invokes itself recursively. Hence, what we consider to be a shape predicate technically is two predicates; an `entry` predicate and a `p` predicate. The rule schemas for the `entry` and `p` rules are as follows, where † and ‡ indicate potential values at that position (see below) and where numbers in the exponent denote how often such a value is repeated:

$$\begin{aligned} \texttt{entry(}\ \ddagger^{m}\ \texttt{, This, }\ \ddagger^{n}\ \texttt{)} \quad &\texttt{:-} \quad \texttt{field(This, }\ \ddagger\ \texttt{)}^{f}\texttt{, p(}\ \dagger^{p}\ \texttt{)}^{r}\texttt{.}\\ \texttt{p(This, }\ \ddagger^{p-1}\ \texttt{)} \quad &\texttt{:-} \quad \texttt{field(This, }\ \ddagger\ \texttt{)}^{f}\texttt{, p(}\ \dagger^{p}\ \texttt{)}^{r}\texttt{.} \end{aligned}$$

Here, variables m and n stand for the number of parameters in the `entry` rule such that $m+n+1$ is equal to the number of entry pointers. The `entry` rule operates on the first entry pointer if $m = 0$, and on the last entry pointer if $n = 0$. Variable f indicates the number of fields, and `field` indicates the respective field name. We use variable p to denote the number of parameters, including `This` as the first `p` rule parameter. Variable r captures the number of recursive predicate calls. Note that the number of fields f is fixed by the provided type information, and the number of parameters of the `entry` rule is fixed by the input memory graph(s). In contrast, variables p and r remain adjustable to derive rules of varying complexity. While variable p is the same for all `p` rules, variable r may differ. Intuitively, larger values yield more complex and more expressive rules.

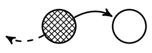
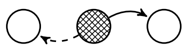
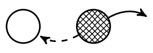
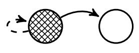
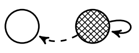
In addition to these constraints on rule structure, symbols † and ‡ express constraints on potential values. The former symbol denotes the atom `null` or an already bound variable; the latter symbol may additionally denote a fresh variable. Observe that the concrete value of fresh, i.e., existentially quantified field variable can be accessed directly from the memory graph without backtracking. Furthermore, rule parameters are existentially quantified, but recursive calls are restricted to already bound variables. Hence, the overall shape predicate can be considered to be existential quantifier free and, thereby, encodes the shape in a constructive style. Also note that the candidate rules form a *test predicate*, i.e., a predicate that evaluates to either true or false but does not compute a value.

Supplying these rule schemas with concrete variable values enables the generation of a finite list of candidate rules. The correctly learned rules for the running example (see Fig. 2(d)) are part of the variable configuration $f = 2$, i.e., fields `next` and `prev`, $p = 2$, and $r \in \{0, 1\}$. We denote fresh parameters with a leading capital `P`, and fresh pointer variables by their upper case field name. Throughout the paper we assume that values of variables within a rule are unique and not equal to the atom `null`, but we leave out explicit inequality clauses to ease readability.

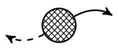
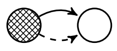
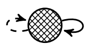
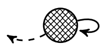
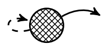
Listing 1. Doubly-linked list (DLL) predicate built from singleton rules

```
entry(This) :- next(This, Next), prev(This, Prev), p(Next).
p(This)     :- next(This, null), prev(This, Prev).
p(This)     :- next(This, Next), prev(This, Prev), p(Next).
```

Table 1. Ten possible abstract nodes for the DLL example (see Fig. 2). Solid/dashed arrows indicate `next`/`prev` pointers; in each case, the crosshatched node indicates the current node.

`next ↦ Var`
`prev ↦ null`
	`next ↦ Var1`
	`prev ↦ Var2`
		`next ↦ null`
		`prev ↦ Var1`
			`next ↦ Var`
			`prev ↦ This`
				`next ↦ This`
				`prev ↦ Var`

`next ↦ null`
`prev ↦ null`
	`next ↦ Var`
	`prev ↦ Var`
		`next ↦ This`
		`prev ↦ This`
			`next ↦ This`
			`prev ↦ null`
				`next ↦ null`
				`prev ↦ This`

2.3 Pruning of Candidate Rules

To prevent the learning of overly simple shape predicates and to increase the performance during the following rule search, pruning techniques are applied which reduce the number of generated candidate rules that are relevant for the rule search.

Singleton rule heuristics. A *singleton rule* contains a singleton variable, i.e., a variable that appears only once within a rule. While such a rule is not per se incorrect, it often yields overly simple shape predicates that ignore important characteristics of the data structure and, thus, our pruning cuts such rules. For example, a singly-linked list (SLL) predicate can be adjusted to also match memory graphs exhibiting a DLL shape by ignoring the value of the `prev` pointer field, i.e., binding it to a new value `Prev` that remains unused. However, such an overly simple shape predicate requires singleton rules (see Listing 1) which are removed via this heuristics.

Static rule analysis. While candidate rules are expressed in Prolog syntax, they are meant to be translated to separation logic later on. Candidate rules are evaluated during rule search according to the semantics of separation logic, i.e., in a resource-like fashion in the sense that the current node `This` is 'consumed' during evaluation and consuming it a second time would yield a resource failure. A simple static analysis, using the following three criteria, is conducted on a per-rule basis to detect rules that always cause a resource failure:

1. *A recursive call with* `null` *as first argument:* Each rule consumes the node referenced by its first parameter, although value `null` does not represent a valid node and consuming it would yield a failure.

2. *A recursive call with* `This` *as first argument:* Because the current rule operates on `This`, the invoked rule yields a failure by consuming `This` a second time.

3. *Recursive calls with the same first argument:* Even if the first call succeeds, the second call fails, because the node on which it operates has already been consumed.

Table 2. Excerpt from the derived list of candidate rules for the running DLL example with complexity $p = 1$; symbols ✗/✓denote that a rule is to be removed/retained wrt. a pruning technique.

Rule	**Singleton rule heuristics**	**Static rule analysis**	**Node abstraction**
`entry(This) :- next(This, Next), prev(This, null), p(Next).`	✓	✓	✓
`p(This)     :- next(This, Next), prev(This, Prev), p(Next), p(Prev).`	✓	✓	✓
`p(This)     :- next(This, null), prev(This, Prev), p(Prev).`	✓	✓	✓
`entry(This) :- next(This, Next), prev(This, null).`	✗$^{\texttt{Next}}$	✓	✓
`entry(This) :- next(This, Next), prev(This, null), p(This).`	✗$^{\texttt{Next}}$	✗^{2}	✓
`p(This)     :- next(This, Next), prev(This, Next), p(Next).`	✓	✓	✗
`p(This)     :- next(This, Next), prev(This, null), p(null).`	✗$^{\texttt{Next}}$	✗^{1}	✗
`p(This)     :- next(This, null), prev(This, Prev), p(Prev), p(Prev).`	✓	✗^{3}	✓

Observe that the third criterion sets an upper bound on the number r of recursive calls within a single rule: r cannot exceed the number of fresh variables present in a particular rule because, otherwise, there exists at least one fresh variable that is used as the same first argument in two different recursive calls. Recall that only parameters and fields may introduce fresh variables, and only the former is adjustable. Hence, the overall complexity for deriving candidate rules solely depends on the number p of rule parameter, with 1 being its initial value. Note that the above rules are sound, but obviously not complete. Hence, the later rule search wrt. input memory graph(s) is necessary.

Node abstraction. A static rule analysis informed by the input memory graph(s) can prune rules whose field clauses do not match any node of the input memory graph(s). The abstraction computes a new value for each field of a node, indicating whether it points to `null`, the current node `This`, or another symbolic node `Var`. While the number of possible abstract nodes grows exponentially wrt. the number of fields, data structures seldomly contain many pointer fields and the number of different abstract nodes is small in practice.

Table 1 depicts the ten possible node abstractions for our DLL example. We use the node abstraction of entry nodes to prune `entry` rules and use node abstractions of the remaining nodes to prune `p` rules. Observe that the four nodes in the supplied memory graph are concrete instantiations of only three different abstract nodes: the abstract node in the first cell of the top row in Table 1 encodes entry node `n1`, and cells 2 and 3 of the top row encode the remaining nodes `n2`/`n3` and `n4`, resp. Overall, seven out of the ten candidate rules are pruned.

Table 2 provides an excerpt of the 228 candidate rules derived for the running DLL example (see Fig. 2) with complexity $p = 1$, together with the outcome of our pruning techniques, where symbols ✗/✓ denotes that a rule is removed/retained. The conflicting static rule analysis criteria and the singleton variable are supplied in the superscripts. Note that a rule is removed if at least one pruning technique yields ✗. Overall, 94/192/181 candidate rules (41%/84%/79%) are pruned via the singleton rule heuristics/static rule analysis/node abstraction; 225 rules (99%) are pruned in total, i.e., all but the three rules shown. The subsequent rule search will reveal that the remaining rules do not contain a subset that characterizes the input memory graph; hence, a complexity of $p = 1$ is insufficient to characterize the running DLL example.

2.4 Rule Search via a Meta-Interpreter

Querying Prolog whether the `entry` predicate holds for the generated candidate rules in the context of the input memory graphs yields either `True` or `False`. In the latter case, rule generation is re-started with increased complexity, i.e., by incrementing parameter p. In the former case, the learned shape predicate contains all rules that have been used during the evaluation and 'proves' that the input memory graph(s) are an instance of the `entry` predicate. However, a standard Prolog interpreter neither reports on utilized rules nor does it allow for a resource-like evaluation. The later translation also requires that the learned predicate does not backtrack. While backtracking on variables is excluded by the avoidance of existentially quantified variables, backtracking on rules is prevented during rule search. These requirements of the evaluation and an additional triviality check are implemented in a meta-interpreter.

The meta-interpreter primarily ensures that (*i*) utilized rules are collected during evaluation and returned as output; (*ii*) all nodes are consumed exactly once; (*iii*) rules that would introduce backtracking fail. To do so, the meta-interpreter attaches state to the evaluation and updates it when encountering particular 'keyword clauses'. A pre-processing step injects these clauses to provide additional information to the meta-interpreter during evaluation. To ensure a resource-like evaluation, clause `node(This)` is prepended to each rule body and triggers the meta-interpreter to label the node denoted by variable `This` as consumed. The evaluation fails if a node has already been consumed, and fails in the end if not all nodes have been consumed. Note that a failure does not terminate the overall evaluation, but forces the meta-interpreter to backtrack, i.e., consider a different rule. In addition, clause `rule(id,guard)` is prepended to each rule body, where `id` is replaced by a value to uniquely identify rules within the list of candidate rules, and `guard` is replaced by a value that groups those rules that share the same guard, i.e., solely differ in their recursive calls. This grouping prevents the learning of predicates that backtrack on rules. When prompted by a `rule` clause, evaluation fails if a different rule with the same guard has previously been applied; otherwise, the rule and guard are added to the current state. Thereby, the search space is exponential in the number of guards with a base linear in the average number of rules per guard. Note that the worst-case complexity is rarely the case, because irrelevant rules and their associated guards are pruned and the remaining rules are not uniformly distributed over the remaining guards.

We already employ the singleton rule heuristics during rule pruning to exclude overly simple rules that ignore available information. In addition, we utilize a triviality check during the rule search, which raises a failure if a rule captures the memory graph in a trivial way: a rule is considered non-trivial if fresh variables of pointer fields point to unconsumed nodes. Thereby, pointer fields pointing to an already consumed node must be equal to a prior pointer field or a parameter. This is achieved by injecting a `fresh` clause after each fresh pointer field during pre-processing and checking that the referenced node is indeed unconsumed during rule search.

Invoking the meta-interpreter `mi` requires a call to the `entry` predicate and a list of consumable nodes as input, and produces a list of utilized rules as output. For the running example, we invoke the meta-interpreter as follows: `?-mi(entry(n1), [n1, n2, n3, n4], Rules)`. The learned rules are shown in Fig. 2(d). We provide an alternative `mi` predicate that exposes the complete state, i.e., the initial and computed value for the nodes, rules, and guards. This enables the learning of a single predicate from *multiple* memory graphs by chaining meta-interpreter calls that pass on their computed state; learning from multiple memory graphs makes our approach more powerful for use in program verification.

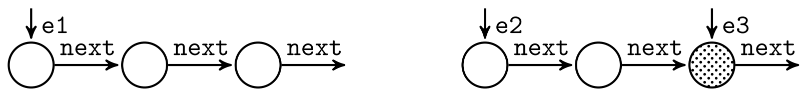


Figure 3. Memory graph exhibiting an SLL shape (`e1`) and an LSEG shape (`e2`, `e3`).

2.5 Translation to Separation Logic

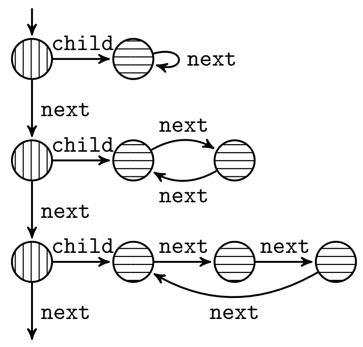
The fact that learned Prolog rules have been derived via a resource-like evaluation and do not backtrack facilitates a straightforward translation to the separation logic dialect of the VeriFast [19] program verifier. Furthermore, the latter enables the construction of predicates that only use a constructive fragment of separation logic and thus allow, e.g., for the synthesis of efficient secure wrappers in [35]. Such predicates are called *precise* in VeriFast terminology [19]. The translated VeriFast predicates consist of three parts: (*i*) the node consumption expressed by a `malloc_block` statement; (*ii*) the field dereference that assigns each field to a symbolic variable via operator "`|->`"; (*iii*) the guarded predicate calls. Each rule is translated to an equivalent guarded predicate call, where the guard consists of the explicit pointer (in)equalities among parameters and pointer fields. A failure is raised if none of the guards matches.

The translated VeriFast shape predicates of the running DLL example are depicted in Fig. 2(e). Note that redundant pointer inequalities may be dropped to ease readability, e.g., the inequality `This != Next` (see Lines 1–4) is redundant wrt. its associated recursive call `p(Next, This)`, because that call would fail when assuming the opposite, i.e., `This == Next`. While we have not used this dropping technique in our benchmark (see Sect. 4), it is potentially interesting for use cases where the learned predicate is inspected or further refined by a developer.

3 Memory Graph Decomposition

The core shape learning algorithm assumes that each input memory graph exhibits a single data structure. However, this assumption does not hold for complex memory graphs where multiple data structures are accessible at the same time or combined via nesting. We reduce the problem of learning a shape predicate from complex memory graphs to multiple instances of learning a shape predicate from a simpler memory graph that exhibits a single data structure. We decompose the input memory graph(s) into sub-graphs, each exhibiting a single unshared data structure, and recompose the individually learned shape predicates to an overall shape predicate that characterizes the input memory graph(s) as a whole. The remainder of this section presents two de-/recomposition techniques for complex memory graphs with separate memory regions and nested data structures.

Separate memory regions. The data structure shapes that are reachable via entry pointers whose reachable nodes do not intersect in any input memory graph, can be learned independently. For example, the entry pointers `e2` and `e3` shown on the right in Fig. 3 share a reachable node, i.e., the node with a dotted fill, whereas entry pointer `e1` on the left does not. The number of sub-graphs is maximized by splitting the entry pointers into the groups {`e1`} and {`e2, e3`}. The former points to an SLL shape, whereas the latter encodes a list segment (LSEG) shape, i.e., a sublist of an SLL characterized by its start and end nodes. The overall shape predicate is the conjunction of the calls to each learned shape predicate for each sub-graph and encoded by a dedicated `entry` rule that characterizes a memory graph when supplied with its present entry pointers as arguments, i.e., `entry(E1, E2, E3) :- sll(E1), lseg(E2, E3)`.



(a) Observed memory graph

```
struct parent {
  struct parent *next;
  struct child  *child;
};

struct child {
  struct child  *next;
};
```

(b) Available type information

```
1  entry(This) :- child(This, C), next(This,    N), p(N), entry_child(C).
2  p(This)     :- child(This, C), next(This, null),       entry_child(C).
3  p(This)     :- child(This, C), next(This,    N), p(N), entry_child(C).
4  entry_child(This) :- next(This, This).
5  entry_child(This) :- next(This, N), p_child(N, This).
6  p_child(This, P1) :- next(This, P1).
7  p_child(This, P1) :- next(This, N), p_child(N,   P1).
```

(c) Combined predicate for the nested data structure in Prolog syntax. Lines 1–3/4–7 refer to the parent/child.

Figure 4. Nesting example involving an SLL as parent and a cyclic SLL as child. Nodes with vertical/horizontal line pattern indicate parent/child nodes.

Nested data structures. Composing data structures via nesting is an important step to model complex data structures. For example, an SLL (parent) may contain a nested cyclic SLL (child) such that each node of the parent points to an *unshared* instance of the child. However, the given type information cannot reveal whether a child is indeed unshared at runtime, nevertheless we use it to identify potential parents, i.e., structs containing a pointer to a different struct. Note that a pointer used for traversing the parent is not required. Thereby, wrapper structs, i.e., a special form of nesting where a struct solely contains pointers to other data structures, are natively handled by this technique. We relax the notion of nesting even further by considering that a parent might use more than one pointer field to describe a child data structure, e.g., a parent using two pointer fields to characterize a child LSEG.

A potential nesting claim is valid if the child pointer of all parent nodes in all input memory graphs points to an unshared child instance. If so, each memory graph is split into sub-graphs, i.e., one sub-graph containing the parent nodes (child pointers are removed) and one sub-graph for each child instance (child pointers become entry pointers). We group the derived sub-graphs according to their role, i.e., parent or child, and learn their shape predicates. Recomposition re-introduces the child pointer and injects a call to the child predicate in each rule of the parent with the value of the child pointer as argument.

The memory graph shown in Fig. 4 exhibits the nesting configuration mentioned and validates `child` as a child pointer. Fig. 4(c) shows the recomposed shape predicate: observe that the corner case of a cyclic SLL with a single node present in the first child sub-graph leads to the learned rule in Line 4; both remaining child sub-graphs learn the same rules (see Lines 5–7).

Table 3. Benchmark results

Name	**Type**	**Memory graph size** (V/E)	**Learned**	**Runtime** (secs)	**Shape**	**Search space** (number of rules)	**Search space** (number of guards)	**Learned rules**	**Parameters**
`treebnh` [3]	`real`	7 / 6	✓	0.13	BT	3	3	3	1
`treeadd` [2]	`real`	15 / 13	✓	0.24	BT $*$ BT	3	3	7	1
`bash-pipe` [1]	`real`	5 / 9	✗	t/o	W $\times$ CDLL	180k	27	–	–
`weiss-sll-cut1` [37]	`tb`	19 / 18	✓	0.09	SLL	3	3	3	1
`weiss-stack-cut1` [37]	`tb`	4 / 3	✓	0.10	SLL	3	3	3	1
`wolf-queue-cut` [40]	`tb`	11 / 10	✓	0.09	SLL	3	3	3	1
`wolf-dll` [40]	`tb`	5 / 8	✓	0.19	DLL	41	13	3	2
`SLL-BT-nesting`	`syn`	12 / 11	✓	0.29	SLL $\times$ BT	12	3	9	1
`SLL-headPtr`	`syn`	7 / 13	✓	0.24	SLL$^{\triangledown}$	38	13	3	2
`SLL-tailPtr`	`syn`	7 / 13	✓	0.43	SLL$^{\triangledown}$	33	12	4	2
`CSLL`	`syn`	3 / 3	✓	0.13	CSLL	12	5	3	2
`CDLL-degen`	`syn`	4 / 7	✓	37.65	CDLL$^{\triangledown}$	700	43	4	3
`BT-parent`	`syn`	5 / 8	✓	5.65	BT$^{\triangledown}$	351	20	4	2
`BT-root`	`syn`	5 / 9	✓	5.23	BT$^{\triangledown}$	351	20	4	2
`two-dlls-direct` [38]	`lit`	5 / 16	✓	1.62	DLL$^{\triangledown}$	72	14	3	2

4 Evaluation

This section outlines the implementation and evaluation of our automatic shape learning approach. We introduce our employed benchmark, discuss general observations and those specific to particular examples, and point out current limitations.

Implementation. We have implemented our approach in the prototype tool `ShaPE` (*Shape Predicate Extractor*). The meta-interpreter is written in SWI-Prolog [39], spanning approx. 100 LOC, and the rest of the tool, e.g., de-/recomposition, rule generation and pruning, and translation to separation logic, is programmed in Python3 [32], spanning approx. 2k LOC. Memory graphs are input as Prolog facts, whereas the learned shape predicates are returned as Prolog rules or translated to VeriFast predicates [19]. VeriFast predicates are also accompanied by a proof witness, i.e., a C program that reconstructs the memory graphs provided as input and invokes a function annotated with the learned predicates as precondition. This function is used in our evaluation to verify that learned predicates do indeed capture the input memory graphs. The source code of the prototype tool and the following benchmark are available online at `https://github.com/uniba-swt/ShaPE` under an open source license.

Benchmark. Since no suitable benchmark is readily available in the literature, we evaluate `ShaPE` on a self-compiled benchmark consisting of real-world (`real`), textbook (`tb`), and synthetic (`syn`) examples, as well as examples from the research literature (`lit`). We consider only two examples of *nested* data structures (`bash-pipe` and `SLL-BT-nesting`), because our bench-

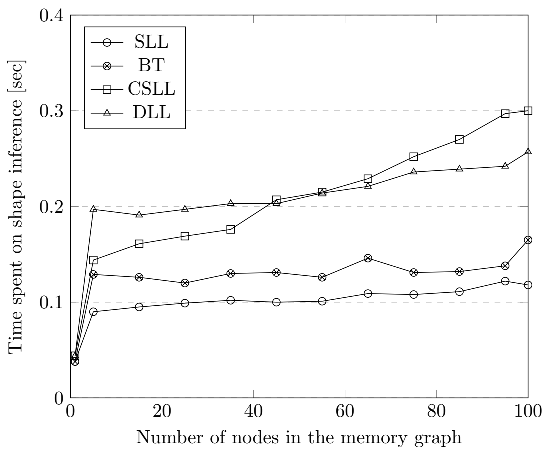


Figure 5. Runtime as a function of memory graph size.

mark is primarily intended to evaluate our core shape learning algorithm; note that nesting detection achieved via the de-/re-composition is *independent* from the underlying data structure shape. Each example comes with a single memory graph that is representative of its data structure in order to demonstrate that our learning approach works even when only one representative memory graph is available. A representative memory graph is captured via the DSI tool [38] at a program location at which the data structure is in a stable shape, e.g., not during insertion or removal operations which typically break a data structure's shape. For our evaluation, we have determined stable states manually, but this could be automated, e.g., via a quiescent periods heuristics as in [16]. Moreover, small memory graphs are sufficient for our learning approach; memory graphs of similar sizes are used in related work, e.g., in [23], where a fixed size of ten nodes is taken.

Our benchmark results are shown in Table 3 and have been obtained on a laptop equipped with a 3.7GHz Intel i7 processor and 32GB of main memory, running Ubuntu 18.04.3 LTS. We encode the shape of each example using typical abbreviations, e.g., BT for binary trees, SLL for singly-linked lists, DLL for doubly-linked lists, and a prefixed 'C' indicates a cyclic shape. Symbol '∇' denotes a variation from the standard shape, e.g., an implementation with an additional pointer field in a struct definition. Standard parent-child nesting is indicated by '$\times$', where a parent shape of 'W' (wrapper struct) means that access to the data structure is wrapped in a separate node. We use symbol '$*$' in example `treeadd` to denote that the memory graph contains separate memory regions characterized by different shapes. For each example, we capture the size of the memory graph in terms of nodes (V) and non-null edges (E), and the time needed to learn a shape (averaged over ten repetitions and given a 5 min timeout, t/o). Additionally, we indicate the search space by the number of pruned candidate rules and the number of guards among which they are distributed. The number of learned rules and the required number of parameters determine a shape's complexity.

General observations. The runtime of our learning algorithm depends on the size of the input memory graph and the complexity of the shape predicate to be learned. However, the complexity dominates the runtime, because one increment in the number of considered predicate parameters exponentially increases the number of candidate rules and guards. In contrast, checking whether a subset of our candidate rules captures the data structure is linear in the size of the memory graph. Fig. 5 shows the runtimes of the learning algorithm for memory graphs of up to one hundred nodes for standard data structures. While the runtimes for shapes with a single parameter (SLL and BT) are almost independent from the number of memory graph nodes, the runtimes for shapes with two parameters (CSLL and DLL) increase linearly.

Also note that the number of candidate rules, guards, and learned rules for data structures without additional pointers such as SLL and BT are identical. This is because in this case, where a single parameter suffices to characterize the shape, the static pruning techniques of `ShaPE` only retain required rules. However, the learned shape for example `SLL-BT-nesting` contains 12 candidate rules overall due to nesting, and example `treeadd` (see Fig. 6(a)) contains three rules per separate memory region and one additional global `entry` rule. Similar to real-world example `treebnh`, these rules do not capture the case where only one pointer points to null, because no such node exists in the memory graph provided as input.

Finally, observe that the learned shape predicates (see, e.g., Figs. 2(d) and 6) are concise, i.e., the least complex non-trivial shape predicate in each case. This is not at least because predicate parameters exchange information relevant to describe the shape, e.g., the expected value of the next node's previous pointer for the DLL example. Therefore, the shape of a data structure is intuitively encoded and does, up to naming, not differ much from what a software developer would have specified manually as the data structure's shape invariant. However, these shape predicates might not directly lend themselves for a use case in formal verification. For example, the manually crafted separation logic DLL predicate in [36] differs from ours in that it considers payload data and is tailored to the verification of a DLL reversal operation.

```
entry(E1, E2) :- entry1(E1), entry2(E2).
entry1(This)  :- left(This, Left), right(This, Right), p1(Left), p1(Right).
p1(This)      :- left(This, null), right(This, null).
p1(This)      :- left(This, Left), right(This, Right), p1(Left), p1(Right).
entry2(This)  :- left(This, Left), right(This, Right), p2(Left), p2(Right).
p2(This)      :- left(This, null), right(This, null).
p2(This)      :- left(This, Left), right(This, Right), p2(Left), p2(Right).
```

(a) Learned shape predicate for example `treeadd`

```
entry(This)            :- next(This, Next), prev(This, Prev), p(Next, This, Prev, This).
p(This, P1, P2  , P1) :- next(This, Next), prev(This,   P1), p(Next, This,   P2,   P1).
p(This, P1, P2  , P3) :- next(This, Next), prev(This,   P1), p(Next, This,   P2,   P3).
p(This, P1, P2  , P3) :- next(This,   P2), prev(This,   P1), p(P2,   This,   P2,   P3).
p(This, P1, This, P3) :- next(This,   P3), prev(This,   P1).
```

(b) CDLL shape predicate for example `bash-pipe`

```
entry(This)     :- next(This, Next), prev(This, null), p(Next, This, This).
p(This, P1, P2) :- next(This,   P2), prev(This,   P1).
p(This, P1, P2) :- next(This, Next), prev(This,   P1), p(Next, This,   P2).
p(This, P1, P1) :- next(This, Next), prev(This,   P1), p(Next, This,   P1).
```

(c) Learned shape predicate for example `CDLL-degen`

Figure 6. Sample shape predicates from the benchmark.

Specific observations. We discuss examples that exhibit implementation variations of standard shapes. In `two-dlls-direct`, list nodes contain duplicate `next` and `prev` pointers, which makes the shape difficult to detect with a predefined DLL predicate but is of no problem for our approach. For this example, `ShaPE`'s node abstraction pruning is key, because it identifies that only three out of the approx. 360 node abstractions need to be considered for the rule search.

In comparison, the shape predicate learned for `wolf-dll` is the standard DLL (see Fig. 2(d)) already used as our running example and exhibits three out of ten possible node abstractions.

In `SLL-head/tailPtr`, each node has an additional pointer to the first/last element of the list. Similarly, in examples `BT-parent/root`, each node includes an additional pointer to the parent node or to the root node. While the number of required parameters of the corresponding shape predicate is identical to examples `SLL-head/tailPtr`, the additional pointer field leads to a larger number of generated candidate rules.

Most interesting is `CDLL-degen`: it captures a CDLL in an unstable shape, where the `next` pointers already form a cycle, but the `prev` pointer of the first list element still points to null. In contrast to the typical CDLL shape, this shape can be characterized completely with only three parameters. Unstable shapes such as this one are not part of any shape predicate library but are needed for proper verification in VeriFast [19], and illustrate the advantage of our generic shape learning approach. For the learned shape predicate (see Fig. 6(c)), parameter `P1` stores the expected previous node and `P2` denotes the first node of the cycle.

Limitations. The real-world example `bash-pipe` employs a CDLL data structure to encode piped commands, where an initial node wraps access to the underlying CDLL via a `head` pointer. The nesting detection feature of `ShaPE` correctly splits the memory graph into two sub-graphs, i.e., containing the single wrapper struct node and the CDLL. Learning a predicate for the former sub-graph is trivial, but `ShaPE` fails to learn a predicate for the latter: four parameters yield approx. 180k pruned candidate rules, which causes a timeout. Indeed, the correct CDLL shape predicate (see Fig. 6(b)) requires four parameters, where `P1` captures the expected previous pointer and `P2`/`P3` the last/first element of the list. Conceptually, however, our approach is sound and complete wrt. our imposed constraints such as the restriction to constructive shape predicates. If desired, `ShaPE` can be supplemented with predefined shape predicates phrased as Prolog rules, and then skips learning if an existing predicate matches a memory graph outright.

Summary. The evaluation shows that learning shape predicates *without* predefined predicates is possible and that the learned predicates are concise and match their expected definitions. Learnable shapes are not restricted to standard data structures, but also include implementation variations employed in system-level code. `ShaPE`'s pruning techniques and the search strategy of the meta-interpreter enable the fast learning of shape predicates with up to three parameters. This is sufficient to capture many interesting data structures such as (cyclic) singly-/doubly-linked lists with parent/tail pointers and binary trees with parent/root pointers.

5 Related Work

Early work on formally specifying dynamic data structures typically employ a declarative approach. *Graph types* [21] utilize routing expressions to encode the relative addresses of additional pointers such as head and tail pointers, whereas `ShaPE` uses parameters to implicitly include the information relevant for additional pointers. While *shape types* [13] use a context-free grammar to encode shape predicates, we restrict the expressiveness of learnable shape predicates to Prolog predicates that do not backtrack and contain no existentially quantified variables, so as to synthesize constructive separation logic shape predicates as required by the wrapper generator in [35]. The remainder of this section focuses on the learning aspect and compares the capabilities of our shape learning tool `ShaPE` to techniques and tools from the domains of program comprehension, formal verification, and inductive logic programming.

Program comprehension. Typical program comprehension tools conduct a dynamic analysis for detecting dynamic data structures. Prominent tools such as *DDT* [20], *HeapDbg* [24], and *DSI* [38] are limited by their usage of predefined shape predicates. DDT employs a sophisticated library to match observed memory graphs against given "shape invariants". HeapDbg conducts a course-grained shape analysis using graph-theoretic notions of, e.g., trees, based on memory snapshots. DSI uses a taxonomy of data structures on top of its memory abstraction, but requires a complete execution trace similar to DDT rather than execution snapshots in the form of memory graphs. While recent work [8] outlined a translation from DSI's data structure descriptions to shape predicates, this translation inherits the mentioned weaknesses.

Formal verification. Early *static* shape analysis tools such as *SpaceInvader* [12, 41] and *SLAyer* [7] conduct a static source code analysis using predefined shape predicates. SpaceInvader is restricted to singly-linked list segments, while SLAyer provides a generalized second-order list segment predicate [5]. Although formerly rooted in separation logic, *Predator* [17] uses symbolic memory graphs to reason about low-level C programs and supports potentially nested or shared lists using predefined predicates. *Infer* [10, 11] employs bi-abduction to prove memory safety properties; however, it only deals with shape properties for simple singly-linked lists data structures. In contrast to `ShaPE`, these tools are limited because they all employ predefined predicates that are hard-coded into the tools and difficult to extend.

The work in [18] augments the static shape analyzer *Forester* [15] with automatic 'box' learning for encoding repetitive graph patterns as forest automata. While no translation from forest automata to separation logic has been proposed, the resulting predicates are unlikely to be constructive, because 'boxes' are not limited to dereferencing the current node. Comparable to `ShaPE`, but different from its employed technique and analysis, *S2* [22] uses static second-order bi-abduction to infer shape predicates. It handles nested data structures and may be supplemented with predefined predicates. Similarly, the technique proposed in [14] infers predicates automatically, but is limited to data structures with a tree-like backbone.

In contrast, *Locust* [9], *Slearner* [30], and *SLING* [23] conduct a *dynamic* analysis supported by predefined shape predicates. Locust interprets the prediction of a separation logic formula as a classification problem on a fixed grammar, and uses symbolic execution in combination with a program verifier to generate positive and negative memory graphs in a refinement loop. The technique lacks generality because the prediction model – currently supporting simple lists and trees – requires manually defined classification features and a large body of training data. For `ShaPE`, a pre-trained model could be used to sort candidate rules according to their likelihood of characterizing the exhibited data structure, with the aim of increasing the performance of the rule search. Slearner uses graph mutation to derive negative examples from observed memory graphs, which provide additional information for a later classification. `ShaPE` natively supports negative inputs by negating the respective meta-interpreter call, although we have currently no use case for this. Finally, SLING matches predefined predicates against the observed memory graphs to construct a shape predicate, and decomposes input memory graphs similar to `ShaPE`.

Inductive logic programming. The learning problem tackled by our work is an instance of the more general problem of inductive reasoning, where in contrast to deductive reasoning, a generalized theory is to be derived on the basis of observed examples. This is closely related to the field of *Inductive Logic Programming* (ILP) [27], which investigates the inductive derivation of first-order clausal theories from observed examples and background knowledge. Indeed, the shape learning problem can be understood as an ILP problem where the input memory graphs constitute positive examples and predefined shape predicates denote background knowledge.

We thus compare our approach to two prominent ILP systems: *Aleph* [34] and *Metagol* [26]. Aleph employs a saturation and reduction technique to learn a predicate from (potentially) only positive examples. However, most ILP systems including Aleph cannot learn multiple predicates at the same time, which is a problem known as predicate invention. Without this feature, only flat tree shapes without additional pointers can be constructed. The meta-interpretive learning technique implemented in Metagol natively supports predicate invention. It fetches predefined higher-order meta-rules wrt. the current goal and stores the meta-substitutions as a predicate. Hence, `ShaPE` can be seen as a meta-interpretive learning system that employs a rule schema to derive candidate rules (cf. meta-rules) of varying complexity, but in contrast to Metagol constructs an explicit search space. Note that both Aleph and Metagol use first-order logic instead of separation logic.

6 Conclusions and Future Work

This paper introduced a novel approach for automatically learning shape predicates in a constructive separation logic from memory graphs that are obtained from a dynamic program analysis. In contrast to related work, our approach does *not* require predefined predicates. Learning trivial predicates is prevented by a fixed rule schema, pruning singleton rules, and a triviality check conducted during rule search. Additionally, the rule complexity present in the search space is increased incrementally such that the learned shape predicate is the least complex and non-trivial instance that characterizes the input memory graph(s). The outcome of our approach's evaluation – using real-world, textbook, and synthetic examples, as well as examples from the research literature – shows that our prototype tool `ShaPE` does learn concise shape predicates for standard data structures and their implementation variations, except for very complex data structures such as cyclic doubly-linked lists that require predicates with at least four parameters, or skip lists with arbitrary length.

While the learned shape predicates can be used, e.g., to fully automatically synthesize secure wrappers [35], we believe that also program comprehension and visualization techniques and static memory safety verification can benefit from our learning approach. For example, `ShaPE` could replace the *Platypus* component in [9] to learn shape invariants for formal verification without the limitations of predefined predicates.

Current limitations and future work. We plan to devise additional static pruning techniques inspired by the triviality check used during rule search, so as to be able to handle even more complex data structures such as cyclic doubly-linked lists. Also the handling of general skip lists is a subject of future research. Additionally, we aim to extend our de-/re-composition approach to enable the learning of shape predicates that include certain existentially quantified variables, e.g., a terminal node in tree-like structures or cyclic lists of lasso shape. Finally, we wish to integrate simple arithmetic reasoning into the learning algorithm for dealing with, e.g., sorted lists. Introducing arithmetic output parameters as well would further increase the expressiveness of our learnable shape predicates, e.g., for capturing balanced trees. Lifting these limitations will help us to apply the learning approach of `ShaPE` to other use cases, especially in the field of formal verification.

Acknowledgments. We thank the anonymous reviewers for their insightful suggestions that have helped to improve the paper's presentation.

References

[1] GNU bash v4.3.30. https://www.gnu.org/software/bash/. Accessed: 17th May 2016.

[2] Olden Benchmark v1.01. http://www.martincarlisle.com/olden.html. Accessed: 17th May 2016.

[3] The Computer Language Benchmarks Game: Binary Tree, contributed by K. Carson. https://benchmarksgame.alioth.debian.org/u64q/program.php?test=binarytrees&lang=gcc&id=1. Accessed: 17th May 2016.

[4] P. Agten, B. Jacobs, and F. Piessens. Sound modular verification of C code executing in an unverified context. In *Principles of Programming Languages (POPL '15)*, pages 581–594. ACM, 2015.

[5] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *Computer Aided Verification (CAV '07)*, volume 4590 of *LNCS*, pages 178–192. Springer, 2007.

[6] J. Berdine, C. Calcagno, and P.W. O'Hearn. A decidable fragment of separation logic. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS '04)*, volume 3328 of *LNCS*, pages 97–109. Springer, 2004.

[7] J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory safety for systems-level code. In *Computer Aided Verification (CAV '11)*, volume 6806 of *LNCS*, pages 178–183. Springer, 2011.

[8] J.H. Boockmann, G. Lüttgen, and J.T. Mühlberg. Generating inductive shape predicates for runtime checking and formal verification. In *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '18)*, volume 11245 of *LNCS*, pages 64–74. Springer, 2018.

[9] M. Brockschmidt, Y. Chen, B. Cook, P. Kohli, S. Krishna, D. Tarlow, and H. Zhu. Learning to verify the heap. Technical Report MSR-TR-2016-17, Microsoft Research, April 2016.

[10] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P.W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium (NFM '15)*, volume 9058 of *LNCS*, pages 3–11. Springer, 2015.

[11] D. Distefano, M. Fähndrich, F. Logozzo, and P.W. O'Hearn. Scaling static analyses at Facebook. *Commun. ACM*, 62(8):62–70, 2019.

[12] D. Distefano, P.W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '15)*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.

[13] P. Fradet and D. Le Métayer. Shape types. In *Principles of Programming Languages (POPL '97)*, pages 27–39. ACM Press, 1997.

[14] B. Guo, N. Vachharajani, and D.I. August. Shape analysis with inductive recursion synthesis. In *Programming Language Design and Implementation (PLDI '07)*, pages 256–265. ACM, 2007.

[15] P. Habermehl, L. Holík, A. Rogalewicz, J. Simácek, and T. Vojnar. Forest automata for verification of heap manipulation. In *Computer Aided Verification (CAV '11)*, volume 6806 of *LNCS*, pages 424–440. Springer, 2011.

[16] I. Haller, A. Slowinska, and H. Bos. Mempick: High-level data structure detection in C/C++ binaries. In *Working Conference on Reverse Engineering (WCRE '13)*, pages 32–41. IEEE Computer Society, 2013.

[17] L. Holík, M. Kotoun, P. Peringer, V. Soková, M. Trtík, and T. Vojnar. Predator shape analysis tool suite. In *Haifa Verification Conference (HVC '16)*, volume 10028 of *LNCS*, pages 202–209, 2016.

[18] L. Holík, O. Lengál, A. Rogalewicz, J. Simácek, and T. Vojnar. Fully automated shape analysis based on forest automata. In *Computer Aided Verification (CAV '13)*, volume 8044 of *LNCS*, pages 740–755. Springer, 2013.

[19] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*

(NFM '11), volume 6617 of *LNCS*, pages 41–55. Springer, 2011.

[20] C. Jung and N. Clark. DDT: Design and evaluation of a dynamic program analysis for optimizing data structure usage. In *Microarchitecture (MICRO '09)*, pages 56–66. ACM, 2009.

[21] N. Klarlund and M.I. Schwartzbach. Graph types. In *Principles of Programming Languages (POPL '93)*, pages 196–205. ACM Press, 1993.

[22] Q.L. Le, C. Gherghina, S. Qin, and W. Chin. Shape analysis via second-order bi-abduction. In *Computer Aided Verification (CAV '14)*, volume 8559 of *LNCS*, pages 52–68. Springer, 2014.

[23] T.C. Le, G. Zheng, and T. Nguyen. SLING: Using dynamic analysis to infer program invariants in separation logic. In *Programming Language Design and Implementation (PLDI '19)*, pages 788–801. ACM, 2019.

[24] M. Marron, C. Sánchez, Z. Su, and M. Fähndrich. Abstracting runtime heaps for program understanding. *IEEE Trans. Software Eng.*, 39(6):774–786, 2013.

[25] F. McKeen, I. Alexandrovich, A. Berenzon, C.V. Rozas, H. Shafi, V. Shanbhogue, and U.R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Hardware and Architectural Support for Security and Privacy (HASP '13)*, page 10. ACM, 2013.

[26] S.H. Muggleton, D. Lin, and A. Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.

[27] S.H. Muggleton and L.D. Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994.

[28] J. Noorman, J. van Bulck, J.T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F.C. Freiling. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Trans. Priv. Secur.*, 20(3):7:1–7:33, 2017.

[29] M.J. Parkinson and G.M. Bierman. Separation logic and abstraction. In *Principles of Programming Languages (POPL '05)*, pages 247–258. ACM, 2005.

[30] L.H. Pham, J. Sun, and Q.L. Le. Compositional verification of heap-manipulating programs through property-guided learning. *CoRR*, abs/1908.10051, 2019.

[31] P. Philippaerts, J.T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. Software verification with VeriFast: Industrial case studies. *Sci. Comput. Program.*, 82:77–97, 2014.

[32] Python Software Foundation. Python language reference, version 3.6. Available at `https://docs.python.org/3.6/reference`. Accessed: 24th October 2019.

[33] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS '02)*, pages 55–74. IEEE Computer Society, 2002.

[34] A. Srinivasan. The ALEPH manual. Comput. Lab. Oxford Univ. `https://www.cs.ox.ac.uk/activities/programinduction/Aleph/aleph.html`. Accessed: 24th October 2019.

[35] N. van Ginkel, R. Strackx, and F. Piessens. Automatically generating secure wrappers for SGX enclaves from separation logic specifications. In *Programming Languages and Systems (APLAS '17)*, pages 105–123. Springer, 2017.

[36] S. van Staden. Full functional correctness proof of doubly-linked list reversal. `https://github.com/verifast/verifast/blob/master/examples/doubly_linked_list.c`. Accessed: 22nd April 2020.

[37] M.A. Weiss. *Data structures and algorithm analysis in C.* Cummings, 1993.

[38] D.H. White, T. Rupprecht, and G. Lüttgen. DSI: An evidence-based approach to identify dynamic data structures in C programs. In *Software Testing and Analysis (ISSTA '16)*, pages 259–269. ACM, 2016.

[39] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *TPLP*, 12(1-2):67–96, 2012.

[40] J. Wolf. C von A bis Z. Galileo Computing, 2009.

[41] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P.W. O'Hearn. Scalable shape analysis for systems code. In *Computer Aided Verification (CAV '08)*, volume 5123 of *LNCS*, pages 385–398. Springer, 2008.