Identifying Dynamic Data Structures by Learning Evolving Patterns in Memory

David H. White and Gerald Lüttgen

Software Technologies Group, University of Bamberg, Germany
{david.white,gerald.luettgen}@swt-bamberg.de

Abstract. We investigate whether dynamic data structures in pointer programs can be identified by analysing program executions only. This paper describes a first step towards solving this problem by applying machine learning and pattern recognition techniques to analyse executions of C programs. By searching for repeating temporal patterns in memory caused by multiple invocations of data-structure operations, we are able to first locate and then identify these operations. Applying a prototypic tool implementing our approach to pointer programs that employ, e.g., lists, queues and stacks, we show that the identified operations can accurately determine the data structures used.

Keywords: Program comprehension, pointer programs, dynamic data structures, machine learning, pattern recognition.

1 Introduction

Programs making heavy use of pointers are notoriously difficult to analyse. To do so one needs to understand which dynamic data structures and associated operations the program employs. Analysis tools for pointer programs, such as those based on shape analysis [17] and pointer graph abstraction [8], rely on an abstraction methodology that must be crafted for each specific data structure, and thus require *a priori* knowledge of the program to be analysed.

Knowing the shape of a data structure is, however, sometimes insufficient for understanding its behaviour. For example, to recognise a linked list implementing a stack, the operations that manipulate the data structure are of key importance. Static analyses typically provide only approximations for this type of behaviour, due to imprecision in the analysis. The task is further complicated when dealing with legacy code, programs with unavailable source code and, even worse, programs with obfuscated semantics such as malware. Hence, the question arises whether pointer programs can be understood, with high confidence via a dynamic analysis that identifies dynamic data structures and the operations that manipulate them from an execution trace of the program under analysis.

Identifying (or in machine-learning terminology: *labelling*) operations appearing in a program trace is a difficult problem. The initial obstacle is simply locating data structure operations, i.e., determining which *events* (e.g., a pointer write) in the trace correspond to an operation and which do not. This problem

is compounded by the fact that invocations of the same operation may look very different: clearly the addresses appearing in pointer variables will differ, but there may also be significant differences in the control path taken due to traversal or corner cases, such as inserting to an empty data structure.

The key idea is to locate an operation by learning the repetition in the program trace caused by multiple invocations of that operation. For this to work, we must construct an abstraction of the trace that lessens the differences between invocations, and thus exposes the repetition. However, we need to make some realistic assumptions for this to be feasible. Firstly, there should be a sufficiently large number of invocations to expose the repetition, and secondly, the surrounding context of the invocations should vary; otherwise, the context could be included in the repeating pattern.

However, merely locating repetition in the program trace is insufficient as it is highly likely that repetition resulting from non-operations will also be discovered. Furthermore, as there are many different ways to code a data structure operation, it is unlikely that it will be possible to assign a label at the granularity of repeating pattern elements. To solve both problems, we consider an instance of a repeating pattern a *potential operation*. We then construct a snapshot of the pre- and post-memory states of the potential operation, and assign a label based on the difference between these. With the set of data structure operations to hand, identifying the program's data structures is an easy task.

Contribution & Approach. Our contribution is the automated identification of dynamic data structures appearing in an execution trace of a C program via a labelling of the operations that manipulate them. We have written a prototypic software tool to evaluate our approach on a number of textbook programs implementing dynamic data structures, in addition to real-world examples. The current prototype employs user-specified templates to identify iterative data structures such as lists, queues, stacks, etc., and has a couple of limitations that should be addressed by future work: nested data structures/operations are not handled and patterns for non-tail recursive operations cannot be learned.

We divide the description of our approach into three parts: Sec. 2 shows how we compute a suitable abstraction from an execution, Sec. 3 introduces the machine learning of repetition, and Sec. 4 explains the labelling process for operations and data structures. We begin Sec. 2 by describing the type of events we wish to capture from an execution, in addition to how the instrumentation is performed. Thus, the execution of the instrumented source code gives an *event trace*. For each event we compute a *points-to graph*, and the sequence of these is the *points-to trace*. However, the points-to trace is unsuitable as input to the machine learning as the specific information about an event is captured very inefficiently. Therefore, we construct a second abstraction for each points-to graph that captures the semantics of the event; using machine learning terminology we term this abstraction a *feature*.

The search for repeating structure takes place on the *feature trace*, where the goal is to learn the set of *patterns* that best captures the repetition (Sec. 3). The notion of "best" is determined by a *Minimum Description Length* [6] criterion

that evaluates how successful a set of patterns is at compressing the feature trace. The search is performed using a genetic algorithm, which is particularly good at finding globally good solutions. Each *occurrence* of a pattern corresponds to a potential operation, and labelling is performed by matching against a repository of *templates* for known data structure operations (Sec. 4). Data structure labelling is then achieved by considering the set of operations that manipulated a connected component in the points-to graph.

Our prototype tool is implemented using a combination of C++, the C Intermediate Language (CIL) [12] and Evolving Objects [4] (11k LOC) and took nine person-months to develop. It is employed to evaluate the effectiveness of our approach at locating and labelling data structure operations written in C (Sec. 5). We first consider data structure source code taken from textbooks [3, 18–20]. We then apply the tool to real-world programs [1, 11, 13]. Finally, in Sec. 6, we discuss related work, give conclusions and describe future work.

2 Trace Generation and Preprocessing

In this section we present the construction of the event trace, points-to trace and feature trace required for our machine learning approach.

Generating the Event Trace. We consider a dynamic data structure to be a set of *objects* (C `structs`) linked by pointers. There are three types of program events that must be captured in the trace: pointer writes, dynamic memory events, and stack pointer variables leaving scope. To record these events during a program's execution, we instrument the source code using the CIL API [12]. We now describe and motivate each event type.

The abstraction must capture the topology of a data structure, and since the topology is defined by pointer writes and their types, this information must be captured in the trace. All pointer-write events have the following attributes: sourceAddr, targetAddr and pointerType. We differentiate between two types of pointer writes: those occurring in the *context* of an encapsulating object, i.e., assignments to `context.ptr` or `context->ptr`, where the `context` is the `struct` in which the pointer appears, and those with no context. If the write has context, then the predicate *hasContext* on the event is true and two additional attributes are set, namely encapsulatingObjectAddr and encapsulatingObjectType.

The deallocation of memory is also key to the abstraction. After a memory region has been deallocated, any information the abstraction was tracking about this region should be disregarded. Attributes for this event type record the beginning and end of the memory region: bFreeAddr and eFreeAddr, respectively. Memory allocations are not recorded as separate events and are instead combined with the pointer write storing the allocation's return value. For writes of this type, the predicate *isAlloc* is true and the attribute allocSize is defined.

We want to understand how the operations affect the data structures beyond the internal modifications; consider removing the front element of a linked list by only updating the head pointer. To identify such modifications we must track the *entry points* to the dynamic data structure. This is simple as we already

```
1  typedef struct node *N_ptr;
2  typedef struct node {
3     int key;
4     N_ptr next, prev;
5  } Node;
6
7  void dllInsertFront(
8     N_ptr *list, int key) {
9     N_ptr new
10       = malloc(sizeof(Node));
11    new->key = key;
12    new->next = *list;
13    new->prev = NULL;
14    if (*list != NULL)
15       (*list)->prev = new;
16    *list = new; }
```

Fig. 1. An operation to insert in the front of a doubly linked list.

record all pointer writes; however, care must be taken if the pointer write is in the stack as this memory has a lifetime defined by its scope. Thus, events of this type store the address of the pointer variable leaving scope in attribute varAddr.

To exemplify our approach, we give a running example based on the insert-front doubly linked list operation in Fig. 1. It executes in one of two *modes*, inserting to the front of an empty list or a non-empty list. Instrumentation will be inserted at lines 9, 12, 13, 15 and 16 to record pointer writes, and after line 16 to record local variables that go out of scope.

Constructing the Points-to Trace. For each event in the event trace $\langle E_1, \ldots, E_n \rangle$, a points-to graph is constructed that describes the effect of that event on the memory state. Points-to graph G_i is constructed by applying event E_i to points-to graph G_{i-1}, where the initial points-to graph is G_0.

A points-to graph $G = (\mathcal{V}, \mathcal{E})$ is composed of a vertex set $\mathcal{V}$ and an edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. There is exactly one of each of the following three special vertices in each points-to graph: v_{null}, the target of null pointers; v_{undef}, the target of undefined pointers; and $v_{\text{disconnect}}$, a vertex with no edges that is used as a placeholder return value. All remaining vertices represent objects, and each has the following set of attributes obtained from the event trace: a beginning address (bAddr), an end address (eAddr) and a type (type). A type t has a set of pointer fields $\{f_1, f_2, ...\} = t.\text{fields}$. A compound variable object may have any number of pointer fields (including zero), while a raw pointer has exactly one; raw pointers are used as entry points to the data structure. Each field has an associated type ($f_i.\text{type}$) and offset ($f_i.\text{offset}$). An edge $e \in \mathcal{E}$ represents a pointer and has a source address attribute (sAddr). We do not require that the source addresses of the out-edges of $v \in \mathcal{V}$ be compatible with the field offsets given by $v.\text{type}$.

The pseudocode in Fig. 2 describes how the points-to graph is updated for an event. A pointer write provides two opportunities for adding information to the points-to graph beyond adding the written pointer. If the write has context, then we can add information about the object encapsulating the pointer, and we may always add information about the target object based on the pointer type. This occurs between lines 2-10 of Fig. 2 and in FINDORADDVERTEX. Now, the vertices representing the source and target objects of the pointer are stored in v_s and v_t, respectively. Using this, the edge representing the pointer is added and

```
1: if isPointerWrite(E) then
2:   if hasContext(E) then
3:     A ← E.encapsulatingObjectAddr; T ← E.encapsulatingObjectType
4:   else
5:     A ← E.sourceAddr; T ← E.pointerType
6:   v_s ← findOrAddVertex(A, T)
7:   if E.targetAddr ≠ NULL then
8:     v_t ← findOrAddVertex(E.targetAddr, deref(E.pointerType))
9:   else
10:    v_t ← v_null
11:  E ← E − {e ∈ E : e.sAddr = E.sourceAddr} ∪ {(v_s, v_t)⟨E.sourceAddr⟩}
12: else
13:  if isMemoryFree(E) then
14:    V_remove ← {v ∈ V : [v.bAddr, v.eAddr) ⊆ [E.bFreeAddr, E.eFreeAddr)}
15:  else if isVarOutOfScope(E) then
16:    V_remove ← {v ∈ V : v.bAddr = E.varAddr}
17:  for all v ∈ V_remove do
18:    E ← E − edges(v) ∪ {(v_s, v_undef)⟨(v_s, v_t).sAddr⟩ : (v_s, v_t) ∈ inEdges(v)}
19:  V ← V − V_remove

20: procedure findOrAddVertex(A : Address, T : Type)
21: if ∃v ∈ V : [A, A + T.size) ⊆ [v.bAddr, v.eAddr]) then return v
22: else
23:  v_new ← createVertex(type = T, bAddr = A, eAddr = A + T.size)
24:  for all v ∈ V − {v_new} : [v.bAddr, v.eAddr) ⊆ [v_new.bAddr, v_new.eAddr) do
25:    E ← E − inEdges(v) ∪ {(v_s, v_new)⟨(v_s, v_t).sAddr⟩ : (v_s, v_t) ∈ inEdges(v)}
26:    E ← E − outEdges(v) ∪ {(v_new, v_t)⟨(v_s, v_t).sAddr⟩ : (v_s, v_t) ∈ outEdges(v)}
27:    V ← V − {v}
28:  forall f ∈ T.fields do
29:    if ∄e ∈ outEdges(v_new) : e.sAddr = A + f.offset then
30:      E ← E ∪ {(v_new, v_undef)⟨A + f.offset⟩}
31:  return v_new
```

Fig. 2. Updating of the points-to graph based on event E.

any pre-existing edge for this pointer is removed (line 11). We use the notation $e\langle \mathrm{A} \rangle$ to initialize the source address attribute of edge e to address A.

findOrAddVertex(A, T) returns the vertex that represents the memory needed by type T starting at address A. If there is no suitable pre-existing vertex, then one is added (line 23). However, there may be pre-existing vertices representing subsections of the region, and any information stored by these vertices must be aggregated into the vertex of the new larger region. This process is performed in lines 24-27 where, for each defunct vertex, in-edges are updated to point to the new vertex, out-edges are added to the new vertex, and lastly, the vertex is removed. Finally, for any field of the new vertex's type that does not already have a pointer, an edge is added from that field to v_{undef} (lines 28-30).

Deallocation and variable-out-of-scope events are handled in lines 13-19. The only distinction is that a deallocation event may remove a set of vertices, while

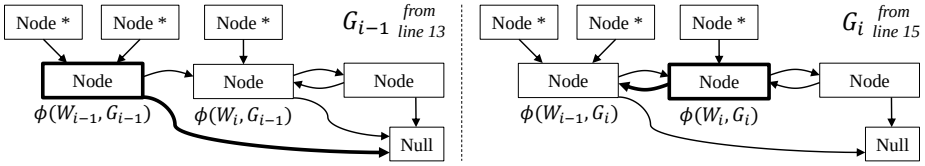

Fig. 3. Points-to graphs generated from the code in Fig. 1. The highlighted pointer is written in the event, and the highlighted vertex is the written vertex (discussed later). The vertices labelled "Node *" are entry points to the data structure.

an out-of-scope event will remove only one vertex. If there were any in-edges to a removed vertex, then the edges' targets are set to v_{undef} (not shown in Fig. 3).

Fig. 3 displays the points-to graphs after the pointer writes on line 13 and 15 of Fig. 1 have been performed on the third call to `dllInsertFront()`.

Constructing the Feature Trace. We construct a *feature trace* $F = \langle F_1, \ldots, F_n \rangle$, where F_i captures the effect of E_i in a way that exposes repetition in the trace. A feature F_i is composed of two types of sub-features: structural sub-features that abstractly describe the change in local topology between G_{i-1} and G_i, and temporal sub-features that capture the relationship between the written pointers in E_{i-1} and E_i.

Let $W_i = E_i.\text{sourceAddr}$ if $isPointerWrite(E_i)$; otherwise, W_i is set to a dummy value that will never be used as an address. We term the vertex in graph G_i that contains address W_i the *written vertex*. In general, a vertex v in a graph G containing address A is computed as follows: $\phi(A, G) = v$ if $\exists v \in \mathcal{V} : A \in [v.bAddr, v.eAddr)$, otherwise $\phi(A, G) = v_{\text{disconnect}}$ (note that there is at most one vertex representing a particular address).

Each structural sub-feature records one aspect of the incoming or outgoing edges of the written vertex. Some sub-features concern the pointer arrangement before the event was performed (i.e., before E_i and calculated on G_{i-1}), and some the arrangement afterwards (i.e., after E_i and calculated on G_i). Further discrimination of pointers is based on the type of the two objects they connect, and whether the pointer is null or undefined. If the event is a memory allocation or free, then additional features are calculated. The full list of features can be seen in Table 1, including example values for the event shown in Fig. 3. The sub-features for out-pointers constructed on the post-state of E_i deserve discussion. Here, additional discrimination is performed based on whether the source and target objects of a pointer have been in the same connected component of the graph (given by COMP) before the event is performed. The rationale behind these sub-features is to capture components being joined or separated.

The first temporal sub-feature records whether the addresses of the written vertices in E_{i-1} and E_i are the same. Sequences of events where this property is true usually represent traversal. The next sub-feature records whether the written vertex in E_i is reachable from E_{i-1} by following one pointer forwards or backwards. The last sub-feature records whether the written vertices in E_i and E_{i-1} are in the same component.

Dynamic Memory Features		*Example*
Allocate	**if** $isAlloc(E_i)$ **then** E_i.allocSize **else** 0	0
Deallocate	**if** $isFree(E_i)$ **then** E_i.freeSize **else** 0	0
Pre and Post Event Structural Features, where $x \in \{pre, post\}$		$(\bowtie)$
In Pointers	$\lvert\{e \in \text{INEDGES}(v_x^i) : \text{SOURCE}(e).\text{type} \bowtie v_x^i.\text{type}\}\rvert$	*pre*: $2(=), 1(\neq)$ *post*: $2(=), 1(\neq)$
Null Pointers	$\lvert\{e \in \text{INEDGES}(v_x^i) : \text{TARGET}(e) = v_{\text{null}}\}\rvert$	*pre*: 1, *post*: 0
Undef Point.	$\lvert\{e \in \text{INEDGES}(v_x^i) : \text{TARGET}(e) = v_{\text{undef}}\}\rvert$	*pre*: 0, *post*: 0
Pre-Event Structural Features		$(\bowtie)$
Out Pointers	$\lvert\{e \in \text{OUTEDGES}(v_{\text{pre}}^i) : \text{TARGET}(e).\text{type} \bowtie v_{\text{pre}}^i.\text{type}\}\rvert$	$1(=), 0(\neq)$
Post-Event Structural Features		$(\bowtie_1, \bowtie_2)$
Out Pointers	$\lvert\{e \in \text{OUTEDGES}(v_{\text{post}}^i) : \text{TARGET}(e).\text{type} \bowtie_1 v_{\text{post}}^i.\text{type} \wedge \text{COMP}(v_{\text{post}}^i) \bowtie_2 \text{COMP}(\phi(\text{TARGET}(e).\text{sAddr}), G_{i-1})\}\rvert$	$2(=,=), 0(=,\neq)$ $0(\neq,=), 0(\neq,\neq)$
Temporal Features		
Same Object	$v_{\text{pre}}^{i-1}.\text{sAddr} = v_{\text{post}}^i.\text{sAddr}$	false
Temporal Features, where $x \in \{pre, post\}$		*pre*, *post*
1 Forward	$\lvert\text{OUTEDGES}(v_x^{i-1}) \cap \text{INEDGES}(v_x^i)\rvert > 0$	true, true
1 Backward	$\lvert\text{OUTEDGES}(v_x^i) \cap \text{INEDGES}(v_x^{i-1})\rvert > 0$	false, true
Component	$\text{COMP}(v_x^i) = \text{COMP}(v_x^{i-1})$	true, true

Table 1. This table describes how the feature vector F_i is computed for event E_i. To save space some rows represent multiple features ($\bowtie \in \{=, \neq\}$). The features are based on properties concerning the written vertex of events E_{i-1} and E_i. We use the following shorthand for the written vertices: $v_{\text{pre}}^j = \phi(W_j, G_{i-1})$ and $v_{\text{post}}^j = \phi(W_j, G_i)$. The right column shows the value of each feature for event E_i depicted in Fig. 1.

The features described above are sufficiently selective to be able to recognise operations on linked lists and trees (cf. Sec. 5). They are also compact enough to enable an efficient machine learning of patterns. In the following, the exact value vector of a feature will be unimportant; thus, we simply denote features by symbols f_a, f_b, etc., where different indices mean that the features differ.

3 Locating Data Structure Operations in the Trace

We describe repetition in the feature trace in terms of a pattern set, where a pattern is sequentially composed of (i) feature sequences and/or (ii) repetitions of feature sequences. This two-level structure allows repetition to be learned. For example, a feature sequence $[f_a, f_b, f_c, f_d, f_c, f_d, f_e]$ might be represented by the pattern $[[f_a, f_b], [f_c, f_d]^+, [f_e]]$, where the middle sequence is allowed to match multiple times. A feature from a pattern and a feature from the trace match if all sub-features have identical values. There is a small caveat due to the temporal features; we do not require a match of any temporal sub-feature for the first feature in a pattern; this is because we do not want to restrict the matching of a pattern based on the preceding context. As discussed earlier, `dllInsertFront`

from Fig. 1 operates in two different modes. Therefore, we would expect these two modes to manifest themselves as two different feature sequences. This is indeed the case as we obtain the two sequences $[f_a, f_b, f_c, f_d]$ and $[f_a, f_e, f_f, f_g, f_h]$. Note that the first feature is identical as the object storing the `malloc` result is disconnected from everything else. The remainder of the sequence diverges due to the differing number of in-/out-pointers to/from the written vertex.

We solve the problem of locating repetition in the feature trace by considering how it may be compressed, the intuition being that the best compression has identified the most repetition. The *Minimum Description Length* [6] (MDL) principle makes this definition precise; it states that the following should be minimized: $L(H) + L(D|H)$, i.e., the length of the hypothesis (the set of patterns chosen to represent the data) summed with the length of the data encoded with the hypothesis. This is a commonly used criteria since it avoids two of the most common pitfalls in machine learning: over-fitting (penalized by the $L(H)$ term) and over-generalizing (penalized by the $L(D|H)$ term).

The MDL criterion determines the *fitness* of a pattern set. We choose a genetic algorithm to explore the space of possible pattern sets as we expect the fitness function to be highly non-continuous. The algorithm proceeds by evolving an initial set of individuals via two operators, *mutation* and *crossover*, until a stopping condition is met. In each generation of the evolution, the fitness of an individual is assessed, and this determines its inclusion in the next generation.

We use a random initialization of the population where each individual is a set of randomly selected patterns. When crossover is applied to a pair of individuals (with probability GA_c), some of the patterns from each are swapped to the other. When mutation is applied to an individual (probability GA_m), a random pattern is selected and one of two operations is applied: (i) the front or back of the pattern is extended or contracted; or (ii) if the pattern contains consecutively repeating subsequences, then these are collapsed into a single instance that is allowed to match multiple times. The front or back of the pattern may only be extended to a feature sequence that occurs in the feature trace. The search terminates when there has been no improvement in the fitness for GA_t generations. Parameters GA_c, GA_m and GA_t are chosen by us as documented in Sec. 5.

4 Labelling Operations and Data Structures

We now determine which potential operations are real data structure operations and then label them, and which are just noise in the trace. The greedy application of the best set of patterns to the feature trace gives the set of potential operations $\mathcal{P}$. An operation $P \in \mathcal{P}$ is a subsequence of the event trace, i.e., $P = \langle E_i, \ldots, E_j \rangle$. Given this definition, $pre(P) = G_{i-1}$ is the points-to graph before the operation was performed and $post(P) = G_j$ is the points-to graph afterwards.

Labelling Operations. We label operations via a template matching scheme, i.e., we manually define a repository of templates $\mathcal{T}$ for the pre and post points-to graphs of any operation we wish to identify, and attempt to match each template in turn. Templates are defined for operations on a singly linked list (SLL), a queue

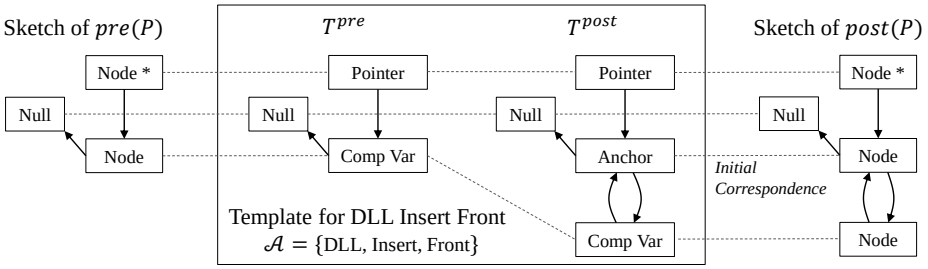

Fig. 4. An example of matching the template for inserting at the front of a doubly linked list. Note that, in T^{pre}, the second pointer to `NULL` is omitted to allow the template to match a DLL insert front operation on a list of any length.

as SLL, a stack as SLL, a doubly linked list (DLL) and a binary tree. There is typically not a 1-1 correspondence between a template match and a real-world data structure operation; so instead of labelling a potential operation directly, a successful match adds a set of attributes to the operation (see below). After a match of all templates has been attempted, the operation label is determined from the identified set of attributes.

Template. A template $(T^{pre}, T^{post}, \mathcal{A}) \in \mathcal{T}$ consists of a pair of template graphs, which are matched against an operation P, and a set of attributes $\mathcal{A}$. A template graph places constraints on which types of template vertices may match which types of points-to graph vertices; we distinguish four types: compound variable vertices, raw pointer vertices (which can be distinguished from compound variable types with one field on the basis of the C types), v_{null} and v_{undef}. One of the template graphs must have more vertices than the other, and the vertex set of the smaller graph is a subset of the larger. Compound variable vertices appearing only in the larger graph are termed *anchor vertices*; at least one of these is always required. An example of a template is given in Fig. 4.

Matching. Anchor vertices are used to provide the initial correspondence(s) for the match. If $|pre(P)| > |post(P)|$ ($|\cdot|$ only counts non-pointer object vertices), then templates with anchor vertices in T^{pre} are applicable for matching; or, if $|post(P)| > |pre(P)|$, then templates with anchor vertices in T^{post} are applicable. Those difference vertices between $pre(P)$ and $post(P)$ that are compound variable vertices provide the set of vertices to be initially mapped to the anchor vertices. With the initial correspondences established, all remaining vertices and edges in the template graph are matched to those in the points-to graph. If this succeeds, and the other template graph can be matched to the other points-to graph given the previous correspondences, then the template is matched. In case there are multiple possible initial mappings to the anchor vertices, all possible permutations are tried. Thus, it does not matter if some of the difference vertices are irrelevant to the operation. Note that our reliance on difference vertices is not a restriction in practice, since all dynamic data structures have some operations (e.g., insert and remove) that exhibit this characteristic and do not have

only, e.g., traversing operations. In Fig. 4, $|post(P)| > |pre(P)|$, and T^{post} has an anchor vertex, so T^{post} is first matched to $post(P)$ using the difference vertex for the initial correspondence. Since this matches, we check whether T^{pre} matches $pre(P)$ given the correspondences from the first step. This also matches, and hence, so does the whole template.

Labelling. After all templates have been tested, we examine the set of present and absent attributes now associated with a potential operation to determine its label. Attributes may record *data structures* (SLL, DLL, bTree), *coding style* (Payload, Null-terminated, Header-node, Sentinel-node, Tail-pointer), *mode* (Insert, Remove) and *position* (Front, Middle, End). A formula over the attribute set allows the potential operation to be labelled, e.g., an operation satisfying $\text{SLL} \wedge \neg\text{DLL} \wedge \neg\text{bTree} \wedge \text{insert} \wedge \neg\text{remove} \wedge \text{front} \wedge \neg\text{middle}$ is labelled SLL Insert Front, and one satisfying $\text{DLL} \wedge \neg\text{bTree} \wedge \text{insert} \wedge \neg\text{remove} \wedge \text{front} \wedge \neg\text{middle}$ is labelled DLL Insert Front. If the set of attributes is not consistent with exactly one predicate, then we report that operation to the user.

Note that the templates for SLL are easy to match on many non-SLL operations. It is only through a combination of templates and attributes that we are able to handle them correctly. Continuing with our previous example (Fig. 4), a template for SLL Insert Front will also match P, but this is ruled out as the final attribute set will only be consistent with the predicate for DLL Insert Front.

Labelling Data Structures. The final part of this phase is to label the data structures that are manipulated by the operations. For each connected component in the graphs, over the whole points-to trace, we check what combination of operations manipulated this component. This is almost always possible since the set of components is typically stable in real-world programs. If the combination of operations is consistent (within a tolerance t_{ops}, see Sec. 5) for a data structure, then this component is labelled. However, if the component has a severely inconsistent set of operations, e.g., equal numbers of Tree Insert and SLL Insert Middle, then this could be an indication of a programming error. Working out correct combinations is non-trivial since, in special circumstances, one data structure can look like another. For example, if there are many operations for both SLL Insert Front and SLL Insert Front with Header, then the label SLL No Header would be preferred.

5 Evaluation

We first evaluate our prototype tool on data structure source code taken from textbooks [3, 18–20] (SLLs, DLLs, queues, stacks, binary trees). These show that our approach recognizes a number of different data structures and works when operations are coded in different styles. We reinforce these conclusions with experiments on correctly mutated SLL and DLL operations (i.e., permuting statements within the operation in a way that does not change the operation's semantics, e.g., by changing the placing of a call to malloc). Next, we demonstrate that our approach can handle program traces that record events for multiple data structures. Lastly, we apply our approach to real-world programs [1, 11, 13]. All

experiments were run under Ubuntu on a modern 16 core PC. The most time consuming step is locating repetition with the genetic algorithm (GA), and this is trivially parallelizable. The largest trace analysed has 15k events and takes 25 minutes (80% spent in GA) and 1GB RAM. GA parameters in Evolving Objects [4] are as follows: $GA_c = 0.1$, $GA_m = 0.1$ and $GA_t = 500$.

Methodology. To evaluate the data structures taken from textbooks, we construct a program for each example to simulate its use in a typical setting. Each program repeatedly chooses an operation applicable to the current state of the data structure to perform. To provide meaningful results, we average the measurements taken over 10 different runs, i.e., simulating the data structure being used in 10 deterministic programs. To make each textbook program more realistic, a randomly chosen "noise" function is sometimes invoked to simulate the program performing other tasks, such as preparing the payload for the data structure. This noise is generated via a set of functions that are indicative of those found in real programs. The noise comprises 30% to 50% of each trace.

When analysing the trace produced by a run, we let $\mathcal{R}$ stand for the set of *real operations*. The set of potential operations that correctly represent a real operation is given by $\mathcal{P}_\mathcal{R} \subseteq \mathcal{P}$. A potential operation has an associated label that is either "NoLabel" or a data structure operation label. Thus, the set of labelled operations is $\mathcal{L} = \{P \in \mathcal{P} : \mathit{label}(P) \neq \text{NoLabel}\}$, and we denote the set of correctly labelled operations by $\mathcal{L}_\mathcal{R} \subseteq \mathcal{L}$.

To evaluate the success of locating repeating patterns, we must compute the set $\mathcal{P}_\mathcal{R}$. This is tricky since potential operations do not need to perfectly map to real operations for the approach to be successful; we consider an overlap of 50% sufficient. Formally, we record that $P \in \mathcal{P}$ is a member of $\mathcal{P}_\mathcal{R}$ if the number of events P has in common (operator $\cap$) with the most appropriate real operation (given by $\psi(P) = \text{argmax}_{R' \in \mathcal{R}}\{|P \cap R'|\}$) is above 0.5. This also enables a definition for the set of *correctly labelled* operations.

$$\mathcal{P}_\mathcal{R} = \{P \in \mathcal{P} : \frac{|P \cap \psi(P)|}{|\psi(P)|} > 0.5\} \quad \mathcal{L}_\mathcal{R} = \{P \in \mathcal{P}_\mathcal{R} : \mathit{label}(P) = \mathit{label}(\psi(P))\}$$

This measure does not penalize potential operations for over-matching a real operation. However, this is only of concern if the user is analysing a program with the labelled operations, and irrelevant parts of the program are included, e.g., if an operation includes noise or part of another operation. We therefore introduce a second measure for the usefulness of a labelled operation to the user, where each summand expresses the proportion of the operation that agrees with the most appropriate real operation, minus the proportion that disagrees:

$$\mathcal{L}_{\text{quality}} = \frac{1}{|\mathcal{L}|} \sum_{L \in \mathcal{L}} \left(\frac{|L \cap \psi(L)|}{|\psi(L)|} - \frac{|L| - |L \cap \psi(L)|}{|L|} \right)$$

We report the following quantities to assess our approach. Firstly, we give $\frac{|\mathcal{P}_\mathcal{R}|}{|\mathcal{R}|}$, the fraction of operations that correctly locate a real operation (this may be

Test Program	$\lvert\mathcal{R}\rvert$	Potential Ops		Labelled Ops				Data Structures		
		$\lvert\mathcal{P}\rvert$	$\frac{\lvert\mathcal{P}_\mathcal{R}\rvert}{\lvert\mathcal{R}\rvert}$	$\frac{\lvert\mathcal{L}_\mathcal{R}\rvert}{\lvert\mathcal{P}_\mathcal{R}\rvert}$	$\frac{\lvert\mathcal{L}_\mathcal{R}\rvert}{\lvert\mathcal{R}\rvert}$	$1-\frac{\lvert\mathcal{L}_\mathcal{R}\rvert}{\lvert\mathcal{L}\rvert}$	$\mathcal{L}_{\text{quality}}$	Label	Cor-rect	% of $\mathcal{L}$ supports
Wolf SLL	200	468.1	0.97	0.94	0.91	0.03	1.00	SLL	✓	100%
Weiss SLL	200	445.4	0.93	0.73	0.69	0.13	0.99	SLL^H	✓	100%
Wolf DLL*	200	450.8	1.00	0.35	0.35	0.57	0.98	DLL	✓	100%
Wolf DLL 2	200	405.2	0.95	0.84	0.80	0.00	1.00	DLL	✓	100%
Wolf Stack	200	529	0.88	0.83	0.73	0.00	0.92	S^H_{FF}	✓	100%
Sedgewick Stack	200	522	0.85	0.79	0.68	0.00	0.93	S_{FF}	✓	100%
Weiss Stack	200	418.1	0.80	0.68	0.56	0.00	0.96	S^H_{FF}	✓	100%
Wolf Queue	200	434.4	0.85	0.77	0.67	0.05	0.95	Q^H_{BF}	✓	95%
Deshpande Tree	300	759.3	1.10	0.61	0.67	0.11	0.91	bTree	✓	91%
SLL Perm	300	395	0.96	0.73	0.70	0.06	0.89	SLL	✓	100%
DLL Perm	300	382	0.95	0.69	0.66	0.08	0.91	DLL	✓	100%
Multiple DSs	800	868.5	1.04	0.70	0.73	0.04	0.96	N/A		
mp3reorg†	111.2	111.8	0.98	0.99	0.97	0.00	0.58	SLL	✓	100%
Acidblood†*	439.8	804.8	0.77	0.71	0.55	0.03	0.67	DLL	✓	100%
Olden Health*	92.9	504.7	0.76	0.51	0.40	0.20	0.68	DLL	✓	86%

Table 2. Results from applying our tool to several programs. Programs marked with (without) † have results averaged over 5 (10) runs. Symbol $*$ denotes a program that is currently at the limits of our approach. Superscript X^H means data structure X uses a header node. For queues (Q) and stacks (S), $I \in \{F, B\}$ (front, back) is the position of inserts in the list, and $D \in \{F, B\}$ is the position of deletes in the subscript X_{ID}.

greater than 1 due to the loose classification of a "correct" potential operation). $|\mathcal{P}_\mathcal{R}|$ imposes an upper bound on the success of the labelling, as we may only label potential operations. Thus, we report $\frac{|\mathcal{L}_\mathcal{R}|}{|\mathcal{P}_\mathcal{R}|}$, which is the quality of the labelling wrt. the potential operations. $\frac{|\mathcal{L}_\mathcal{R}|}{|\mathcal{R}|}$ is the overall success of the approach in identifying operations. The false-positive (FP) rate is given by $1 - \frac{|\mathcal{L}_\mathcal{R}|}{|\mathcal{L}|}$, i.e., the fraction of incorrectly labelled operations. Lastly, we determine the data structure label and report the percentage of $\mathcal{L}$ supporting this choice.

Results. The results for our technique are presented in Table 2. When interpreting these, we must keep in mind our goal, namely to be able to classify data structures based on the operations that manipulate them. Therefore, while the overall number of operations that are correctly labelled is important, the FP rate is just as important. In other words, a low FP rate and a reasonable number of correctly labelled operations gives strong evidence for the label of a data structure. It is important to note that the FP rate is reported *in terms of labelled operations*. For example, in Wolf SLL, on average 91% of operations are correctly labelled, and only 3% of the 91% are false positives.

Our tool identifies the following operation categories: SLL/DLL Insert/Remove Front/Middle/Back and Tree Insert/Remove. The position element is mandatory to identify data structures such as queues and stacks when implemented using lists, as our textbook examples are; thus, we must also identify when a

list uses a header node. For all examples, the type of the data structure being manipulated is correctly inferred. Obviously, there is some overlap between queues, stacks and SLLs; however, by preferring the label of the more restrictive data structure when the evidence is within a tolerance of the other possibilities ($t_{\text{ops}} = 10\%$ in our experiments), the correct label is easily inferred. For example, in Wolf Queue, 100% of operations support SLL and 95% support Q^H_{BF}; since Q^H_{BF} is the more restrictive label and within the tolerance, this label is chosen.

In general, the fraction of operations correctly labelled is high, and the FP rates are low. The FP rates for lists, stacks and queues are all explained by the operation position being incorrectly identified. This is indeed the case for Wolf DLL, where a tail pointer makes the shape symmetric and causes the position to be incorrectly identified. When this experiment is re-run without requiring correct position (Wolf DLL 2), the results are much improved. We discuss solutions to these types of problems in the next section. The only examples to have operations that oppose the chosen labelling are Wolf Queue and Deshpande Tree. As elements are always inserted to the back of the queue, these negative examples arise when inserting to an empty queue, and hence, an Insert Front operation is recognized. For Deshpande Tree, the operations are coded iteratively and, therefore, display many modes of execution; some of the patterns inadequately cover the operations and cause an operation to be identified as an SLL operation.

In SLL Perm and DLL Perm we correctly permute insert and delete operations to check the robustness of our approach against various coding styles. Four variants are tested, and these can all be recognized with a low FP rate.

Program Multiple DSs uses an SLL, a DLL, a cyclic DLL and a binary tree together, where each data structure maintains a sorted set of integers. Repeatedly, the program randomly chooses a data structure, and randomly chooses an insert or delete operation to perform. The recognition rates are high and FP rates are low, showing that the combination of templates and attributes provides good discrimination of operations. The only overlap that occurs is in corner cases, such as inserting into an empty tree or DLL. The different data structures all use the same type (except SLL), so discrimination based on these is impossible. We do not require the position to be correctly identified for this test.

Program mp3reorg [11] is a small open-source program ($\approx$ 450 LOC) for organizing the layout of mp3 files from their ID3 tags. We vary the mp3s in the input directory to obtain multiple runs. The trace contains noise in the form of pointers for handling files, and the list elements have pointer payloads of malloc'ed strings, thus confusing the set of difference vertices. Nevertheless, we achieve very good recognition rates for this program.

Acidblood [1] is a medium-sized open-source program ($\approx$ 5k LOC) implementing an IRC bot. It uses 14 different user-defined `struct`s for servers, commands, users, networking, etc., and some of these represent linked lists. We allow it to connect to a server and then simulate privileged users being randomly added and removed. Traces from this program include much noise in the form of pointer writes for network management. Furthermore, some `struct`s contain many fields, meaning that preparing the payload is a significant portion

of an insert. Our tool can recognize a significant number of the DLL operations correctly, and thus, we can accurately infer the data structure used.

Program Olden Health ($\approx$ 500 LOC) gives the results for the program *health* from the Olden benchmark [13]. This program contains much noise and has nested data structures, so we are operating at the limit of our approach here. However, despite a slightly low recognition rate, the type of the list data structure being used can be correctly determined as a DLL.

This proves that our prototype tool is successful at identifying data structures based on the set of operations that manipulate them. Typically, operation labels have a low FP rate, showing that the probability of a data structure being assigned an incorrect label is small.

6 Related Work & Conclusions

Discovering Data Structures. The shape of a data structure is commonly abstracted by a shape graph, which permits finite representations of unbounded recursive structures. These may be discovered for profiling and optimization [15], detecting abnormal data structure behaviour [9] and constructing program signatures [2]. In contrast, the whole heap is modelled in [14] to capture the dynamic evolution of data structures in Java programs and is used to collect summary statistics, including tree/DAG/cycle classification (a classification similar in scope to the static approach of [5]). However, none of these approaches consider the operations affecting the data structures and, hence, fail to capture dynamic properties such as linked lists implementing queues.

DDT [10] is the closest to our approach and functions by exploiting the coding structure in standard library implementations to identify interface functions for data structures. Invariants are then constructed, describing the observed effects of an interface function, and these are used in turn to classify the data structures. The reliance on well-structured interface functions means the approach is not designed for the customised interfaces appearing in OS/Legacy Software and C programs, or the replicated interfaces that appear due to function inlining. In contrast, our machine learning approach makes fewer assumptions about the structure of the code implementing operations.

Verifying Data Structure Usage. Today, shape analysis [17] is one of the predominate ways to reason about heap pointer structures. This framework is based on static analysis and enables the automated proof of program properties that relate to the shape of data structures on the heap. To configure the framework via custom predicates, some information on the shape of the data structure under analysis must be known *a priori*, although there exists some work on inferring predicates automatically [7]. The situation is similar for the proofs carried out in separation logic [16] and abstraction-based techniques such as [8], where abstractions need to be tailored to the data structures at hand.

Conclusions & Future Work. We presented an approach for learning the data structure operations employed by a pointer program given only an execution trace. Our evaluation on a prototypic implementation showed that the

false-positive rate is low, and thus, the labelled operations can accurately infer the data structures they manipulate. We wish to apply this work to various domains, including automated verification, program comprehension and reverse engineering, and to make our prototype available after having been generalised wrt. nested data structures, non-tail-recursive operations and object code analysis. Last but not least, we wish to thank the anonymous reviewers for their valuable comments and suggestions.

References

1. Acidblood IRC Bot. `freecode.com/projects/acidblood`. Accessed: 30.9.12.
2. A. Cozzie, F. Stratton, H. Xue, and S.T. King. Digging for data structures. In *OSDI*, pages 255–266. USENIX, 2008.
3. P.S. Deshpande and O.G. Kakde. *C & Data Structures.* Charles River, 2004.
4. Evolving Objects (EO). `eodev.sourceforge.net`. Accessed: 30.9.12.
5. R. Ghiya and L.J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL*, pages 1–15. ACM, 1996.
6. P.D. Grünwald. *The Minimum Description Length Principle.* MIT Press, 2007.
7. B. Guo, N. Vachharajani, and D.I. August. Shape analysis with inductive recursion synthesis. In *PLDI*, pages 256–265. ACM, 2007.
8. J. Heinen, T. Noll, and S. Rieger. Juggrnaut: Graph grammar abstraction for unbounded heap structures. *ENTCS*, 266:93–107, 2010.
9. M. Jump and K. S. McKinley. Dynamic shape analysis via degree metrics. In *ISMM*, pages 119–128. ACM, 2009.
10. C. Jung and N. Clark. DDT: design and evaluation of a dynamic program analysis for optimizing data structure usage. In *MICRO*, pages 56–66. ACM, 2009.
11. MP3 File Reorganizer. `sourceforge.net/projects/mp3reorg`. Accessed: 30.9.12.
12. G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, volume 2304 of *LNCS*, pages 213–228, 2002.
13. Olden Benchmark. `www.martincarlisle.com/olden.html`. Accessed: 30.9.12.
14. S. Pheng and C. Verbrugge. Dynamic data structure analysis for Java programs. In *ICPC*, pages 191–201. IEEE, 2006.
15. E. Raman and D.I. August. Recursive data structure profiling. In *MSP*, pages 5–14. ACM, 2005.
16. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE, 2002.
17. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.
18. R. Sedgewick. *Algorithms in C – Parts 1-4 (3rd ed.).* Addison-Wesley, 1998.
19. M.A. Weiss. *Data structures and algorithm analysis in C.* Cummings, 1993.
20. J. Wolf. *C von A bis Z.* Galileo Computing, 2009.