To appear in: CAV 2007.

Shape Analysis for Composite Data Structures

Josh Berdine[†], Cristiano Calcagno[♯], Byron Cook[†], Dino Distefano[‡],
Peter W. O'Hearn[‡], Thomas Wies[⋆], and Hongseok Yang[‡]

† = Microsoft Research, ♯ = Imperial College
‡ = Queen Mary ⋆ = University of Freiburg

Abstract. We propose a shape analysis that adapts to some of the complex composite data structures found in industrial systems-level programs. Examples of such data structures include "cyclic doubly-linked lists of acyclic singly-linked lists", "singly-linked lists of cyclic doubly-linked lists with back-pointers to head nodes", etc. The analysis introduces the use of generic higher-order inductive predicates describing spatial relationships together with a method of synthesizing new parameterized spatial predicates which can be used in combination with the higher-order predicates. In order to evaluate the proposed approach for realistic programs we have performed experiments on examples drawn from device drivers: the analysis proved safety of the data structure manipulation of several routines belonging to an IEEE 1394 (firewire) driver, and also found several previously unknown memory safety bugs.

1 Introduction

Shape analyses are program analyses which aim to be accurate in the presence of deep-heap update. They go beyond aliasing or points-to relationships to infer properties such as whether a variable points to a cyclic or acyclic linked list (*e.g.*, [6, 8, 11, 12]). Unfortunately, today's shape analysis engines fail to support many of the composite data structures used within industrial software. If the input program happens only to use the data structures for which the analysis is defined (usually unnested lists in which the field for forward pointers is specified beforehand), then the analysis is often successful. If, on the other hand, the input program is mutating a complex composite data structure such as a "singly-linked list of structures which each point to five cyclic doubly-linked lists in which each node in the singly-linked list contains a back-pointer to the head of the list" (and furthermore the list types are using a variety of field names for forward/backward pointers), most shape analyses will fail to deliver informative results. Instead, in these cases, the tools typically report false declarations of memory-safety violations when there are none. This is one of the key reasons why shape analysis has to date had only a limited impact on industrial code.

In order to make shape analysis generally applicable to industrial software we need methods by which shape analyses can adapt to the combinations of data structures used within these programs. Towards a solution to this problem, we propose a new shape analysis that dynamically adapts to the types of data structures encountered in systems-level code.

In this paper we make two novel technical contributions. We first propose a new abstract domain which includes a higher-order inductive predicate that specifies a family of linear data structures. We then propose a method that synthesizes new parameterized spatial predicates from old predicates using information found in the abstract states visited during the execution of the analysis. The new predicates can be defined using instances of the inductive predicate in combination with previously synthesized predicates, thus allowing our abstract domain to express a variety of complex data structures.

We have tested our approach on set of small (*i.e.* <100 LOC) examples representative of those found in systems-level code. We have also performed a case study: applying the analysis to data-structure manipulating routines found in a Windows IEEE 1394 (firewire) device driver. Our analysis proved safety of the data structure manipulation in a number of cases, and found several previously unknown memory-safety violations in cases where the analysis failed to prove memory safety.

Related work. A few shape analyses have been defined that can deal with more general forms of nesting. For example, the tool described in [7] infers new inductive data-structure definitions during analysis. Here, we take a different tack. We focus on a single inductive predicate which can be instantiated in multiple ways using higher-order predicates. What is discovered here is the predicates for instantiation. The expressiveness of the two approaches is incomparable. [7] can handle varieties of trees, where the specific abstraction given in this paper cannot. Conversely, our domain supports doubly-linked list segments and lists of cyclic lists with back-pointers, where [7] cannot due to the fact that these data structures require inductive definitions with more than two parameters and the abstract domain of [7] cannot express such definitions.

The parametric shape analysis framework of [9, 16] can in principle describe any finite abstract domain: there must exist some collection of instrumentation predicates that could describe a range of nested structures. Indeed, it could be the case that the work of [10], which uses machine learning to find instrumentation predicates, would be able in principle to infer predicates precise enough for the kinds of examples in this paper. The real question is whether or not the resulting collection of instrumentation predicates would be costly to maintain (whether in TVLA or by other means). There has been preliminary work on instrumentation predicates for composite structures [14], but as far as we are aware it has not been implemented or otherwise evaluated experimentally.

Work on analysis of complex structures using regular model checking includes an example on a list of lists [3]. The encoding scheme in [3] seems capable of describing many of the kinds of structure considered in this paper; again, the pertinent question is about the cost of the subsequent fixed-point calculation. It would be interesting to apply that analysis to a wider range of test programs.

A recent paper [17] also considers a generalized notion of linear data structure. It synthesizes patterns from heap configurations in a way that has some similarities with our predicate discovery method, in particular in generalizing repeated subgraphs with a kind of list structure. However, unlike ours, the abstract domain in [17] does not treat nested data structures such as lists of lists.

2 Synthesized Predicates and General Induction Schemes

The analysis described in this paper fits into the common structure of shape analyses based on abstract interpretation (*e.g.* [15, 16]) in which a fixed-point computation performs *symbolic execution* (*a.k.a.* update) together with *focusing* (*a.k.a.* rearrangement or coercion) to partially concretize abstract heaps and *abstraction* (*a.k.a.* canonicalization or blurring) to aid convergence to a fixed point. In this work we use a representation of abstract states based on separation logic formulæ, building on the methods of [1, 4].

There are two key technical ideas used in our new analysis:

Generic inductive spatial predicates: We define a new abstract domain which uses a higher-order generalization of the list predicates considered in the literature on separation logic.[1] In effect, we propose using a restricted subset of a higher-order version of separation logic [2]. The list predicate used in our analysis, $\mathsf{ls}\ \Lambda\ (x, y, z, w)$, describes a (possibly empty, possibly cyclic, possibly doubly-) linked list segment where each node in the segment itself is a data structure (*e.g.* a singly-linked list of doubly-linked lists) described by Λ. The ls predicate allows us to describe lists of lists or lists of structs of lists, for example, by an appropriate choice of Λ.

Synthesized parameterized non-recursive predicates: The abstraction phase of the analysis, which simplifies the symbolic representations of heaps, in our case is also designed to *discover new predicates* which are then fed as parameters to the higher-order inductive (summary) predicates, thereby triggering further simplifications. It is this predicate discovery aspect that gives our analysis its adaptive flavor.

Example. Fig. 1 shows a heap configuration typical of a Windows device driver. This configuration can be found, for example, in the Windows device driver supporting IEEE 1394 (firewire) devices, `1394DIAG.SYS`. In this figure the pointer `devObj` is a pointer to a *device object*, defined by a Windows kernel structure called `DEVICE_OBJECT`. Each device object has a pointer to a *device extension*, which is used in essence as a method of polymorphism: device drivers declare their own driver-specific device extension type. In the case of `1394DIAG.SYS`, the device extension is named `DEVICE_EXTENSION` and is defined to hold a number of locks, lists, and other data. For simplicity, in Fig. 1 we have depicted only three of the five cyclic doubly-linked lists in `DEVICE_EXTENSION`. Two of the three circular lists contain nested acyclic lists, and the nodes of these two lists have pointers back to the shared header `DEVICE_EXTENSION`. A subtle point is that these nested lists (via `pMdl` or `IsochDescriptor_Mdl`) can be either empty or nonempty. This requires using a Λ in $\mathsf{ls}\ \Lambda\ (x, y, z, w)$ that covers both empty

[1] In this paper we concentrate on varieties of linked list, motivated by problems in device drivers, but the basic ideas might also be applied with other structures.

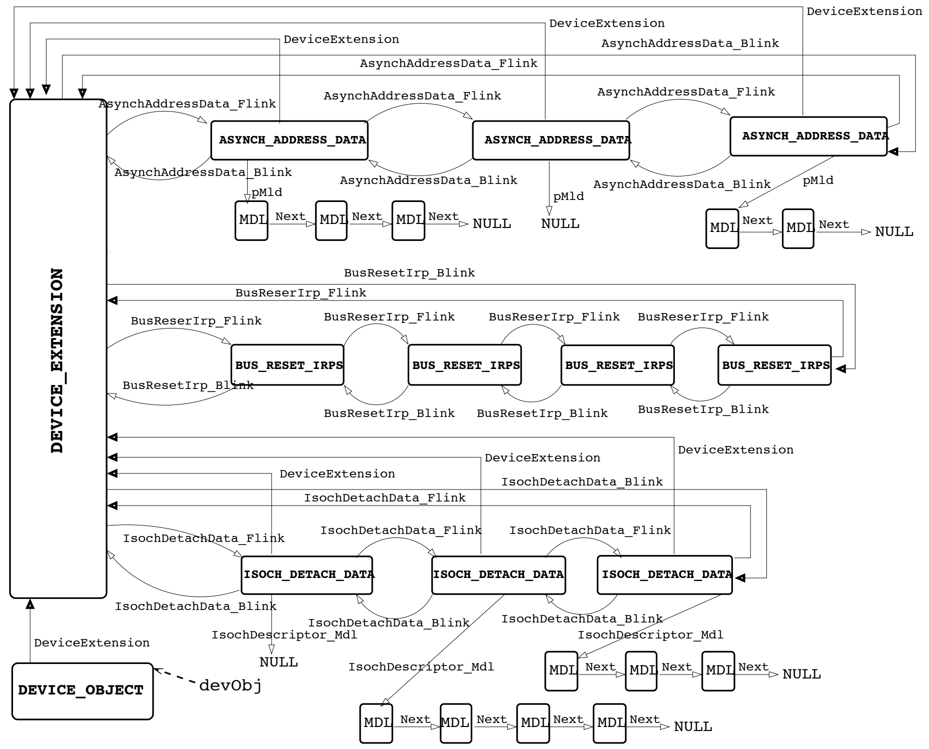


Fig. 1. Device driver-like heap configuration.

and nonempty linked lists; in contrast, when dealing with lists without nesting, it was possible to consider nonempty lists only [4].

There is further nesting that the kernel can see, that we have not depicted in the diagram. Each DEVICE_OBJECT participates in two further linked lists, one a list of all firewire drivers connected to a system, and the other a stack containing various drivers. This yields a “lists of lists of lists” nesting structure. More significantly, since DEVICE_OBJECT nodes participate in different linked lists we have overlapping structures, resulting in “deep sharing” reminiscent of that found in graphs. It is possible to write a logical formula to describe such structures. But, as far as we are aware, a tractable treatment of deep sharing remains an open problem in shape analysis. This paper is no different. Our abstract domain can describe nesting of disjoint sublists, but not overlapping structures. We state this just to be clear about this limitation of our approach.

When the abstraction step from our analysis is applied to the heap in Fig. 1 several predicates are discovered. Consider the singly-linked lists coming out of nodes in the first and third doubly-linked lists. Fig. 1 shows six of those lists, two of which are empty. These lists consist of C structures of type MDL, and they can be described by “$\mathsf{ls}\ \Lambda_{MDL}\ (e', \_, 0, \_)$” for some e'. Here the predicate $\Lambda_{MDL}(x', y', z', w')$ (shown in Fig. 2) is a predicate that takes in four parameters

$$
\begin{aligned}
\mathit{\Lambda}_{MDL} &\triangleq \lambda[x', y', z', w', ()].\, x{=}w' \wedge x' \mapsto \mathtt{MDL(Next\colon}\ z'\mathtt{)}\\
\mathit{\Lambda}_{Async} &\triangleq \lambda[x', y', z', w', (e')].\, x'{=}w' \wedge\\
&\qquad x' \mapsto \mathtt{ASYNC\_ADDRESS\_DATA(AsyncAddressData\_Blink\colon}\ y', \mathtt{AsyncAddressData\_Flink\colon}\ z',\\
&\qquad\qquad\qquad \mathtt{DeviceExtension\colon de, pMdl\colon}\ e'\mathtt{)} \ast \mathsf{ls}\ \mathit{\Lambda}_{MDL}\ (e', \_, 0, \_)\\
\mathit{\Lambda}_{Bus} &\triangleq \lambda[x', y', z', w', ()].\, x'{=}w' \wedge x' {\mapsto} \mathtt{BUS\_RESET\_IRP(BusResetIrp\_Blink\colon}\ y', \mathtt{BusResetIrp\_Flink\colon}\ z'\mathtt{)}\\
\mathit{\Lambda}_{Isoch} &\triangleq \lambda[x', y', z', w', (e')].\, x'{=}w' \wedge\\
&\qquad x' \mapsto \mathtt{ISOCH\_DETACH\_DATA(IsochDetachData\_Blink\colon}\ y', \mathtt{IsochDetachData\_Flink\colon}\ z',\\
&\qquad\qquad\qquad \mathtt{DeviceExtension\colon de, IsochDescriptor\_Mdl\colon}\ e'\mathtt{)} \ast\\
&\qquad \mathsf{ls}\ \mathit{\Lambda}_{MDL}\ (e', \_, 0, \_)\\[1ex]
H &\triangleq \mathtt{devObj} \mapsto \mathtt{DEVICE\_OBJECT(DeviceExtension\colon de)} \ast\\
&\quad \mathtt{de} \mapsto \mathtt{DEVICE\_EXTENSION(AsyncAddressData\_Flink\colon}\ a', \mathtt{AsyncAddressData\_Blink\colon}\ a'',\\
&\qquad\qquad \mathtt{BusResetIrp\_Flink\colon}\ b', \mathtt{BusResetIrp\_Blink\colon}\ b'',\\
&\qquad\qquad \mathtt{IsochDetachData\_Flink\colon}\ i', \mathtt{IsochDetachData\_Blink\colon}\ i''\mathtt{)} \ast\\
&\quad \mathsf{ls}\ \mathit{\Lambda}_{Async}\ (a', \mathtt{de}, \mathtt{de}, a'') \ast \mathsf{ls}\ \mathit{\Lambda}_{Bus}\ (b', \mathtt{de}, \mathtt{de}, b'') \ast \mathsf{ls}\ \mathit{\Lambda}_{Isoch}\ (i', \mathtt{de}, \mathtt{de}, i'')
\end{aligned}
$$

Fig. 2. Parameterized predicates inferred from the heap in Fig. 1, and the result H of abstracting the heap with those predicates. In the predicates $\mathit{\Lambda}_{Async}$ and $\mathit{\Lambda}_{Isoch}$, e' is not a parameter, but an existentially quantified variable inside the body of $\mathit{\Lambda}$.

(as do all parameterized predicates in this work) and then, using $\mapsto$ from separation logic, says that a cell with type `MDL` is allocated at the location pointed to by x', and that the value of the `Next` field is equal to z'.

Next, the three doubly-linked lists are described with further instances of ls, obtained from predicates $\mathit{\Lambda}_{Async}$, $\mathit{\Lambda}_{Bus}$, and $\mathit{\Lambda}_{Isoch}$ in Fig. 2. $\mathit{\Lambda}_{Async}$ and $\mathit{\Lambda}_{Isoch}$ describe nodes which have pointers to the header `de`, and which also point to nested singly-linked lists. Those predicates are built from $\mathit{\Lambda}_{MDL}$, a parameterized predicate for describing singly-linked lists.

The original heap is covered by the separation logic formula H in Fig. 2. The separating conjunction $\ast$ is used to describe three distinct doubly-linked lists which themselves are disjoint from structures `de` and `devObj`. In reading these formulæ, it is crucial to realize that the device extension, `de`, is not one of the nodes in the portion of memory described by any of the three $\ast$-conjuncts at the bottom. For instance, $\mathsf{ls}\ \mathit{\Lambda}_{Async}\ (a', \mathtt{de}, \mathtt{de}, a'')$ describes a "partial" doubly-linked list from a' to a'', with an incoming pointer from `de` to a' and an outgoing pointer from a'' to `de`. Circularity in this case is decomposed into the $\ast$-composition of a single node, `de`, and an acyclic structure. The formula H is more abstract than the beginning heap in that the lengths of the doubly-linked lists and of nested singly-linked lists have been forgotten: this formula is also satisfied by heaps similar to that in Fig. 1 but of different size.

3 Symbolic Heaps with Higher-Order Predicates

We now define the abstract domain of symbolic heaps over which our analysis is defined. Let Var be a finite set of program variables, and Var' be an infinite set of variables disjoint from Var. We use Var' as a source of auxiliary variables to represent quantification, parameters to predicates, etc. Let Fld be a finite set of field names and *Loc* be a set of memory locations.

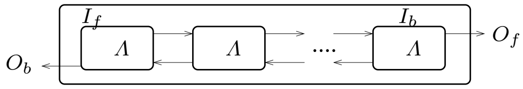
In this paper, we consider the storage model given by $\mathit{Stack} \triangleq (\mathit{Var} \cup \mathit{Var}') \rightarrow \mathit{Val}$, $\mathit{Heap} \triangleq \mathit{Loc} \rightharpoonup_{\mathsf{fin}} (\mathit{Fld} \rightharpoonup \mathit{Val})$, and $\mathit{States} \triangleq \mathit{Stack} \times \mathit{Heap}$. Thus, a state consists of a stack s and a heap h, where the stack s specifies the values of program (non-primed) variables as well as those of auxiliary (primed) variables. In our model, each heap cell stores a whole structure; when $h(l)$ is defined, it is a partial function k where the domain of k specifies the set of fields declared in the structure l, and the action of k specifies the values of those fields.

Our analysis uses symbolic heaps specified by the following grammar:

$x \in \mathit{Var}$	variables
$x' \in \mathit{Var}'$	primed variables
$f \in \mathit{Fld}$	fields
$E ::= x \mid x' \mid \mathsf{nil}$	expressions
$\Pi ::= \mathsf{true} \mid E{=}E \mid E{\neq}E \mid \Pi \wedge \Pi$	pure formulæ
$\Sigma ::= \mathsf{emp} \mid \Sigma * \Sigma \mid E{\mapsto}T(\vec{f}\colon \vec{E}) \mid \mathsf{ls}\ \Lambda\ (E, E, E, E) \mid \mathsf{true}$	spatial formulæ
$H ::= \Pi \wedge \Sigma$	symbolic heaps
$\Lambda ::= \lambda[x', x', x', x', \vec{x'}].\, H$	par. symb. heaps

When $\Lambda = \lambda[x', y', z', w', \vec{v'}].\, H$, we could have written $\Lambda(x', y', z', w', \vec{v'}) = H$. We write $\Lambda[D, E, F, G, \vec{C}]$ for the symbolic heap obtained by instantiating Λ's parameters: $(\lambda[x', y', z', w', \vec{v'}].\, H)[D, E, F, G, \vec{C}] = H[D/x', E/y', F/z', G/w', \vec{C}/\vec{v'}]$.

The predicate "$\mathsf{ls}\ \Lambda\ (I_f, O_b, O_f, I_b)$" represents a segment of a (generic) doubly-linked list, where the shape of each node in the list is described by the first parameter Λ (*i.e.*, each node satisfies this parameter), and some links between this segment and the rest of the heap are specified by the other parameters. Parameters I_f (the *forward input link*) and I_b (the *backward input link*) denote the (externally visible) memory locations of the first and last nodes of the list segment. The analysis maintains the links from the outside to these exposed cells, so that the links can be used, say, to traverse the segment. Usually, I_f denotes the address of the "root" of a data structure representing the first node, such as the head of a singly-linked list. The common use of I_b is similar. Parameters O_b (called *backward output link*) and O_f (called *forward output link*) represent links from (the first and last nodes of) the list segment to the outside, which the analysis decides to maintain. Pictorially this can be viewed as:



When lists are cyclic, we will have $O_f{=}I_f$ and $O_b{=}I_b$.

Generalized ls. The formal definition of ls is given as follows. For a parameterized symbolic heap Λ, $\mathsf{ls}\ \Lambda\ (I_f, O_b, O_f, I_b)$ is the least predicate that holds iff

$$(I_f = O_f \wedge I_b = O_b \wedge \mathsf{emp}) \vee (\exists x', y', \vec{z'}.\ (\Lambda[I_f, O_b, x', y', \vec{z'}]) * \mathsf{ls}\ \Lambda\ (x', y', O_f, I_b))$$

where $x', y', \vec{z'}$ are chosen fresh. A list segment is empty, or it consists of a node described by an instantiation of Λ and a tail satisfying $\mathsf{ls}\ \Lambda\ (x', y', O_f, I_b)$. Note that Λ is allowed to have free primed or non-primed variables. They are used to express the links from the nodes that are targeted for the same address, such as head pointers common to every element of the list.

Examples. The generic list predicate can express a variety of data structures:

- When Λ_s is $\lambda[x', y', z', x', ()].\ (x' \mapsto \mathtt{Node}(\mathtt{Next}\colon z'))$ then the symbolic heap $\mathsf{ls}\ \Lambda_s\ (x, y', z, w')$ describes a standard singly-linked list segment from x to z. (Here note how we use the syntactic shorthand of including x' twice in the parameters instead of adding an equality to the predicate body.)
- A standard doubly-linked list segment is expressed by $\mathsf{ls}\ \Lambda_d\ (x, y, z, w)$ when Λ_d is $\lambda[x', y', z', x', ()].\ x' \mapsto \mathtt{DNode}(\mathtt{Blink}\colon y', \mathtt{Flink}\colon z')$.
- If Λ_h is $\lambda[x', y', z', x', ()].\ x' \mapsto \mathtt{HNode}(\mathtt{Next}\colon z', \mathtt{Head}\colon k)$, the symbolic heap $\mathsf{ls}\ \Lambda_h\ (x, y', \mathsf{nil}, w')$ expresses a nil-terminated singly-linked list x where each element has a head pointer to location k.
- Finally, when Λ is

$$\begin{aligned}&\lambda[x', y', z', x', (v', u')].\\ &\quad x' \mapsto \mathtt{SDNode}(\mathtt{Next}\colon z', \mathtt{Blink}\colon u', \mathtt{Flink}\colon v') * \mathsf{ls}\ \Lambda_d\ (v', x', x', u')\end{aligned}$$

  then $\mathsf{ls}\ \Lambda\ (x, y', \mathsf{nil}, w')$ describes a singly-linked list of cyclic doubly-linked lists where each singly-linked list node is the sentinel node of the cyclic doubly-linked list.

Abstract domain. Let $\mathsf{FV}(X)$ be the non-primed variables occurring in X and $\mathsf{FV}'(X)$ be the primed variables. Let $\mathsf{close}(H)$ be an operation which existentially quantifies all the free primed variables in H (*i.e.* $\mathsf{close}(H) \triangleq \exists \mathsf{FV}'(H).\, H$). We use $\vDash$ to mean semantic entailment (*i.e.* that any concrete state satisfying the antecedent also satisfies the consequent). The meaning of a symbolic heap H (*i.e.* set of concrete states H represents) is defined to be the set of states that satisfy $\mathsf{close}(H)$ in the standard semantics [13]. Our analysis assumes a sound theorem prover $\vdash$, where $H \vdash H'$ implies $H \vDash \mathsf{close}(H')$. The abstract domain $\mathcal{D}^\#$ of our analysis is given by: $\mathcal{SH} \triangleq \{H \mid H \nvdash \mathsf{false}\}$ and $\mathcal{D}^\# \triangleq \mathcal{P}(\mathcal{SH})^\top$. That is, the abstract domain is the powerset of symbolic heaps with the usual subset order, extended with an additional greatest element $\top$ (indicating a memory-safety violation such as a double disposal). Semantic entailment $\vDash$ can be lifted to $\mathcal{D}^\#$ as follows: $d \vDash d'$ if d' is $\top$, or if neither d nor d' is $\top$ and any concrete state that satisfies the (semantic) disjunction $\bigvee d$ also satisfies $\bigvee d'$.

4 Canonicalization

As is standard, our shape analysis computes an invariant assertion for each program point expressed by an element of the abstract domain. This computation is accomplished via fixed-point iteration of an abstract post operator that over-approximates the concrete semantics of the program.

Define $\mathsf{spatial}(\Pi \wedge \Sigma)$ to be Σ.

$$\frac{}{E{=}x' \wedge H \;\rightsquigarrow\; H[E/x']}\ \textsf{(Equality)} \qquad \frac{x' \notin \mathsf{FV}'(\mathsf{spatial}(H))}{E{\neq}x' \wedge H \;\rightsquigarrow\; H}\ \textsf{(Disequality)}$$

$$\frac{\mathsf{FV}(I_f, I_b) = \emptyset \qquad \mathsf{FV}'(I_f, I_b) \cap \mathsf{FV}'(\mathsf{spatial}(H)) = \emptyset}{H * \mathsf{ls}\ \Lambda\ (I_f, O_b, O_f, I_b) \;\rightsquigarrow\; H * \mathsf{true}}\ \textsf{(Junk 1)}$$

$$\frac{\mathsf{FV}(E) = \emptyset \qquad \mathsf{FV}'(E) \cap \mathsf{FV}'(\mathsf{spatial}(H)) = \emptyset}{H * (E \mapsto T(\vec{f}\colon \vec{E})) \;\rightsquigarrow\; H * \mathsf{true}}\ \textsf{(Junk 2)}$$

$$\frac{H_0 \;\vdash\; H_1 * \mathsf{ls}\ \Lambda\ (I_f, O_b, x', y') \wedge I_f \neq x' \qquad \{x', y'\} \cap \mathsf{FV}'(\mathsf{spatial}(H_1)) \subseteq \{I_f, I_b\}}{H_0 * \mathsf{ls}\ \Lambda\ (x', y', O_f, I_b) \;\rightsquigarrow\; H_1 * \mathsf{ls}\ \Lambda\ (I_f, O_b, O_f, I_b)}\ \textsf{(Append Left)}$$

$$\frac{H_0 \;\vdash\; H_1 * \mathsf{ls}\ \Lambda\ (x', y', O_f, I_b) \wedge x' \neq O_f \qquad \{x', y'\} \cap \mathsf{FV}'(\mathsf{spatial}(H_1)) \subseteq \{I_f, I_b\}}{H_0 * \mathsf{ls}\ \Lambda\ (I_f, O_b, x', y') \;\rightsquigarrow\; H_1 * \mathsf{ls}\ \Lambda\ (I_f, O_b, O_f, I_b)}\ \textsf{(Append Right)}$$

$$\frac{\begin{array}{c}\Lambda \in \mathsf{Preds}(H_0) \qquad H_0 \vdash H_1 * \Lambda[I_f, O_b, x', y', \vec{u'}] * \Lambda[x', y', O_f, I_b, \vec{v'}] \\ (\{x', y'\} \cup \vec{u'} \cup \vec{v'}) \cap \mathsf{FV}'(\mathsf{spatial}(H)) \subseteq \{I_f, I_b\}\end{array}}{H_0 \;\rightsquigarrow\; H_1 * \mathsf{ls}\ \Lambda\ (I_f, O_b, O_f, I_b)}\ \textsf{(Predicate Intro)}$$

Fig. 3. Rules for Canonicalization.

The abstract semantics consists of three phases: materialization, execution, and canonicalization. That is, the abstract post $[\![C]\!]$ for some loop-free concrete command C is given by the composition $\mathit{materialize}_C$; $\mathit{execute}_C$; $\mathit{canonicalize}$. First, $\mathit{materialize}_C$ partially concretizes an abstract state into a set of abstract states such that, in each, the footprint of C (that portion of the heap that C may access) is concrete. Then, $\mathit{execute}_C$ is the pointwise lift of symbolically executing each abstract state individually. Finally, *canonicalize* abstracts each abstract state in effort to help the analysis find a fixed point.

The materialization and execution operations of [1, 4] are easily modified for our setting. In contrast, the canonicalization operator for our abstract domain significantly departs from [4] and forms the crux of our analysis. We describe it in the remainder of this section.

Canonicalization performs a form of over-approximation by soundly removing some information from a given symbolic heap. It is defined by the rewriting rules ($\rightsquigarrow$) in Fig. 3. Canonicalization applies those rewriting rules to a given symbolic heap according to a specific strategy until no rules apply; the resulting symbolic heap is called *canonical*.

The AppendLeft and AppendRight rules (for the two ends of a list) roll up the inductive predicate, thereby building new lists by appending one list onto another. Note that the appended lists may be single nodes (*i.e.* singleton lists). Crucially, in each case we should be able to use the same parameterized predicate Λ to describe both of the to-be-merged entities: The canonicalization rules build homogeneous lists of Λ's. The variable side-conditions on the rules are necessary for precision but not soundness; they prevent the rules from firing too often.

The Predicate Intro rule from Fig. 3 represents a novel aspect of our canonicalization procedure. It requires a predicate Λ that can be used to describe similar portions of heap, and two appropriately connected Λ nodes are removed from the symbolic heap and replaced with an $\mathsf{ls}\ \Lambda$ formula. The function Preds in the rule takes a symbolic heap as an argument and returns a set of predicates Λ. It is a parameter of our analysis. One possible choice for Preds is "fixed abstraction", where a fixed finite collection of predicates is given beforehand, and Λ is drawn from that fixed collection. Another approach is to consider an "adaptive abstraction", where the predicates Λ are inferred by scrutinizing the linking structure in symbolic heaps encountered during analysis. There is a tradeoff here: the fixed abstraction is simpler and can be effective, but requires more input from the user. We describe an approach to adaptive abstraction in the next section.

There is one further issue to consider in implementing the Predicate Intro rule. The first has to do with the entailment $H_0 \vdash -$ in the premise of the rule. We require a *frame inferring theorem prover* [1]—a prover for entailments $H_0 \vdash H_1 * H_2$ where only H_0 and H_2 are given and H_1 is inferred. While the aim of a frame inferring theorem prover is to find a decomposition of H_0 into H_1 and H_2 such that the entailment holds, frame inference should just decompose the formula, not weaken it (or else frame inference could always return $H_1 = \mathsf{true}$). So for a call to frame inference, we not only require the entailment to hold, but also require that there exists a disjoint extension of the heap satisfying H_2, and the extended heap satisfies H_0.[2]

There is a progress measure for the rewrite rules, so $\rightsquigarrow$ is strongly normalizing. The crucial fact underlying soundness is that all canonicalization rules correspond to true implications in separation logic, *i.e.* we have that $H \vDash H'$ whenever $H \rightsquigarrow H'$. This means that however we choose to apply the rules, we will always maintain soundness of the analysis. In particular, soundness is independent of the choice of the Preds function used in the Predicate Intro rule.

There are two sources of nondeterminism in the $\rightsquigarrow$ relation: the choice of order in which rules are applied, and the choice of which Λ to use in the Predicate Intro rule. The latter appears to be much more significant in practice than the former. In the implementation we have used a deterministic reduction strategy with no backtracking. But changes in the strategy for choosing Λ can have a dramatic impact on the performance and precision of the analysis algorithm.

5 Predicate Discovery

We now give a particular specification of the Preds function in the (Predicate Intro) rule, based on the idea of similar repeated subgraphs. We emphasize that the graph view of a symbolic heap is intuitive but does not need semantic analysis here: as we indicated above, soundness of the analysis is independent of Preds. We are just describing one particular instance of Preds, which might be viewed as an heuristic constraint on the choice of new predicates.

The idea is to treat the spatial part of a symbolic heap H as a graph, where each atomic $*$-conjunct in H becomes a node in the graph; for instance,

[2] Different strengths of prover $\vdash$ can be considered. A weak one would essentially just do graph decomposition for frame inference.

$E \mapsto T(\vec{f}\colon \vec{E})$ becomes a node E with outgoing edges $\vec{E}$. The algorithm starts by looking for nodes that are connected together by some fields, in a way that they can in principle become the forward and/or backward links of a list. Call these potential candidates root nodes, say E_l and E_r. Once root nodes are found, the procedure $\mathsf{Preds}(H)$ traverses the graph from E_l as well as from E_r simultaneously, and checks whether those two traverses can produce two disjoint isomorphic subgraphs. The shape defined by these subparts is then generalized to give the definition of the general pattern of their shape which provides the definition of the newly discovered predicate Λ. $\mathsf{Preds}(H)$ returns the candidate heaps for use in the ($\mathsf{Predicate\ Intro}$) rule.

```
$\mathsf{discover}(H\colon \text{symbolic heap})\colon \text{predicate} =$
  **let** $\Sigma = \mathsf{spatial}(H)$
  **let** $\Sigma_\Lambda = \mathsf{emp}$
  **let** $I = \emptyset$: set of expression pairs
  **let** $C = \emptyset$: multiset of expression pairs
  **choose** $(E_l, E_r) \in \{(E_l, E_r) \mid \Sigma = E_l{\mapsto}f\colon E_r * E_r{\mapsto}f\colon E * \Sigma'\}$
  **let** $W = \{(E_l, E_r)\}$: multiset of expression pairs
  **do**
    **choose** $(E_0, E_1) \in W$
    **if** $E_0 \neq E_1$ **then**
      **if** $(E_0, E_1) \notin C \wedge E_0 \notin rng(I) \wedge E_1 \notin dom(I)$ **then**
        **if** $\Sigma \vdash P(E_0, \vec{F_0}) * P(E_1, \vec{F_1}) * \Sigma'$ **then**
          $W := W \cup \{(F_{0,0}, F_{1,0}), \ldots, (F_{0,n}, F_{1,n})\}$
          $I := I \cup \{(E_0, E_1)\}$
          $\Sigma := \Sigma'$
          $\Sigma_\Lambda := \Sigma_\Lambda * P(E_0, \vec{F_0})$
        **else fail**
      $C := C \cup \{(E_0, E_1)\}$
    $W := W - \{(E_0, E_1)\}$
  **until** $W = \emptyset$
  **let** $\vec{I_f}, \vec{O_f} = [(E, F) \mid \exists G.\, (F, G) \in C \wedge (E, F) \in I]$
  **let** $\vec{I_b}, \vec{O_b} = [(E, F) \mid \exists G.\, (F, E) \in C \wedge (E, G) \in I]$
  **let** $\vec{x'} = \mathsf{FV}'(\Sigma_\Lambda) - \mathsf{FV}'(\vec{I_f}, \vec{O_f}, \vec{I_b}, \vec{O_b})$
  **return** $(\lambda(\vec{I_f}, \vec{O_b}, \vec{O_f}, \vec{I_b}, \vec{x'}).\, \Sigma_\Lambda)$
```

Fig. 4. Predicate discovery algorithm, where $\mathsf{Preds}(H) = \{P \mid P = \mathsf{discover}(H)\}$

Fig. 4 shows the pseudocode for the discovery algorithm. So far we have, in the interest of clarity, dealt with Λ's with parameters such as $x', y', z', w', \vec{v}'$, however in this section we admit that the analysis actually treats the more general situation where there are multiple links between nodes, and so predicates take parameters $\vec{x}', \vec{y}', \vec{z}', \vec{w}', \vec{v}'$. The algorithm is expressed as a nondeterministic function, using **choose** twice. Preds then collects the set of all possible results, for instance by enumerating through the nondeterministic choices. The set I denotes the subgraph isomorphism between the already traversed subgraphs reachable from the chosen root nodes. The algorithm ensures that the two traverses are disjoint. Here $dom(I)$ denotes the projection of I to the left traverse starting from root node E_l, respectively $rng(I)$ denotes the right traverse starting from

Input symbolic heap
$H = x'_0 \mapsto T(f\colon x'_1, g\colon y'_0) * x'_1 \mapsto T(f\colon x'_2, g\colon y'_1) * x'_2 \mapsto T(f\colon x'_3, g\colon y'_2) *$ $\mathsf{ls}\ \Lambda_1\ (y'_0, \mathsf{nil}, z'_1, \mathsf{nil}) * y'_1 \mapsto S(f\colon \mathsf{nil}, b\colon x'_0) * \mathsf{ls}\ \Lambda_1\ (y'_2, x'_1, z'_2, \mathsf{nil})$ where $\Lambda_1 = (\lambda(x'_1, x'_0, x'_2, x'_1).\, x'_1 \mapsto S(f\colon x'_2, b\colon x'_0))$

#Iters	**W**	**C**	**I**	$\mathbf{\Sigma_\Lambda}$
0	$\{(x'_1, x'_2)\}$	$\emptyset$	$\emptyset$	emp
1	$\{(x'_2, x'_3), (y'_1, y'_2)\}$	$\{(x'_1, x'_2)\}$	$\{(x'_1, x'_2)\}$	$x'_1 \mapsto T(f\colon x'_2, g\colon y'_1)$
2	$\{(y'_1, y'_2)\}$	$\{(x'_1, x'_2), (x'_2, x'_3)\}$	$\{(x'_1, x'_2)\}$	$x'_1 \mapsto T(f\colon x'_2, g\colon y'_1)$
3	$\{(x'_0, x'_1)\}$	$\{(x'_1, x'_2), (x'_2, x'_3), (y'_1, y'_2)\}$	$\{(x'_1, x'_2), (y'_1, y'_2)\}$	$x'_1 \mapsto T(f\colon x'_2, g\colon y'_1) *$ $\mathsf{ls}\ \Lambda_1\ (y'_1, x'_0, z'_1, \mathsf{nil})$
4	$\emptyset$	$\{(x'_1, x'_2), (x'_2, x'_3), (y'_1, y'_2), (x'_0, x'_1)\}$	$\{(x'_1, x'_2), (y'_1, y'_2)\}$	$x_1 \mapsto T(f\colon x'_2, g\colon y'_1) *$ $\mathsf{ls}\ \Lambda_1\ (y'_1, x'_0, z'_1, \mathsf{nil})$

Discovered predicate
$\lambda(x'_1, x'_0, x'_2, x'_1, (y'_1, z'_1)).\, x'_1 \mapsto T(f\colon x'_2, g\colon y'_1) * \mathsf{ls}\ \Lambda_1\ (y'_1, x'_0, z'_1, \mathsf{nil})$

Table 1. Example run of discovery algorithm

E_r. The set C marks how often each pair of nodes is reachable from the two root nodes. It is used for cycle detection and ensures termination of the traversal.

Whenever a new pair of nodes E_0, E_1 in the graph is discovered, the algorithm needs to check whether they actually correspond to $*$-conjuncts of the same shape. The simplest solution would be to check for syntactic equality. Unfortunately, this makes the discovery heuristic rather weak, *e.g.* we would not be able to discover the list of lists predicate from a list where the sublists are alternating between proper list segments and singleton instances of the sublist predicate. Instead of syntactic equality our algorithm therefore uses the theorem prover to check that the two nodes have the same shape. If they are not syntactically equal, then the theorem prover tries to generalize it via frame inference:

$$\Sigma \vdash P(E_0, \vec{F_0}) * P(E_1, \vec{F_1}) * \Sigma' \ .$$

Here the predicate $P(E, \vec{F})$ stands for either a points-to predicate or a list segment $\mathsf{ls}\ \Lambda\ (\vec{I_f}, \vec{O_b}, \vec{O_f}, \vec{I_b})$ where $E \in \vec{I_f} \cup \vec{I_b}$ and $\vec{F} = \vec{O_f}, \vec{O_b}$. The generalized shape $P(E_0, \vec{F_0})$ of the node in the left traverse then contributes to the spatial part of the discovered predicate.

Once the body of the predicate is complete the parameter list is constructed to determine forward and backward links between instances of the predicate. The forward and backward links between the two traverses are encoded in sets I and C: *e.g.* if for a pair of nodes $(F, G) \in C$ we have that F is in the right traverse then there is a forward link going from the left traverse to node F. Thus F is an outgoing forward link and the node E which is isomorphic to F is the corresponding input link into the left traverse. If a pair (E, F) is reachable from the root nodes in more than one way, then C keeps track of all of them. Multiple occurrences of the same pair (E, F) in C then may contribute multiple links.

Table 1 shows an example run of the discovery algorithm. The input heap H consists of a doubly-linked list of doubly-linked sublists where the backward link in the top-level list comes from the first node in the sublist. The discovery of the predicate describing the shape of the list would fail without the use of frame inference. Note that Λ_1 in the input symbolic heap could have been discovered

Routine	LOC	Space (Mb)	Time (sec)	Result
t1394_BusResetRoutine	718	322.44	663	✓
t1394Diag_CancelIrp	693	1.97	0.56	⊘
t1394Diag_CancelIrpFix	697	263.45	724	✓
t1394_GetAddressData	693	2.21	0.61	⊘
t1394_GetAddressDataFix	698	342.59	1036	✓
t1394_SetAddressData	689	2.21	0.59	⊘
t1394_SetAddressDataFix	694	311.87	956	✓
t1394Diag_PnpRemoveDevice	1885	>2000.00	>9000	T/O
t1394Diag_PnpRemoveDevice*	1801	369.87	785	✓

Table 2. Experimental results on IEEE 1394 (firewire) Windows device driver routines. "✓" indicates the proof of memory safety and memory-leak absence. "⊘" indicates that a genuine memory-safety warning was reported. The lines of code (LOC) column includes the struct declarations and the environment model code. The t1394Diag_PnpRemoveDevice* experiment used a precondition expressed in separation logic rather than non-deterministic environment code. Experiments conducted on a 2.0GHz Intel Core Duo with 2GB RAM.

by a previous run of the algorithm on a more concrete symbolic heap, possibly one containing no Λ's at all.

6 Experimental Results

Before applying our analysis to larger programs we first applied it to a set of small challenge problems reminiscent of those described in the introduction (*e.g.* "Creation of a cyclic doubly-linked list of cyclic doubly-linked lists in which the inner link-type differs from the outer list link-type", "traversal of a singly-linked list of singly-linked list which reverses each sublist twice", etc). These challenge problems were all less than 100 lines of code. We also intentionally inserted memory leaks and faults into variants of these and other programs, which were also correctly discovered.

We then applied our analysis to a number of data-structure manipulating routines from the IEEE 1394 (firewire) device driver. This was much more challenging than the small test programs. We used an implementation restricted to a simplified, singly-linked version of our abstract domain, in order to focus experimentation with the adaptive aspect of the analysis (we do not believe this restriction to be fundamental). As a result, our model of the driver's data structures was not exactly what the kernel can see. It turns out that the firewire code happens not to use reverse pointers (except in a single library call, which we were able to model differently) which means that our model is not too inaccurate for the purpose of these experiments. Also, the driver uses a small amount of address arithmetic in the way it selects fields (the "containing record idiom"), which we replaced with ordinary field selection, and our tool does not check array bounds errors, concentrating on pointer structures.

Our experimental results are reported in Table 2. We expressed the calling context and environment as non-deterministic C code that constructed five circular lists with common header, three of which had nested acyclic lists, and two of which contained back-pointers to the header; there were additionally numerous other pointers to non-recursive objects. In one case we needed to manually supply a precondition due to performance difficulties. The analysis proved safety of a number of driver routines' usage of these data structures, in a sequential execution environment (see [5] for notes on how we can lift this analysis to a concurrent setting). We also found several previously unknown bugs. As an example, one error (from `t1394_CancelIrp`, Table 2) involves a procedure that commits a memory-safety error on an empty list (the presumption that the list can never be empty turns out not to be justified). This bug has been confirmed by the Windows kernel team and placed into the database of device driver bugs to be repaired. Note that this driver has already been analyzed by SLAM and other analysis tools—These bugs were not previously found due to the limited treatment of the heap in the other tools. Indeed, SLAM *assumes* memory safety.

The routines did scanning of the data structures, as well as deletion of a single node or a whole structure. They did not themselves perform insertion, though the environment code did. Predicate discovery was used in handling nesting of lists. Just as importantly, it allowed us to infer predicates for the many pointers that led to non-recursive objects, relieving us of the need to write these predicates by hand. The gain was brought home when we wrote the precondition in the t1394Diag_PnpRemoveDevice* case. It involved looking at more than 10 struct definitions, some of which had upwards of 20 fields.

Predicate discovery proved to be quite useful in these experiments, but further work is needed to come to a better understanding of heuristics for its application. And, progress is needed on the central scalability problem (illustrated by the timeout observed for `t1394Diag_PnpRemoveDevice`) if we are to have an analysis that applies to larger programs.

7 Conclusion

We have described a shape analysis designed to fill the gap between the data structures supported in today's shape analysis tools and those used in industrial systems-level software. The key idea behind this new analysis is the use of a higher-order inductive predicate which, if given the appropriate parameter, can summarize a variety of composite linear data structures. The analysis is then defined over symbolic heaps which use the higher-order predicate when instantiated with elements drawn from a cache of non-recursive predicates. Our abstraction procedure incorporates a method of synthesizing new non-recursive predicates from an examination of the current symbolic heap. These new predicates are added into the cache of non-recursive predicates, thus triggering new rewrites in the analysis' abstraction procedure. These new predicates are expressed as the combination of old predicates, including instantiations of the higher-order predicates, thus allowing us to express complex composite structures.

We began this work with the idea sometimes heard, that systems code often "just" uses linked lists, and we sought to test our techniques on such code. We obtained encouraging, if partial, experimental results on routines from a firewire device driver. However, we also found that lists can be used in combination in subtle ways, and we even encountered an instance of sharing (described in Section 2) which, as far as we know, is beyond current automatic shape analyses. In general, real-world systems programs contain much more complex data structures than those usually found in papers on shape analysis, and handling the full range of these structures efficiently and precisely presents a significant challenge.

Acknowledgments. We are grateful to the CAV reviewers for detailed comments which helped us to improve the presentation. The London authors were supported by EPSRC.

References

[1] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.

[2] B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines and higher-order separation logic. In *ESOP*, 2005.

[3] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract tree regular model checking of complex dynamic data structures. *SAS 2006.*

[4] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.

[5] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *To appear in PLDI*, 2007.

[6] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*. 2005.

[7] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, 2005.

[8] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transfomers. In *CAV*. 2006.

[9] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. SAS 2000.

[10] A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement via inductive learning. CAV 2005.

[11] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI*. 2005.

[12] A. Podelski and T. Wies. Boolean heaps. In *SAS*, 2005.

[13] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[14] N. Rinetzky, G. Ramalingam, M. Sagiv, and E. Yahav. Componentized heap abstraction. TR-164/06, School of Computer Science, Tel Aviv Univ., Dec 2006.

[15] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, 1998.

[16] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[17] M. Češka, P. Erlebach, and T. Vojnar. Generalised multi-pattern-based verification of programs with linear linked structures. *Formal Aspects Comput.*, 2007.