Shape Analysis of Low-level C with Overlapping Structures

Jörg Kreiker, Helmut Seidl, and Vesal Vojdani*

Fakultät für Informatik, Technische Universität München
Boltzmannstraße 3, D-85748 Garching b. München, Germany
{kreiker,seidl,vojdanig}@in.tum.de

Abstract. Device drivers often keep data in multiple data structures simultaneously while embedding list or tree related records into the records containing the actual data; this results in *overlapping* structures. Shape analyses have traditionally relied on a graph-based representation of memory where a node corresponds to a whole record and edges to pointers. As this is ill-suited for encoding overlapping structures, we propose and formally relate two refined memory models. We demonstrate the appropriateness of these models by implementing shape analyses based on them within the TVLA framework. The implementation is exemplified using code extracted from cache managing kernel modules.

1 Introduction

Shape analysis of heap-manipulating programs is a very active field of research; however, the focus of most work has been devoted to Java-like data-structures, where pointers are not as heavily manipulated and computed with as in low-level C. While shape analyses addressing pointer arithmetic in a broad sense have recently been designed, e.g., [3, 8–10, 16, 18, 19], we address a related and particularly difficult problem: *overlapping structures*. The term was coined in [3], where the shape analysis of such structures was stated as an open problem.

Overlap is often found in device drivers where data is kept in several data-structures at the same time by means of embedding list or tree related records into the records containing the actual data. An example of such code is shown in Figure 1, where a `node` record (we shall consequently use the more general term record to denote C structs) contains data as well as two list-related components. The first, `hlist_node`, is the record type which embeds the forward and backward pointers of an hlist (see below) into a node; the second, `list_head`, is a record type which serves both as the list head and as the record that embeds standard cyclic doubly-linked lists into nodes.

Hlists (or pprev lists) are in themselves quite tricky data-structures. In order to save memory while maintaining efficient implementation of insertion and deletion, Linux developers use these doubly linked lists with a pointer to the `next` component of the previous element rather than to the element itself. (This is

* On leave from the University of Tartu; partially supported by EstSF grant 6713.

```
struct hlist_head { struct hlist_node *first; };
struct hlist_node { struct hlist_node *next, **pprev; };
struct  list_head { struct  list_head *next, *prev; };

struct node { int data; struct hlist_node list;
                        struct  list_head queue; };

struct hlist_head ht[512];  struct mutex hlock;
struct  list_head cq;

void cleanup_task(void *arg) {
    struct hlist_head garbage; struct node *pos;
    lock(&hlock);
    list_for_each(pos, &cq) {
        hlist_del(&pos→list);
        list_del(&pos→queue);
        hlist_add(&pos→list, &garbage); }
    unlock(&hlock);
    hlist_for_each(pos, &garbage, list) {
        access(pos→data);
        hlist_del(&pos→list); } }
```

Fig. 1: Overlapping data-structures from the Linux kernel.

visualized in a memory snapshot in Figure 2, where the edges from pprev boxes end at the smallest boxes rather than at the medium-sized ones as is the case for edges originating in prev boxes.) Hlists are used in hash-tables where having only a single pointer in the list head can be a significant gain.

The code given in Figure 1 shall serve as a basis for our case study. We use a syntax close to the original code, but abbreviate function names and eliminate some of the parameters to the list-traversal macros. These macros expand into for-loops and use pointer-arithmetic to move from a record embedded within a node to the containing record. The example is based on code for maintaining a cache where the least recently used items are tagged and added to the cleanup queue. This queue is processed asynchronously by a cleanup task whose code is given in the figure. As this task may be executing concurrently with code that accesses the cache, elements in the queue are moved to the thread-local list `garbage` for statistical processing before being deallocated. This minimizes the time that the cleanup task must keep the lock on the cache.

When an object is removed from *all* thread-shared data-structures, the subsequent post-processing of the *privatized* object no longer requires protection through the acquisition of locks. However, if an element resides in two lists simultaneously, traversing these distinct lists may cause a race when accessing the data of the shared element. To prove absence of races in the example, we must infer that an element is in the queue but no longer in the list, although both queue- and list-related records are embedded into the same `node`.

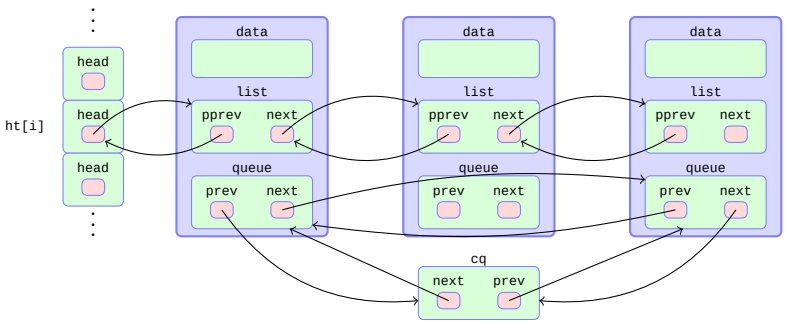

Fig. 2: Overlapping Structures.

Shape analyses often rely on graph-based representations of memory where a node corresponds to a whole record and edges to pointers. For these, it is not immediate how to encode pointers between components. Therefore, we propose two refined memory models which exhibit the low-level details required to reason about overlapping structures. Both memory models are formulated in terms of the TVLA framework [27]. Using the TVLA framework is not necessary but allows for a quick prototype implementation. Since our refinement is conservative, we can also benefit from knowledge and developments in the TVLA realm.

The model of Section 3 uses a one-node-per-component approach, i.e., each box of Figure 2 becomes a single node in the shape graph, and the hierarchy induced by the box nesting is translated into a tree structure. We design a program analysis based on this model and demonstrate its potential on a list element deletion procedure that uses unorthodox pointer manipulations.

Alternatively in Section 4, we propose a coarser and possibly more efficient model that employs a one-node-per-outermost-record paradigm. This approach annotates edge dereferences with access paths into the finer structure of the records, making sources and targets of dereferencing explicit. We exemplify shape analyses based on this coarser semantics by verifying deletion from an hlist. In Section 5, we characterize the relationship between the two models.

In Section 6, we revisit our motivating example of Figure 1. The analysis is conducted w.r.t. the coarse-grained semantics and enables us to verify race detection properties for it. Sections 7 and 8 present related work and conclude.

2 Preliminaries

We begin by introducing the syntax of the C subset under consideration. Our aim is to cover the part of the C language crucial to most low-level programs like kernel code and drivers. We support arbitrarily nested named records and pointers to named types only; integers are not considered. This implies that we have both records and pointers as values. We use the domain **Id** of *variables*

ranged over by x, y, and z, and the domain **Sel** of component *selectors* ranged over by s. Type names are ranged over by t. We consider the following languages of types τ and pointer expressions e:

$$\tau ::= \mathbf{struct}\ t\ \{\tau_1\ s_1, \ldots, \tau_k\ s_k\} \mid \tau\ *$$
$$e ::= \mathbf{0} \mid x \mid *e \mid \&e \mid e \to s \mid \mathbf{malloc}(t) \mid \mathbf{up}(x, t, s)$$

We omit arbitrary pointer arithmetic, unions, and type-casts; rather, we restrict pointer manipulation to component selection and the expression $\mathbf{up}(x, t, s)$ used as a primitive to model the `container_of` macro which computes the address of a record of type t from a pointer x to its s component. Since recent versions of the Linux kernel rely on *built-in* support by the compiler to implement this macro, a primitive treatment of the container-of idiom is sensible.

In the presence of overlap the $\mathbf{up}()$-operator is crucial to *change views*. For instance, in the example of Figure 1 one could traverse the cleanup queue, use the operator $\mathbf{up}()$ to jump to the data component of a node, and then continue traversing the `list` components.

As for statements we only consider assignments between pointer expressions. We assume that programs are compiled into a control-flow graph where assignments are attached to edges and where pointer comparisons may serve as guards. Two different semantics will be provided in Sections 3 and 4 in the framework of the *Three-valued Logic Based Shape Analysis (TVLA)* [27]. We therefore briefly recapitulate the basics of the TVLA approach in order to have the necessary notation at hand.

TVLA builds on the notion of logical structures over a certain signature $\mathcal{P}$. A logical structure $S = (U, \iota)$ is a pair of a set of individuals U ranged over by u and an interpretation, ι. Each predicate symbol $p/k \in \mathcal{P}$ of arity k is mapped by ι to a boolean-valued function $\iota(p/k) : U^k \to \mathbb{B}$. The set of all structures over a $\mathcal{P}$ is written $\mathcal{S}[\mathcal{P}]$. We evaluate formulas of first-order logic with transitive closure, FO(TC), on logical structures. Formulas are defined by:

$$\varphi = 0 \mid p(v_1, \ldots, v_k) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists v : \varphi \mid TC(v_1, v_2 : \varphi)(v_1', v_2')$$

where $v \in \mathsf{Var}$ is a logical variable. The transitive closure operator, $TC(v_1, v_2 : \varphi)(v_1', v_2')$ defines a binary relation by φ using free variables v_1 and v_2. The transitive closure of this relation is then evaluated on v_1' and v_2'. The evaluation of a formula φ in structure S and assignment Z (of free variables to individuals) is written $[\![\varphi]\!]^S(Z)$.

Logical structures are used to encode heap graphs. Traditionally, an individual corresponds to a record and a binary predicate s holds of individuals u and u', if there is a pointer-valued component s of the record modeled by u which points to (the head of) the record modeled by u'. The predicates used to encode a heap are essentially the binary selectors, **Sel**, and the unary program variables, **Id**, that hold of records pointed to by the corresponding variables. These predicates are called *core predicates*, the set of which is denoted by $\mathcal{C}$.

The semantics of an assignment st is a mapping $[\![st]\!] : \mathcal{S}[\mathcal{C}] \to \mathcal{S}[\mathcal{C}]$ This is given in terms of *predicate update formulas* that update the value of predicates

affected by the statement. Given for each k-ary predicate $p \in \mathcal{C}$ an update formula φ_p^{st} with free variables $v_1, \ldots, v_k$, the semantics of st is defined as $[\![st]\!](S) = (U', \lambda p.\lambda u_1, \ldots, u_k.[\![\varphi_p^{st}]\!]^S(Z))$ where $Z = [v_1 \mapsto u_1, \ldots, v_k \mapsto u_k]$ and the universe U' is either the same as before or (in the case of memory allocation) extended with fresh individuals.

Abstract states in the TVLA framework are *three-valued logical structures* based on Kleene's three-valued logic. Abstract states are obtained by *canonical abstraction*, an abstraction that summarizes individuals that are indistinguishable under a set of abstraction predicates to *summary nodes*. Due to summarization, knowledge about certain predicates may become indefinite, in which case the logical value $1/2$ is introduced. To avoid serious loss of precision, *instrumentation predicates* are employed; these are additional predicates defined through formulas of FO(TC) using the core predicates $\mathcal{C}$. Instrumentation predicates allow to better distinguish abstract nodes by annotating logical structures with additional information such as reachability, sharing, or cyclicity. Update formulas for instrumentation predicates can be automatically inferred using differencing [25].

As abstract states are still logical structures, the concrete semantics in terms of predicate update formulas is easily lifted to three-valued logical structures, too. As a consequence, it is sufficient to specify predicate update formulas and a set of instrumentation predicates to define a program analysis in the TVLA framework. Soundness then is immediate.

3 Fine-grained Semantics and Analysis

We now present our first refinement of the one-node-per-record paradigm by adopting a *one-node-per-component* representation. More precisely, a record of type **struct** t $\{\tau_1 * s_1, \ldots, \tau_k * s_k\}$ is represented by $k + 1$ nodes, one being the head and one for each pointer component. If the component types are records again, additional nodes for the subcomponents are introduced, until finally pointer types are reached. This corresponds to transforming the hierarchy of boxes in Figure 2 into a tree.

This memory model is both more explicit and more abstract than that of real C. While in C the address of a record and the address of its first component coincide, they are considered as different here. On the other hand, we do not model the order of components or padding between each two of them. In our model, the operation **up**() amounts to moving from a component node to the head of its enclosing record.

Since we rely on the TVLA framework, we aim at encoding a state as a logical structure. Here we use the signature

$$\mathcal{C} = \{x/1 \mid x \in \mathbf{Id}\} \cup \{s/2 \mid s \in \mathbf{Sel}\} \cup \{*/2\}$$

In order to reason about expressions of the form $\&x$, the corresponding predicate x holds of an individual representing the stack location where x's value is stored. In standard TVLA the predicate x holds of the element *pointed to* by x.

st	$\varphi_*^{st}(v_1, v_2) =$
$x = \mathbf{0}$	$*(v_1, v_2) \wedge \neg x(v_1)$
$*x = \mathbf{0}$	$*(v_1, v_2) \wedge \neg\exists\, v' :\ x(v') \wedge *(v', v_1)$
$x \to s = \mathbf{0}$	$*(v_1, v_2) \wedge \neg\exists\, v', v'' :\ x(v') \wedge *(v', v'') \wedge s(v'', v_1)$
$x = y$	$*(v_1, v_2) \vee x(v_1) \wedge \exists\, v' :\ y(v') \wedge *(v', v_2)$
$x = *y$	$*(v_1, v_2) \vee x(v_1) \wedge \exists\, v', v'' :\ y(v') \wedge *(v', v'') \wedge *(v'', v_2)$
$x = \&y$	$*(v_1, v_2) \vee x(v_1) \wedge y(v_2)$
$x = \&y \to s$	$*(v_1, v_2) \vee x(v_1) \wedge \exists\, v', v'' :\ y(v') \wedge *(v', v'') \wedge s(v'', v_2)$
$x = y \to s$	$*(v_1, v_2) \vee x(v_1) \wedge \exists\, v', v'', v''' : y(v') \wedge *(v', v'') \wedge s(v'', v''') \wedge *(v''', v_2)$
$x = \mathbf{up}(y, t, s)$	$*(v_1, v_2) \vee x(v_1) \wedge \exists\, v', v'' :\ y(v') \wedge *(v', v'') \wedge s(v_2, v'')$
$*x = y$	$*(v_1, v_2) \vee \exists\, v', v'' :\ x(v') \wedge *(v', v_1) \wedge y(v'') \wedge *(v'', v_2)$
$x \to s = y$	$*(v_1, v_2) \vee \exists\, v', v'', v''' : x(v') \wedge *(v', v'') \wedge s(v'', v_1) \wedge y(v''') \wedge *(v''', v_2)$

Table 1: Predicate update formulas for nullification and assignments. The latter assume that $*x$, x, and $x \to s$ have been nullified.

The key predicate in our formulation is the binary $*$ for dereferencing. In particular, it holds between the location of a pointer variable and the value the variable points to. An interesting feature of our model is that the only predicate that is affected by assignments is the $*$ predicate: once allocated, the nodes representing a record and its components do not change and neither does the location of a variable.

We now formalize the semantics of our programming language. Each basic statement gives rise to an *update formula*. The update formulas, except for allocation, are given in Table 1. As is often done, we consider only one pointer operation per assignment, and we assume that pointers which are assigned to are always explicitly nullified before-hand, so that updates for assignments only introduce a single new points-to relationship. For example, in the case of $x = \mathbf{up}(y, t, s)$, we have that $*(u_1, u_2)$ holds after the assignment if it either held before, or if u_1 is the individual hosting x and u_2 has an s-component which is pointed to by the individual hosting y.

As for memory allocation, we expand the universe by a set of new individuals depending on the type of the record to be allocated. Recall that we require one individual per (sub)component of each non-pointer type. To this end, we introduce the notion of an *access path*. Such paths are not to be confused with access paths found in storeless semantics [15, 20]. Here, they merely reflect the static structure of a (possibly nested) record. Intuitively, there is a path for each component of a record. Formally, we define the set Π of access paths to be a union over all record types t occurring in the program, $\Pi = \bigcup_t \Pi(t)$, where

$$\Pi(\tau\ *) = \{\varepsilon\}$$
$$\Pi(\mathbf{struct}\ t\ \{\tau_1\ s_1, \ldots, \tau_k\ s_k\}) = \textstyle\bigcup_{i=1}^{k} \{s_i\}.\Pi(\tau_i) \cup \{s_i\}$$

As t ranges over record types, $\varepsilon \notin \Pi$; we write $\Pi_\varepsilon = \Pi \cup \{\varepsilon\}$.

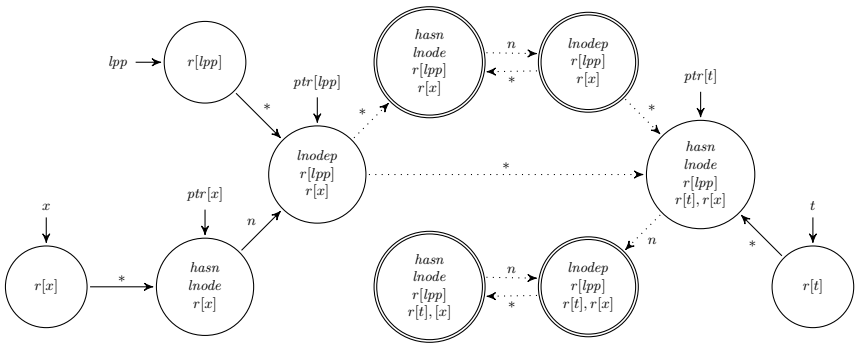

Fig. 3: Shape graph during list traversal with indirect pointer. Double lines indicate summary nodes, solid arrows indicate definite edges, and dotted ones 1/2-edges. Predicates within a node do hold for this node.

Using the access paths from Π, we define the semantics of malloc as $[\![x = \mathbf{malloc}(t)]\!](U, \iota) = (U', \iota')$ where $U' = U \uplus \{u_\pi \mid \pi \in \Pi_\varepsilon(t)\}$ and

$$\iota'(s)(u_1, u_2) = \begin{cases} \iota(s)(u_1, u_2) & \text{if } u_1, u_2 \in U \\ 1 & \text{if } u_1 = u_\pi \wedge u_2 = u_{\pi.s} \\ 0 & \text{otherwise} \end{cases}$$

$$\iota'(*)(u_1, u_2) = \begin{cases} \iota(*)(u_1, u_2) & \text{if } u_1, u_2 \in U \\ 1 & \text{if } \iota(x)(u_1) \wedge u_2 = u_\varepsilon \\ 0 & \text{otherwise} \end{cases}$$

Analysis. As first example which goes beyond the one-node-per-record memory model, we consider a program which iterates over a singly-linked list pointed to by `x` using a pointer, `lpp`, to the `next` *component* of list elements, rather than to the elements themselves. The iteration is driven by the loop

```
for (lpp = &x; *lpp != NULL; lpp = &(*lpp)→next)
```

In the beginning `lpp` points to the *address* of `x`. It is advanced by dereferencing and taking the address of the `next` component of the next element. Once an element to be deleted is found, the assignment `*lpp = (*lpp)→next` removes it from the list. This routine is quite elegant in that it needs only one iterator and no check whether the iterator points to the first element or not. Also it uses pointers to components of records.

We implemented the creation of a fine-grained singly-linked list, the iteration over it, and the deletion of an element from it as outlined above in TVLA.[1]

[1] All our TVLA analysis specification files are provided online at `http://www7.in.tum.de/~joba/overlap.tgz`. These files also contain the precise definitions of instrumentation predicates and integrity constraints.

The encoding of the fine-grained model into TVLA amounted in representing the $*$ predicate together with its update formulas for the basic statements. Through this encoding, we could re-use instrumentation predicates like sharing and reachability (`r[z]`: reachability from program variable `z`) to make the analysis go through and prove memory safety and well-formedness. Essentially, these come for free from TVLA. Additional instrumentation that we had to provide concerned type information (`lnode`, `lnodep`), the location to which pointer variables point (`ptr[z]`), and the fact that each record always has a `next` component (`hasn`).

In Figure 3, we show a sample shape graph, where summary nodes are denoted by double lines, definite edges by solid arrows, and 1/2 edges, which may or may not be there, by dotted arrows. Variable `t` points to an element in the middle of the list and is to be deleted using the code above. The snapshot is taken after the first iteration of the loop, where `lpp` was advanced once. The two *pairs* of summary nodes (double circles) represent any number (at least 1) of list elements before and after `t`. Each pair would be a single node in the standard TVLA memory model. Also observe, that `lpp` indeed points to the `n` *component* of the list element pointed to by `x`.

4 Coarse-grained Semantics and Analysis

The fine-grained model from the last section has a very explicit view of the heap and allows for very detailed modelling. The number of individuals, though, in a logical structure is a multiple of the number occurring in standard TVLA based analyses because each component of a record is modeled by a separate individual. On top of that, care must be taken that individuals belonging to the same record — encoded as the outermost boxes in Figure 2 — are kept together, something not supported by standard TVLA.

Fortunately, we can atone for these drawbacks by exploiting the fact that the structure of a record is completely static. Once allocated, the interpretation of predicates in **Sel** never changes, only the $*$ predicate does. This observation suggests an encoding of records as single nodes after all, rather than representing them explicitly through a linked set of nodes — as in the one-node-per-record paradigm. Still, pointers to the head of a record need be distinguished from pointers to components. We do so by *parameterizing* the $*$ predicate. For example, if $*[p, n]$ is true of two individuals u and u', it means that the p-component of the record modeled by u holds a pointer to the n-component of the record modeled by u'. Analogously, we parameterize the unary predicates encoding pointer variables: if $x[n]$ holds of individual u, it means that x holds a pointer to the n-component of the record modeled by u. In the special case (which in practice is the most common) of a pointer to the head of a record, we write $x[\varepsilon]$.

Addresses of variables can be handled by adding one individual per variable exactly like in the fine-grained model. In order to simplify the presentation, though, we here omit addresses of pointer variables. Unlike in the fine-grained semantics, the unary predicate $x[\varepsilon]$ now holds for the individual pointed to by

st	φ^{st}_p	
$x = \mathbf{0}$	$\varphi^{st}_{x[\pi]}(v)$	$= 0$
$*x = \mathbf{0}$	$\varphi^{st}_{*[\pi,\pi']}(v_1, v_2)$	$= *[\pi, \pi'](v_1, v_2) \wedge \neg x[\pi](v_1)$
$x \to s = \mathbf{0}$	$\varphi^{st}_{*[\pi.s,\pi']}(v_1, v_2)$	$= *[\pi.s, \pi'](v_1, v_2) \wedge \neg x[\pi](v_1)$
$x = y$	$\varphi^{st}_{x[\pi]}(v)$	$= y[\pi](v)$
$x = *y$	$\varphi^{st}_{x[\pi]}(v)$	$= \exists\, v' : \bigvee_{\pi' \in \Pi} y[\pi'](v') \wedge *[\pi', \pi](v', v)$
$x = \&y$	not supported	
$x = \&y \to s$	$\varphi^{st}_{x[\pi.s]}(v)$	$= y[\pi](v)$
$x = y \to s$	$\varphi^{st}_{x[\pi]}(v)$	$= \exists\, v' : \bigvee_{\pi' \in \Pi} y[\pi'](v') \wedge *[\pi'.s, \pi](v', v)$
$x = \mathbf{up}(y, t, s)$	$\varphi^{st}_{x[\pi]}(v)$	$= y[\pi.s](v)$
$*x = y$	$\varphi^{st}_{*[\pi,\pi']}(v_1, v_2)$	$= *[\pi, \pi'](v_1, v_2) \vee x[\pi](v_1) \wedge y[\pi'](v_2)$
$x \to s = y$	$\varphi^{st}_{*[\pi.s,\pi']}(v_1, v_2)$	$= *[\pi.s, \pi'](v_1, v_2) \vee x[\pi](v_1) \wedge y[\pi'](v_2)$

Table 2: Predicate update formulas. Here, $\varphi_{x[\pi]}$ and $\varphi_{*[\pi,\pi']}$ denote rule schemes and stand for one rule per instance of π.

the pointer x, rather than for the location of x itself. Thus, the standard TVLA model is obtained from this version of the coarse-grained model by restricting predicates to the forms $*[s, \varepsilon]$ and $x[\varepsilon]$, i.e., all pointers point to the *heads* of records.

Recall the notion of an access path of the previous section. Using access paths, we define coarse-grained states as logical structures over the following signature, $\mathcal{D}$, serving as our set of core predicates.

$$\mathcal{D} = \{x[\pi]/1 \mid x \in \mathbf{Id}, \pi \in \Pi_\varepsilon\} \cup \{*[\pi_1, \pi_2]/2 \mid \pi_1 \in \Pi, \pi_2 \in \Pi_\varepsilon\}$$

In order to complete the coarse-grained semantics, we provide the predicate update formulas for the predicates in $\mathcal{D}$. The update formulas shown in Table 2 constitute the state transformers both for the concrete and for the abstract semantics. These formulas are more concise than those of the fine-grained model. The update for the **up**() operation, e.g., only requires updating the predicates $x[\pi]$ to be true whenever the corresponding $y[\pi.s]$ used to be true. In the case of memory allocation, the effect of $x = \mathbf{malloc}(t)$ is to extend the universe with one fresh individual for which only the predicate $x[\varepsilon]$ holds.

Analysis. As for the fine-grained semantics, we implemented the coarse-grained transformers inside TVLA. As an example, we analyzed a program that first generates an hlist using the expanded `hlist_add` macro, which was already used in Figure 1, then iterates to some arbitrary point, and then deletes the element there using the `hlist_del` macro. The concrete C code of these macros is available from the `list.h` file of the current Linux distribution.

Being able to handle hlists is mandatory for verifying absence of races in programs such as in Figure 1. In our TVLA implementation, we parameterized the $*$ predicate with source and target components as described in the semantics. Other than that, we could migrate existing analysis specifications for

doubly-linked lists to hlists. The analysis of doubly linked lists uses, e.g., the instrumentation predicate which says that first following the pointers n and then p yields the same element. This predicate now is migrated to a predicate $c[n, p]$ stating that following $*[n, \varepsilon]$ and then $*[p, n]$ results in the same individual.

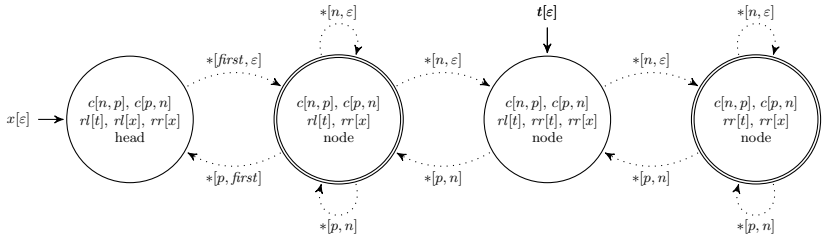

Fig. 4: Shape graph obtained during hlist traversal.

The sample shape of Figure 4 shows a situation where `t` points to the middle of an hlist. Note that there are two sorts of reachability: forward (`tr`, `rr[z]`) and backward (`tl` and `rl[z]`). This shows that `t` is indeed in the middle, well-formedness follows from the `c[]` predicates. Finally, observe that the back pointers either point to the `first` component of the head `x`, `*[p,first]`, or to the `n` component of a predecessor element (`*[p,n]`). In contrast, the forward pointer always points to the head of a record (`*[n,`ε`]`).

We successfully verified well-formedness and memory safety for the hlist example. Before we proceed to the example program of Figure 1, we investigate the formal relation between the fine-grained and the coarse-grained model in terms of expressiveness.

5 Fine-grained versus Coarse-grained

Since the fine-grained model is more detailed, it is able to simulate the coarser one in a sense to be made explicit now.

We start by defining a mapping g from a coarse-grained structure $S_c = (U_c, \iota_c)$ into a fine-grained structure $g(S_c) = (U_f, \iota_f)$. The set of individuals of $g(S_c)$ is given by

$$U_f = \mathbf{Id} \cup \{u_\pi \mid u \in U_c, \pi \in \Pi_\varepsilon(\mathbf{type}(u))\}$$

The interpretation function ι_f then is given by:

$$\begin{aligned}
\iota_f(*)(u_\pi, u'_{\pi'}) &\text{ iff } \iota_c(*[\pi, \pi'])(u, u') \\
\iota_f(*)(x, u_\pi) &\text{ iff } \iota_c(x[\pi])(u) \\
\iota_f(x)(v) &\text{ iff } v = x \\
\iota_f(s)(v, v') &\text{ iff } \exists u_{\pi.s} \in U_f.\ v = u_\pi \wedge v' = u_{\pi.s}
\end{aligned}$$

where $\mathbf{type}(u) = t$ if u was created by $\mathbf{malloc}(t)$. Also we assume that S_c *respects types*, i.e., there are no pointers from or to a π component of node u if $\pi \notin \Pi(\mathbf{type}(u))$.

Since we deal with two different vocabularies, $\mathcal{C}$ and $\mathcal{D}$, on top of the mapping g between structures, a mapping $\mathcal{T}$ is required which translates formulas. Let φ be a FO(TC) formula over $\mathcal{D}$. The translation $\mathcal{T}$ commutes with boolean connectives and additionally is defined by:

$$\begin{aligned}
\mathcal{T}(*[\pi_1, \pi_2](v_1, v_2)) &= \exists v_1', v_2', v : \pi_1(v_1, v_1') \wedge \pi_2(v_2, v_2') \wedge *(v_1', v_2') \\
\mathcal{T}(x[\pi](v)) &= \exists v', v'' : x(v') \wedge \pi(v, v'') \wedge *(v', v'') \\
\mathcal{T}(\exists v : \varphi) &= \exists v : \mathit{head}(v) \wedge \mathcal{T}(\varphi) \\
\mathcal{T}((TC\, v_1, v_2 : \varphi)(v_3, v_4)) &= (TC\, v_1, v_2 : \mathit{head}(v_1) \wedge \mathit{head}(v_2) \wedge \mathcal{T}(\varphi))(v_3, v_4)
\end{aligned}$$

where for $\pi = s_1.\cdots.s_k \in \Pi$, the formula $\pi(v_0, v_k)$ is given by

$$\exists v_1, \ldots, v_{k-1} :\ s_1(v_0, v_1) \wedge \ldots \wedge s_k(v_{k-1}, v_k)$$

and where *head* holds for heads of records in a fine-grained structure only. A node is a head, if it is not the location of a variable and if it has no incoming **Sel** edge. The following theorem states that this translation preserves the valuation of formulas and that it commutes with state transformers, i.e., with predicate update formulas.

Theorem 1. *Let S_c be a type-respecting, coarse-grained logical structure and $S_f = g(S_c)$ the corresponding fine-grained structure. Then we have:*

1. *For every closed FO(TC) formula φ over $\mathcal{D}$, $[\![\varphi]\!]^{S_c} = [\![\mathcal{T}(\varphi)]\!]^{S_f}$.*
2. *For every basic statement st, $g([\![st]\!]_c(S_c)) = [\![st]\!]_f(S_f)$.*

Proof. For an induction argument, we prove the statement for open formulas. Let $Z_c\colon \mathsf{Var} \to U_c$ be an assignment of logical variables to individuals in the coarse-grained universe; we define $Z_f = g(Z_c)\colon \mathsf{Var} \to U_f$ as an assignment selecting the head u_ε for each record u. We show $[\![\varphi]\!]^{S_c}(Z_c) = [\![\mathcal{T}(\varphi)]\!]^{S_f}(Z_f)$ by induction on φ. For the core predicates, we compute for $u = Z_c(v)$:

$$\begin{aligned}
[\![x[\pi](v)]\!]^{S_c}(Z_c) &= \iota_c(x[\pi])(u) = \iota_f(*)(x, u_\pi) \\
&= \exists u' \in U_f :\ \pi(u_\varepsilon, u') \wedge \iota_f(*)(x, u') \\
&= [\![\mathcal{T}(x[\pi](v))]\!]^{S_f}(Z_f)
\end{aligned}$$

And analogously for the binary predicates. We need to further consider cases for $\wedge$, $\exists$, $\neg$, and TC (as the rest follows from DeMorgan's Laws). Conjunction and negation are obvious, while existential quantification and transitive closure rely on the restriction of quantification to heads of records. We consider existential quantification, for which we observe:

$$\begin{aligned}
[\![\exists v : \varphi]\!]^{S_c}(Z_c) &= \exists u \in U_c : [\![\varphi]\!]^{S_c}(Z_c[v \mapsto u]) = \exists u \in U_c : [\![\mathcal{T}(\varphi)]\!]^{S_f}(Z_f[v \mapsto u_\varepsilon]) \\
&= \exists u' \in U_f : \mathit{head}(u') \wedge [\![\mathcal{T}(\varphi)]\!]^{S_f}(Z_f[v \mapsto u']) \\
&= [\![\exists v : \mathit{head}(v) \wedge \mathcal{T}(\varphi)]\!]^{S_f}(Z_f) = [\![\mathcal{T}(\exists v : \varphi)]\!]^{S_f}(Z_f)
\end{aligned}$$

This completes the proof of the first statement. For the second statement, let $S_c = (U_c, \iota_c)$ denote a coarse-grained logical structure. We do a case distinction on the form of basic statements.

Consider, e.g., the statement st given by $x = \mathbf{up}(y, t, s)$. If it exists, let $u \in U_c$ denote the unique individual for which $\iota_c(y[\pi.s])$ holds for some access path π. Then $[\![st]\!]_c(S_c) = S'_c = (U'_c, \iota'_c)$ where $U'_c = U_c$ and ι'_c equals ι_c up to the predicate $x[\pi]$, which is updated such that $\iota'_c(x[\pi])(u')$ holds iff $u' = u$. Let $S_f = g(S_c)$ denote the fine-grained structure corresponding to S_c. This generates for $u \in U_c$ the head $u_\varepsilon \in U_f$ as well as its components, including u_π and $u_{\pi.s}$. Since we assumed that $\iota_c(y[\pi.s])(u)$ is true in S_c, we know that $\iota_f(*)(y, u_{\pi.s})$ must hold in S_f. Thus, $[\![st]\!]_f(S_f) = S'_f = (U'_f, \iota'_f)$ where the set of individuals are the same as S_f and ι'_f equals ι_f up to the predicate $*$ which now additionally holds for the pair (x, u_π). Ultimately, the only change to S_f and S_c is that $\iota'_c(x[\pi])(u)$ holds in S'_c and $\iota'_f(*)(u, u_\pi)$ holds in S'_f. As this is in accord with the definition of g, we conclude that $g(S'_c) = S'_f$. This holds also if $\iota_c(y[\pi.s])$ is false everywhere, in which case $S'_c = S_c$ and $S'_f = S_f$. □

The theorem effectively constitutes a simulation result between fine-grained and coarse-grained semantics. Notice that the restriction of quantified variables to heads of records in the translation $\mathcal{T}$ is an important one. It also demonstrates exactly *how* fine-grained structures are finer: they can talk about record components explicitly and quantify over them, while components occur only implicitly in the coarse-grained model.

Part 1 of Theorem 1 can be lifted to abstract states as well. Assume an abstract, three-valued coarse-grained structure S^3_c and any two-valued coarse-grained structure S^2_c such that $S^2_c \sqsubseteq S^3_c$ using the embedding order of [27]. Then any formula ψ of FO(TC) that holds for S^3_c also holds for S^2_c by the Embedding Theorem. By Theorem 1, $\mathcal{T}(\psi)$ holds in $S_f = g(S^2_c)$. If S^3_c was obtained by the set A of abstraction predicates, then ψ will also hold in the canonical abstraction of S_f using $\mathcal{T}(A)$ as abstraction predicates. Lifting part 2 of Theorem 1 is far more involved, because it needs to take materialization strategies into account.

6 Application

Let us finally consider the motivating program from the Introduction. Its code is shown in Figure 1 and a typical memory configuration in Figure 2. In order to argue about data races in the presence of privatization, reachability information is crucial. In particular, one must reason about reachability along different embedded lists. For instance, in Figure 2, only the first and the third node are in the queue, whereas all three are in the list.

The TVLA tool does not natively support computations on predicates as necessary to conveniently express the string manipulation on access paths as used in the update formulas of Table 2. This makes the implementation cumbersome and look clumsy in places. Also, it introduces a lot of superfluous predicates and coercion constraints greatly slowing down the tool. This, however, is not a

principal restriction of our memory model but the lack of tool support. Therefore, we had to settle for a proof of concept implementation where the cleanup queue is actually a singly-linked list.

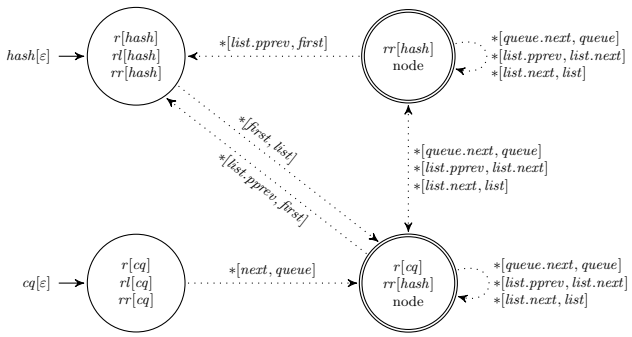

Fig. 5: Shape graph obtained while analyzing overlapping data-structure.

First, we analyzed a program creating a structure like that of Figure 2 from scratch. This amounts to iterating the code

```
n = malloc(sizeof(node));
hlist_add_head(&n→list,&hash);
if (?) list_add(&n→queue, &cq);
```

After this loop, four shapes are obtained, the most general of which is shown in Figure 5. It shows that (i) all nodes are reachable from `hash`, which is the head of the hlist component, a fact indicated by `rr[hash]`; and (ii) only some nodes are reachable from `cq`, indicated by `r[cq]`. This is the arbitrary subset of nodes added to the cleanup queue. Also it shows the $*$ predicates with parameters like `list.pprev`, denoting the pprev component of the hlist component of a node. Recall that `rr[x]` (`rl[x]`) means reachability from `x` along forward (backward) pointers in a doubly-linked list, while `r[x]` is just singly-linked list reachability — which is how we implement the cleanup queue. The precise definitions can be found at `http://www7.in.tum.de/~joba/overlap.tgz`.

Subsequently, the elements of the queue are to be removed from the hlist component using

```
list_for_each_entry(n, &cq, queue) { hlist_del(&n→list); }
```

Here, the challenge for the analysis is the *change of views* implied by traversing the queue and then removing from the hlist. A lot of reachability information is lost; in fact, properties like absence of memory leaks cannot be guaranteed by this analysis. Still, we are able to prove that an element is deleted from the queue using the very same routine that was used in a non-embedded record in

Section 4. Thus, we can infer that the element is no longer reachable from the thread-shared data.

Again, the analysis specifications are available online. Even in this most complicated scenario, the analysis time was just a few seconds.

7 Related Work

The body of work on shape analysis is too large to do equal justice to all techniques. Approaches based on regular model checking [5], symbolic backwards reachability analysis [1], or decision procedures such as [6] seem not to have dealt with the analysis of low-level system code, much less with overlapping records. There are a number of approaches that make use of numeric reasoning to deal with pointer arithmetic. While pioneered by Deutsch [15], who used numeric domains to constrain access paths, Gulwani and Tiwari [19] provide a C semantics which perhaps is even more explicit about blocks and offsets as ours. However, it is unable to deal with structures such as doubly-linked lists. Recent work [18] combines numeric and shape domains. It is focused on tracking partition sizes to prove memory safety and sometimes even termination in presence of arrays of dynamically allocated structures. So far, however, none of the above treats overlapping records.

More direct approaches to shape analysis are based on either TVLA [27] or separation logic [26]. As stated before, most work in the TVLA setting focuses on higher-level programming languages; however, Dor's thesis [16] and subsequent work provided a semantics of low-level C similar to our fine-grained semantics and program analyses based on this semantics. These analyses are mostly concerned with string manipulations.

As for separation logic based approaches which deal with a low-level C semantics and with pointer arithmetic, one early work is [8], which however specifically targets the data-structure of multiword lists. Berdine et al. [3] present a shape analysis of composite data-structures which can reason about lists of lists. They explicitly identify shape analysis of overlapping or embedded structures as presented here as a limitation to their approach. Also, Chang and Rival [9] present a shape analysis based on separation logic and user-specified data-structure specifications called checkers. It also treats combinations of numerical and shape domains, but overlapping records are not considered. Despite some impressive improvements recently [7, 31], in particular concerning scalable shape analyses of real code, a formal treatment of overlapping records has yet to be reported. In addition, most of that work focuses exclusively on memory safety rather than on subtle reachability problems as we face.

Separation logic is also used in the broader context of modular verification and extended static checking. There, one relies on specifications of components, and the analysis operates under the assumption that other components behave as specified [2, 14, 17]. The fine-grained memory model we use for shape analysis is also used by the VCC C verifier [12]; in particular, it uses an implicit type-system to verify that distinct pointers do not reference overlapping objects [13].

In the Havoc verifier [10], a particular reachability predicate is employed which also works on a semantics resembling ours, but is much more numeric in nature, focused on pointer arithmetic. Other techniques exist for dealing with the heap in modular verification, including ownership [11], which is used by Spec# and Java/JML; dynamic frames [21, 29], which is used by VeriCool 1 and Dafny; and implicit dynamic frames [30], which are used in VeriCool 3 and Chalice.

Our interest in shape analysis of overlapping records is derived from attempts to verify absence of data races in low-level C. In static race detection, dynamic memory is treated at a fairly superficial level by *blobbing* together objects into static allocation sites. There are techniques for verifying mutually exclusive access to heap objects when each record contains its own dedicated lock [24]; and analyses relying on reachability information, such as *disjoint reachability analysis* [23] and *region analysis* [28], have been employed to ensure correct synchronization of accesses to disjoint regions of dynamically allocated memory. These analyses, however, cannot deal with object privatization and overlapping structures as occur in our example. On the other hand, by virtue of not tracking the state of the heap at each program point, such analyses can be directly used in a concurrent setting, while our approach requires adaptations of the TVLA approach to handle concurrency [4, 22].

8 Conclusion

We presented a shape analysis for overlapping data-structures, which are ubiquitous in low-level systems code. Using our prototype implementation we were able to establish subtle reachability properties as required, e.g., for reasoning about data races in system code with overlapping records.

For that, we introduced two refinements of existing memory models. This enabled us to implement both approaches within the TVLA framework. Accordingly, our analysis will benefit from any future improvements of the TVLA tool. However, as dynamic manipulation of predicates is not natively supported by the TVLA tool, a new front-end and/or tool extension is desirable as future work.

The step from fine-grained to coarse-grained semantics is essentially a technique of encoding statically known parts of graph structures like the internal structure of records into syntax. Somehow similar, separation logic based approaches rely on inductively defined predicates capturing data-structures. This connection might be exploited to enable the use of different formalisms for different parts of the heap in a common setting.

References

1. Abdulla, P.A., Bouajjani, A., Cederberg, J., Haziza, F., Rezine, A.: Monotonic abstraction for programs with dynamic memory heaps. In: CAV'08. LNCS, vol. 5123, pp. 341–354. Springer (2008)
2. Barnett, M., Chang, B., DeLine, R., Jacobs, B., Leino, K.: Boogie: A modular reusable verifier for Object-Oriented programs. In: FMCO'06. LNCS, vol. 4111, pp. 364–387. Springer (2006)

3. Berdine, J., Calcagno, C., Cook, B., Distefano, D., OHearn, P., Wies, T., Yang, H.: Shape analysis for composite data structures. In: CAV'07. LNCS, vol. 4590, pp. 178–192. Springer (2007)
4. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. In: CAV'08. LNCS, vol. 5123, pp. 399–413. Springer (2008)
5. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: CAV'00. LNCS, vol. 1855, pp. 403–418. Springer (2000)
6. Bouillaguet, C., Kuncak, V., Wies, T., Zee, K., Rinard, M.C.: Using first-order theorem provers in the Jahob data structure verification system. In: VMCAI'07. LNCS, vol. 4349, pp. 74–88. Springer (2007)
7. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL'09. pp. 289–300. ACM Press (2009)
8. Calcagno, C., Distefano, D., OHearn, P., Yang, H.: Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In: SAS'06. LNCS, vol. 4134, pp. 182–203. Springer (2006)
9. Chang, B.Y.E., Rival, X.: Relational inductive shape analysis. In: POPL'08. pp. 247–260. ACM Press (2008)
10. Chatterjee, S., Lahiri, S., Qadeer, S., Rakamarić, Z.: A low-level memory model and an accompanying reachability predicate. Int J Softw Tools Technol Transfer 11(2), 105–116 (2009)
11. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA'98. pp. 48–64. ACM Press (1998)
12. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: TPHOL'09. LNCS, vol. 5674, pp. 23–42. Springer (2009)
13. Cohen, E., Moskal, M., Tobies, S., Schulte, W.: A precise yet efficient memory model for C. In: SSV'09. ENTCS, vol. 254, pp. 85–103. Elsevier (2009)
14. Condit, J., Hackett, B., Lahiri, S.K., Qadeer, S.: Unifying type checking and property checking for low-level code. In: POPL'09. pp. 302–314. ACM Press (2009)
15. Deutsch, A.: Interprocedural may-alias analysis for pointers: beyond k-limiting. In: PLDI'94. pp. 230–241. ACM Press (1994)
16. Dor, N.: Automatic Verfication of Program Cleanness. Master's thesis, Tel Aviv University (2003)
17. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI'02. pp. 234–245. ACM Press (2002)
18. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: POPL'09. pp. 239–251. ACM Press (2009)
19. Gulwani, S., Tiwari, A.: An abstract domain for analyzing heap-manipulating low-level software. In: CAV'07. LNCS, vol. 4590, pp. 379–392. Springer (2007)
20. Jonkers, H.B.M.: Abstract storage structures. In: Algorithmic Languages. pp. 321–343. IFIP (1981)
21. Kassios, I.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: FM'06. LNCS, vol. 4085, pp. 268–283. Springer (2006)
22. Manevich, R., Lev-Ami, T., Sagiv, M., Ramalingam, G., Berdine, J.: Heap decomposition for concurrent shape analysis. In: SAS'08. LNCS, vol. 5079, pp. 363–377. Springer (2008)
23. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: POPL'07. pp. 327–338. ACM Press (2007)
24. Pratikakis, P., Foster, J.S., Hicks, M.: Existential label flow inference via CFL reachability. In: SAS'06. LNCS, vol. 4134, pp. 88–06. Springer (2006)

25. Reps, T.W., Sagiv, S., Loginov, A.: Finite differencing of logical formulas for static analysis. In: ESOP'03. LNCS, vol. 2618, pp. 380–398. Springer (2003)
26. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS'02. pp. 55–74. IEEE Press (2002)
27. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. TOPLAS 24(3), 217–298 (2002)
28. Seidl, H., Vojdani, V.: Region analysis for race detection. In: SAS'09. LNCS, vol. 5673, pp. 171–187. Springer (2009)
29. Smans, J., Jacobs, B., Piessens, F.: VeriCool: an automatic verifier for a concurrent Object-Oriented language. In: FMOODS'08. LNCS, vol. 5051, pp. 220–239. Springer (2008)
30. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: ECCOP'09. LNCS, vol. 5653, pp. 148–172. Springer (2009)
31. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable shape analysis for systems code. In: CAV'08. LNCS, vol. 5123, pp. 385–398. Springer (2008)