Shape analysis for composite data structures

Josh Berdine Cristiano Calcagno Byron Cook
Dino Distefano Peter W. O'Hearn Thomas Wies
Hongseok Yang

January 29, 2007

Technical Report
MSR-TR-2007-13

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

Shape analysis for composite data structures

Josh Berdine†, Cristiano Calcagno$^\sharp$, Byron Cook†, Dino Distefano‡, Peter W. O'Hearn‡, Thomas Wies*, and Hongseok Yang‡

$\dagger$ = Microsoft Research, $\sharp$ = Imperial College
$\ddagger$ = Queen Mary $\star$ = 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 doubly-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 parametrized 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 implemented the analysis and performed experiments with it on examples drawn from device drivers. During our experiments the analysis proved the memory safety of several routines belonging to an IEEE 1394 (firewire) driver, and also found several previously unknown memory safety bugs and memory leaks. To our knowledge this represents the first known successful application of shape analysis to non-trivial systems-level code.

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.*, [1, 7, 11, 16, 17, 9]). 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), today's *fully-automatic* shape analyses will all 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 parametrized 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 memory safety 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 some limited forms of nesting. For example, the tool described in [10] 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. [10] can handle varieties of trees, where the specific abstraction given in this paper cannot. Conversely, our analysis supports doubly-linked list segments, lists of cyclic lists with back-pointers, and lists of cyclic doubly-linked lists (as in the 1394 driver), where [10] cannot due to the fact that these data structures require inductive definitions with more than two parameters and the abstract domain of [10] cannot express such definitions.

Work on analysis of complex structures using regular model checking includes an example on a list of lists [4]. We do not believe that the abstraction there can deal with back-pointers as we have had to in the 1394 driver, but it might be possible to modify it to do so.

The parametric shape analysis framework of [12, 21] 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 [14], 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) [13]. There has been preliminary work on instrumentation predicates for composite structures [19], but as far as we are aware it has not been implemented or otherwise evaluated experimentally.

As previously mentioned, deep shape analyses have not been often applied to the source code of real-world programs that mutate data structures,[1] and this is recognized as a limitation in the field. This paper represents one attempt to do so, and the design of our abstract domain was influenced by the kinds of data structure found in systems programs. To the best of our knowledge, our experiments with the IEEE 1394 driver represent the first known successful application of a shape analysis to non-trivial systems-level code.

2 Synthesized predicates and general inductions schemes

The analysis described in this paper fits in to the common structure of shape analyses based on abstract interpretation (*e.g.* [7, 5, 15, 4, 20]) 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) in order to ensure the existence of a fixed point. In this work we use a representation of abstract states based on separation logic [8, 18] formulæ, building on the methods of [2, 5].

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.[2] In effect, we propose using a restricted subset of a higher-order version of separation logic [3]. 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 parametrized 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`, when it is loaded on

[1] One noteworthy exception is [7], which obtains impressive results by intentionally restricting expressiveness to certain flat list structures, in order to gain efficiency.

[2] In this paper we have concentrated on varieties of linked lists motivated by problems seen during the application of the analysis to device driver code, but we expect that the basic ideas can also be applied with varieties of tree and other structures, and hope to report on that in future work. In order to support trees we would need to add an additional higher-order recursive predicate describing that structure.

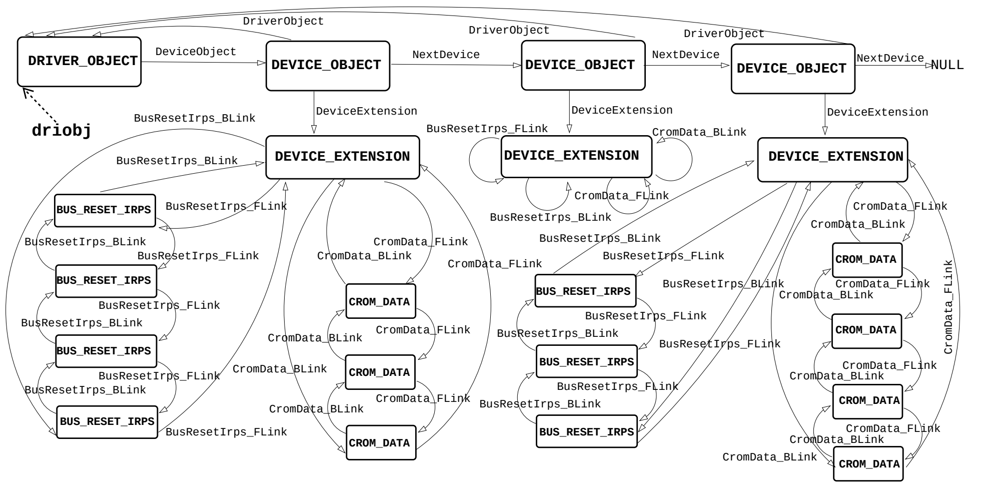


Fig. 1. Device-driver like heap configuration.

$$
\begin{aligned}
\Lambda_{Bus} &\triangleq \lambda[x',y',z',w'].\, x' = w' \wedge (x' \mapsto \texttt{BUS\_RESET\_IRP}(\texttt{BusResetIrps\_Blink}\colon y', \texttt{BusResetIrps\_Flink}\colon z'))\\
\Lambda_{CROM} &\triangleq \lambda[x',y',z',w'].\, x' = w' \wedge (x' \mapsto \texttt{CROM\_DATA}(\texttt{CromData\_Blink}\colon y', \texttt{CromData\_Flink}\colon z'))\\
\Lambda_{Device} &\triangleq \lambda[x',y',z',w'].\, \exists e', b', b'', c', c''.\, x' = w' \wedge\\
&\qquad x' \mapsto \texttt{DEVICE\_OBJECT}(\texttt{NextDevice}\colon z', \texttt{DeviceExtension}\colon e', \texttt{DriverObject}\colon \texttt{driobj}) \ast\\
&\qquad e' \mapsto \texttt{DEVICE\_EXTENSION}(\texttt{BusResetIrps\_Blink}\colon b', \texttt{BusResetIrps\_Flink}\colon b'',\\
&\qquad\qquad\qquad \texttt{CromData\_Blink}\colon c', \texttt{CromData\_Flink}\colon c'') \ast\\
&\qquad \mathsf{ls}\ \Lambda_{Bus}\ (b'', e', e', b') \ \ast\ \mathsf{ls}\ \Lambda_{CROM}\ (c'', e', e', c')
\end{aligned}
$$

Fig. 2. Parametrized predicates inferred from the heap in Fig. 1.

the 1394 bus driver and three 1394 devices are connected to the machine. In this figure the pointer `driobj` is a pointer to a *driver object* defined by a Windows kernel C structure called `DRIVER_OBJECT`. Amongst their many other fields, driver objects hold the sentinel node to a list of *device objects*, defined by another Windows kernel structure called `DEVICE_OBJECT`. Each device object has a back-pointer to the original driver object. Moreover, 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 two of the five cyclic doubly-linked lists in `DEVICE_EXTENSION`: one cyclic list of `BUS_RESET_IRP` structures, and one of `CROM_DATA` structures.

When the abstraction step from our analysis is applied to the heap in Fig. 1 several predicates are discovered. Consider the doubly-linked lists at the bottom of the figure. These are circular doubly-linked lists, and can be described using certain instances of ls. The linked lists of `BUS_RESET_IRP` and `CROM_DATA` entries can be described using predicates of the form "$\mathsf{ls}\ \Lambda_{Bus}\ (x, y, z, w)$" and "$\mathsf{ls}\ \Lambda_{CROM}\ (x, y, z, w)$", where the predicates Λ_{Bus} and Λ_{CROM} are shown in Fig. 2. Λ_{Bus}, for example, is a predicate that takes in four parameters (in this work all parameterized predicates take 4 parameters) and then, using $\mapsto$ from separation logic, declares that a block of memory with type `BUS_RESET_IRP` is allocated at the location pointed to by x'. Λ_{Bus} also specifies the values of the fields in the `BUS_RESET_IRP` to be equal to y' and z'.

Next, the singly-linked list of `DEVICE_OBJECT`s can be defined as a further instance of ls, obtained from a predicate Λ_{Device}, from Fig. 2, that is built from the predicates for the circular linked lists. Λ_{Device} uses three instances of separation logic's separating conjunction connective $*$ to indicate the existence of four distinct segments of the heap pointed to by x', e', b'', and c''. The $*$-conjunction specifies that the only aliasing occurring between these four independent heaps be via the variables in the formula. Note that Λ_{Device} is not itself recursive but is defined in terms of instances of ls where the parameters are previously synthesized predicates. Note also that, using Λ_{Bus} and Λ_{CROM} as parameters to ls, the specific number of elements in each list is forgotten.

After abstraction, the original heap can be covered by the symbolic heap $H \triangleq \mathtt{driobj} \mapsto \mathtt{DRIVER\_OBJECT}(\mathtt{DeviceObject}\colon x'_1) * \mathsf{ls}\ \Lambda_{device}\ (x'_1, \_, \mathsf{nil}, \_)$. This formula is more abstract than the beginning heap in that the lengths of the singly-linked list of `DEVICE_OBJECT`s and of the doubly-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.

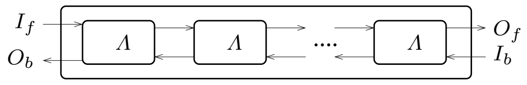
In this paper, we consider the storage model given by $\mathit{Stack} \triangleq (\mathit{Var} \cup \mathit{Var}') \to \mathit{Val}$, $\mathit{Heap} \triangleq \mathit{Loc} \rightharpoonup_{\mathsf{fin}} (\mathit{Fld} \rightharpoonup \mathit{Val})$, and $\mathit{States} \triangleq \mathit{Stack} \times \mathit{Heap}$. 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:

expressions	$E ::= x \mid x' \mid \mathsf{nil}$
pure formulæ	$\Pi ::= \mathsf{true} \mid E{=}E \mid E \neq E \mid \Pi \wedge \Pi$
spatial formulæ	$\Sigma ::= \mathsf{emp} \mid \Sigma * \Sigma \mid E \mapsto T(\vec{f}\colon \vec{E}) \mid \mathsf{ls}\ \Lambda\ (E, E, E, E) \mid \mathsf{true}$
symbolic heaps	$H ::= \Pi \wedge \Sigma$
par. symb. heaps	$\Lambda ::= \lambda[x', y', z', w'].\ \exists \vec{x'}.\ H$

Here x and x' are drawn from Var and Var', respectively, and f from Fld. We will use a macro $\Lambda[D, E, F, G, \vec{E'}]$ to express the symbolic heap derived by instantiating Λ's parameters (x', y', z', w') with (D, E, F, G) and existentially quantified variables $\vec{v'}$ with $\vec{E'}$.

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 $\lambda[x', y', z', x'].\ (x' \mapsto \texttt{Node}(\texttt{Next:}\ z'))$ 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 $\lambda[x', y', z', x'].\ x' \mapsto \texttt{DNode}(\texttt{Blink:}\ y', \texttt{Flink:}\ z')$.
– If Λ_h is $\lambda[x', y', z', x'].\ x' \mapsto \texttt{HNode}(\texttt{Next:}\ z', \texttt{Head:}\ k)$, 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

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

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 [18]. 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 dispose). 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 Shape analysis with spatial predicate discovery

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 ove-approximates the concrete semantics of the program.

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 [2, 5] are easily modified for our setting. In contrast, the canonicalization operator for our abstract domain significantly departs from [5] and forms the crux of our analysis. We describe it in the remainder of this section.

4.1 Canonicalization

Canonicalization performs a form of over-approximation by soundly removing some information from a given symbolic heap. It is defined by the rewriting

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

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

$$\frac{\mathsf{FV}(I_f, I_b) = \emptyset \qquad \mathsf{FV}'(I_f, I_b) \cap \mathsf{FV}'(\mathsf{spatial}(H)) = \emptyset}{H * \mathsf{ls}\ \Lambda\ (I_f, O_b, O_f, I_b) \;\leadsto\; H * \mathsf{true}}\ (\mathsf{Junk\ 1}) \qquad \frac{\mathsf{FV}(E) = \emptyset \qquad \mathsf{FV}'(E) \cap \mathsf{FV}'(\mathsf{spatial}(H)) = \emptyset}{H * (E \mapsto T(\vec{f}\colon \vec{E})) \;\leadsto\; H * \mathsf{true}}\ (\mathsf{Junk\ 2})$$

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

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

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

Fig. 3. Rules for Canonicalization.

rules ($\leadsto$) 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 Predicate Intro rule from Fig. 3 represents a novel aspect of our canonicalization procedure. It scrutinizes the linking structure of a symbolic heap to infer a set of predicates Λ that describe subheaps that form a list. Then two appropriately connected Λ nodes are removed from the symbolic heap and replaced with a $\mathsf{ls}\ \Lambda$ formula. The second step is accomplished by frame inference ($\vdash$, described below), and the first step is accomplished by predicate discovery (Discover, described in Section 4.2).

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. Note that, as in predicate discovery and introduction, frame inference is used in these rules to perform semantic decomposition of formulæ. The variable side-conditions on the rules are necessary for precision but not soundness; they prevent the rules from firing too often.

Frame inference. In practice the canonicalization procedure often fails when syntactic rather than semantic methods are used to infer predicates. For this reason we make use of a *frame inferring theorem prover* [2]— a prover for entailments $H \vdash H_1 * H_0$ where only H and H_0 are given and H_1 is inferred. This enables canonicalization to identify instances of predicates Λ that can only be revealed by taking the semantics of symbolic heaps into account. Thus, taking the semantics of formulæ into consideration when subtracting predicate instances from symbolic heaps makes canonicalization stronger than it would be with pure syntactic subtraction. While the aim of a frame inferring theorem prover is to find a decomposition of H into H_1 and H_0 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 $H \vdash H_1 * H_0$, we not only require the entailment to hold, but also require that there exists a disjoint extension of the heap satisfying H_0, and the extended heap satisfies H.

4.2 Predicate discovery

We next describe our heuristic implementation of $\mathsf{Discover}$, which is designed to generate 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, $E \mapsto T(\vec{f}\colon \vec{E})$ 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_0 and E_1. Once root nodes are found, $\mathsf{Discover}(H)$ traverses the graph from E_0 as well as from E_1 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.

Fig. 4 shows the pseudo code for the discovery algorithm. The set I denotes the disjoint subgraph isomorphism between the traverses of the two subgraphs reachable from the chosen root nodes. The set C marks the already traversed part and is used for cycle detection. The main purpose of sets I and C, however, is to enable construction of the parameter list for the discovered predicate: these sets tell us what the forward and backward links between the two traverses are. For this reason C is not simply a set, but a multiset, i.e. if a pair (E_0, E_1) 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_0, E_1) in C may then contribute multiple links to the parameter list.

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:

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

```
$\mathsf{discover}(H\colon \text{symbolic heap})\colon \text{predicate} =$
    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 $(E_0, E_1) \in \{(E_0, E_1) \mid \Sigma = E_0 \mapsto f\colon E_1 * E_1 \mapsto f\colon E * \Sigma'\}$
    let $W = \{(E_0, E_1)\}$: multiset of expression pairs
    do
      choose $(E_0, E_1) \in W$
      if $E_0 \neq E_1$ then
        if $(E_0, E_1) \notin C \wedge E_0 \notin \mathit{rng}(I) \wedge E_1 \notin \mathit{dom}(I)$ 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 $\vec{I_f}, \vec{O_f} = [(E, F) \mid (E, F) \in C \wedge F \in \mathit{rng}(I)]$
    let $\vec{I_b}, \vec{O_b} = [(E, F) \mid (F, E) \in C \wedge E \in \mathit{dom}(I)]$
    let $\vec{x'} = \mathsf{FV}'(\Sigma_\Lambda) - \mathsf{FV}'(\vec{I_f}, \vec{O_f}, \vec{I_b}, \vec{O_b})$
    return $(\lambda(\vec{I_f}, \vec{O_b}, \vec{O_f}, \vec{I_b}).\ \exists \vec{x'}.\ \Sigma_\Lambda)$
```

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

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 $P(E, \vec{F})$ then contributes to the spatial part of the discovered predicate.

Note that we must perform a bit of bookkeeping on the inputs and outputs to the discovery procedure. Before applying discovery each expression in H should be replaced by fresh primed variables. Later, after a predicate has been discovered, we must then substitute each of these free primed variables in the predicate with the expressions from the original H.

Example. 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. Note that the discovery of the predicate describing the shape of the list would fail without the use of frame inference.

4.3 Soundness

All canonicalization rules correspond to sound implications in separation logic, *i.e.* we have that $H \vDash H'$ whenever $H \leadsto H'$, and this is the crucial fact underlying soundness of canonicalization. The relation $\leadsto$ is nondeterministic, but our implementation uses a specific reduction strategy. There is a progress measure for the rewrite rules, so $\leadsto$ is strongly normalizing. So, mathematically, we

Input symbolic heap
$H = x'_0 \mapsto T(f\colon x'_1, g\colon y'_0) \ast x'_1 \mapsto T(f\colon x'_2, g\colon y'_1) \ast x'_2 \mapsto T(f\colon x'_3, g\colon y'_2) \ast$ $\mathsf{ls}\ \Lambda_1\ (y'_0, \mathsf{nil}, z'_1, \mathsf{nil}) \ast y'_1 \mapsto S(f\colon \mathsf{nil}, b\colon x'_0) \ast \mathsf{ls}\ \Lambda_1\ (y'_2, x'_1, z'_2, \mathsf{nil})$ 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	$\{(x'_0, x'_1)\}$	$\{(x'_1, x'_2), (x'_2, x'_3), (y'_1, y'_2)\}$	$\{(x'_1, x'_2), (y'_1, y'_2)\}$	$x'_1 \mapsto T(f\colon x'_2, g\colon y'_1) \ast \mathsf{ls}\ \Lambda_1\ (y'_1, x'_0, z'_1, \mathsf{nil})$
4	$\emptyset$	$\{(x'_1, x'_2), (x'_2, x'_3), (y'_1, y'_2), (x'_0, x'_1)\}$	$\{(x'_1, x'_2), (y'_1, y'_2)\}$	$x_1 \mapsto T(f\colon x'_2, g\colon y'_1) \ast \mathsf{ls}\ \Lambda_1\ (y'_1, x'_0, z'_1, \mathsf{nil})$

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

Table 1. Example run of discovery algorithm

assume a function $abs\colon \mathcal{SH} \longrightarrow \mathcal{SH}$, where $abs(H)$ is always canonical. We obtain $canonicalize\colon \mathcal{D}^{\#} \longrightarrow \mathcal{D}^{\#}$, by setting $canonicalize(\top) = \top$, and otherwise $canonicalize(d) = \{abs(H) \mid H \in d\}$.

Lemma (Canonicalization Soundness). $d \vDash canonicalize(d)$.

5 Experimental results

In order to evaluate our analysis we have developed a preliminary implementation and applied it to a series of programs. This section describes the results of these experiments.

Small challenge problems. Before applying our analysis to larger programs we first successfully applied our tool to a set of 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 and dereferences a dangling pointer after the second reversal of a sublist, but only if sublists are non-empty", 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.

IEEE 1394 (firewire). We applied our analysis to a number of data-structure manipulating routines from the IEEE 1394 (firewire) device driver discussed in Section 2. In our experiments, we expressed the routine's calling context and environment as non-deterministic C code that constructed representative data structures (though in one case we needed to manually supply a precondition expressed in separation logic due to performance difficulties).

Our experimental results are reported in Table 2. Our analysis proved a number of driver routines to be memory-safe in a sequential execution environment (see [6] for notes on how we can lift this analysis to a concurrent setting). Our analysis also found several previously unknown errors in the routines. As an example, one error (from `t1394_CancelIrp`, Table 2) involves a procedure that takes a list pointed to by a structure contained in another list and commits a memory-safety error when the nested list is empty (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.

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	398	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	785.43	✓

Table 2. Experimental results with the analysis on a IEEE 1394 (firewire) Windows device driver code. "✓" 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.

6 Conclusion

We have described a novel 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 immediately 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. The adaptive abilities of our new shape analysis have enabled us, for the first time, to successfully apply a fully-automatic shape analysis to the source code of a device driver.

References

[1] I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *VMCAI'05*.

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

[3] B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines and higher-order separation logic. In *ESOP*, 2004.

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

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

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

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

[8] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.

[9] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL*, 2006.

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

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

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

[13] A. Loginov. Personal communication.

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

[15] S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in Separation Logic for imperative list-processing programs. 3rd SPACE Workshop, 2006.

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

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

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

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

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

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