HAL
open science

Hierarchical Shape Abstraction of Dynamic Structures in Static Blocks

Pascal Sotin, Xavier Rival

To cite this version:

Pascal Sotin, Xavier Rival. Hierarchical Shape Abstraction of Dynamic Structures in Static Blocks. APLAS 2012 - Asian Conference on Programming Languages And Software, Dec 2012, Kyoto, Japan. pp.131-147, 10.1007/978-3-642-35182-2_10 . hal-00760427

HAL Id: hal-00760427

https://inria.hal.science/hal-00760427v1

Submitted on 8 Jan 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Hierarchical Shape Abstraction of Dynamic Structures in Static Blocks *

Pascal Sotin and Xavier Rival

INRIA Paris–Rocquencourt / CNRS / École Normale Supérieure, Paris, France

Abstract. We propose a hierarchical shape abstract domain, so as to infer structural invariants of dynamic structures such as lists living *inside* static structures, such as arrays. This programming pattern is often used in safety critical embedded software as an alternative to dynamic memory allocation. Our abstract domain precisely describes such hierarchies of structures. It combines several instances of simple shape abstract domains, dedicated to the representation of elementary shape properties, and also embeds a numerical abstract domain. This modular construction greatly simplifies the design and the implementation of the abstract domain. We provide an implementation, and show the effectiveness of our approach on a problem taken from a real code.

1 Introduction

Safety critical embedded systems as found in avionics should meet safety requirements fixed by regulation standards [12]. In particular, software providers should supply evidence that the real time applications will not fail due to resource exhaustion. In practice, this constraint forbids the use of dynamic memory allocation in highly critical software. Though, this does not mean that dynamic data-structures (that is linked structures where pointers may be modified at any time in the execution of the program) cannot be used: indeed, structure elements may be allocated statically (in arrays or in other static sections) and links across elements may be re-computed at any time. Such statically allocated dynamic structures are found in many programs such as the USB driver considered in [22] or the multi-threaded avionic software considered in [21].

In the last decade, dramatic progresses have been accomplished in the verification of absence of runtime errors in safety critical programs [3, 2], yet statically allocated dynamic structures are still very challenging for static analysis tools. Static analyzers such as Astrée [3, 2] do offer some support for the summarization of large memory regions, but will not capture inductive properties of linked data structures such as lists. Inferring that such a structure is a well formed list may require maintaining large disjunctions of cases depending on the elements order. Failure to do so would lead to false alarms, as proving the absence of runtime errors may require proving that the dynamic structures are well formed.

* The research leading to these results has received funding from the European Research Council under the FP7 grant agreement 278673, Project MemCAD.

On the other hand, shape analysis techniques are very smart at summarizing unbounded linked structures [24, 11, 1] but typically do not track the fact that some pieces of data are stored in a fixed, static block, which may be accessed to as an array. Furthermore, existing shape analyses cannot be interfaced with a powerful numerical domain such as the one used in [2].

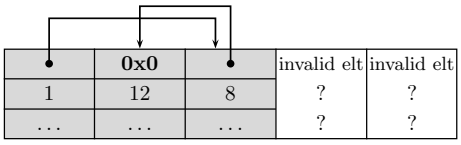
In this paper, we exploit the ability of the shape analysis framework proposed in [5, 17] to attach numeric predicates to shape graph "nodes" that represent concrete values of arbitrary size (addresses or contents of physical memory cells) in order to tie a complex property to a memory region, in a fully modular way from the static analysis design point of view. In particular, the contents of a static region (as a sequence of bytes) is represented by a symbolic variable, which may be characterized in a value abstract domain; we can then choose to consider this sequence of bytes as a "store inside the store", and let another instance of our shape abstract domain take care of its abstraction. In this setup, the analysis uses two instances of the shape abstract domain: one is used to abstract the memory states, whereas the other is used in order to abstract the contents of the static region. The main advantage of this technique is the modularity of the abstraction, as it alleviates the need for a complex monolithic abstract domain expressing all data-structure invariants. It also allows to reuse the abstract domain of [5] as is, and can be combined with a powerful numerical abstract domain. Our main contributions are (1) the design of a framework for the abstraction of hierarchical memory states, where some memory regions are viewed as *sub-memories*, (2) the integration of an array abstraction in a shape abstract domain, to automatically infer sub-memory boundaries and (3) the implementation of the hierarchical abstraction in the MemCAD static analyzer, which implements the framework of [5] using the Apron [16] numerical domain library, and the verification of a simplified excerpt from the avionic code discussed in [21] (leaving out features out of the scope of the issue considered in this paper).

2 Running example

Fig. 1 describes the function considered in our running example, a simplified excerpt of the safety critical application considered in [21]. The data-type (Fig. 1a) is a form of singly linked list (which represents message queues), yet all elements manipulated in the program live in a global array free_pool [100]. A fragment of a concrete state is shown in Fig. 1b. At any point in the execution some of the array elements are active and are members of an existing list structure (the gray elements in the figure) whereas the others are "invalid". We call such an array a *free-pool*. Furthermore, other structural invariants are maintained throughout the program: the list is ordered by increasing priorities and its first and last cells are respectively pointed to by hd and tl. The code inserts a set of elements in the list stored in the free-pool. For each element, it searches the position and performs the insertion. Several cases were omitted for the sake of concision, and we only focus on the case of an insertion within the list, after a traversal to determine the right position. The goal of the analysis discussed in

```
1 typedef struct Cell {
    struct Cell *next;
3   int prio;
    /* other fields */
5 } Cell;
  Cell free_pool[100];
```

(a) Data-type and free-pool.



(b) A concrete structure.

```
  void main() {
2   int free_idx;
    Cell *hd, *tl;
4   hd = null;
    tl = null;
6   for(free_idx = 0; free_idx < 100; free_idx++) {
      int priority; /* = computation(); */
8     if (hd == null) { /* insert first cell */ }
      else if (priority <  hd->prio) { /* insert as head  */ }
10    else if (priority >= tl->prio) { /* insert as queue */ }
      else {
12      Cell *cur = hd;
        while(priority >= cur->next->prio) { cur = cur->next; }
14      assert(cur != tl); // position found
        free_pool[free_idx].next = cur->next;
16      free_pool[free_idx].prio = priority;
        cur->next = &free_pool[free_idx]; } } }
```

(c) Insertion routine.

Fig. 1: A dynamic structure in a static area

the paper is to establish both the preservation of the list structural invariant, and memory safety. In particular, the inner loop should cause no null pointer dereference (although the loop condition does not explicitly check that cur is not null). Moreover, it should verify the assertion at line 14, i.e., that the insertion is not made at the tail of the list in that branch.

Existing memory abstractions: Fig. 1b shows a (simplified) concrete state encountered at the head of the main loop in our example program. At the end of the execution, the list has length 100 and occupies the whole array. Due to the size of the structure, an efficient analysis requires it be *summarized*. The abstraction of [17] allows to summarize the whole list into a predicate which expresses that hd points to a list. However, that abstraction fails to capture the fact the list is allocated inside an array, and does not allow to analyze accesses using array selectors. In the other hand, preserving a fully precise abstraction of the array would not allow the summarization of the list and would require a large case analysis over the list elements ordering, which would be prohibitively expensive.

Fig. 2: Hierarchical abstraction

Hierarchical abstraction. The limitation of all the abstractions examined so far is that they fail to capture *both* the list structure and the fact that it lives inside a contiguous memory region. A solution is to perform a two steps abstraction:

1. The whole array occupies an 800 bytes long contiguous region, which can be abstracted by a single points-to predicate $\alpha \mapsto \beta$ where symbolic variables α and β respectively represent the address of the array and its *whole* contents viewed as a sequence of bytes, as shown in the left part of Fig. 2, whereas variable hd points into the array at some offset o (i.e. from base address α).
2. Symbolic variable β which denotes the array contents can be constrained by any abstraction over the sequence of bytes it represents. The trick is then to view it as a memory state in itself (which we later refer to as the *sub-memory*), and apply a classical shape abstraction to it, which expresses that it stores a well-formed singly linked list structure, the first element of which is at offset o, as shown in the right part of Fig. 2. This abstraction relies on the user-defined **list** inductive predicate below:

$$\alpha \cdot \mathbf{list} := (\mathbf{emp} \wedge \alpha = 0) \vee (\alpha \cdot \mathrm{next} \mapsto \beta_0 * \alpha \cdot \mathrm{prio} \mapsto \beta_1 * \beta_0 \cdot \mathbf{list}) \,.$$

In this view, the analysis should use *two levels* of memory abstractions: one to describe the main memory, and another one to describe the contents of a contiguous region of the memory, viewed as a sub-memory. In that sense, our abstraction is *hierarchical*. Furthermore, we are going to show that both abstractions may share a single implementation, where the memory abstraction consists of a general and parametric shape analysis abstract domain. This approach handles arbitrary nesting of dynamic and static structures (e.g., lists of arrays containing lists...). We formalize this abstraction in Sect. 3.3.

Static analysis in the hierarchical abstract domain. Our static analysis should establish properties of array initialization and list construction routines as well as common list operations (traversal, insertion...). Overall, the analysis algorithms in the hierarchical shape abstract domain are standard shape analysis algorithms including unfolding of inductive definitions and widening over shape graphs [5]. However, a number of specific issues need be solved, such as:

- reasoning about array regions (and about the border between the free zone and the active list) effectively requires integrating array analysis techniques such as those proposed in [8] into a shape abstract domain;
- designing clear interfaces between domains, so as to make the analysis fully modular, letting the main memory abstraction devolve the analysis of operations to the sub-memory abstraction when possible.

Those issues will be considered carefully in Sect. 4.

3 A Hierarchical Shape Abstract Domain

We formalize our abstract domain in this section, and recall elements of the abstract domain introduced in [6, 5] it is based on.

3.1 A Shape Graph Abstract Domain

Concrete model. Intuitively, a *concrete store* can be viewed as a partial function σ from addresses ($a \in \mathbb{A}$) into values ($\mathbb{V}$, where $\mathbb{A} \subseteq \mathbb{V}$). In fact, the structure of memory states is more complex as values of various sizes may be read, so a store is actually characterized by its domain $\mathbf{dom}(\sigma) \in \mathcal{P}(\mathbb{A})$, and the read operation, which maps pair of addresses $a < a' \in \mathbb{A}$ to the value $\mathbf{read}(\sigma, a, a') \in \mathbb{V}$ that can be read in σ between a and a', when $[a, a'[\sqsubseteq \mathbf{dom}(\sigma)$. A concrete value $v \in \mathbb{V}$ thus consists of a sequence of bytes.

Abstraction based on shape graphs. In the abstract level, *symbolic variables* (noted as Greek letters $\alpha, \beta, \ldots \in \mathbb{V}^\sharp$) represent concrete values. These symbolic variables may appear in constraints on the stores structure, on their contents, and possibly simultaneously on both. A *shape graph* $G \in \mathbb{D}^\sharp_{\mathrm{G}}$ describes the structure of concrete stores, as a *separating conjunction* of predicates, called *edges*, which express e.g., that some symbolic variable α is the address of a memory cell containing a value abstracted by another symbolic variable β: this constraint is described by a *points-to* edge of the form $\alpha \mapsto \beta$ (the more general format of points-to edges is shown below). Therefore, concretization $\gamma_{\mathrm{G}}(G)$ of shape graph $G \in \mathbb{D}^\sharp_{\mathrm{G}}$ is defined indirectely. Instead of returning a set of stores, it returns a set of pairs (σ, ν) where $\nu \in \mathbb{V}^{\mathbb{I}} = \mathbb{V}^\sharp \to \mathbb{V}$ is a *valuation*, mapping each symbolic variable to the concrete value it abstracts, i.e., performing a physical mapping of the shape graph.

Shape graphs and concretization. The abstract domain is parameterized by the data of a finite set $\mathbb{I}$ of inductive definitions, such as the **list** definition shown in Sect. 2. The complete grammar of shape graphs is defined below:

$$
\begin{array}{rll}
G ::= & e_0 * e_1 * \ldots * e_k & \text{separating conjunction} \\
e ::= & \alpha_{[o_0, o_1[} \mapsto \beta + o_2 & \text{points-to edge } (\alpha, \beta \in \mathbb{V}^\sharp) \\
| & \alpha \cdot \iota & \text{inductive edge} \\
| & \alpha \cdot \iota \mathrel{*\!\!=} \beta \cdot \iota & \text{segment edge}
\end{array}
$$

Points-to edge $\alpha_{[o_0, o_1[} \mapsto \beta + o_2$, where o_0, o_1, o_2 are linear expressions over symbolic variables, describes a contiguous region between the addresses represented by $\alpha + o_0$ and $\alpha + o_1$ and storing the value represented by $\beta + o_2$ (thus its size corresponds to $o_1 - o_0$). Edge $\alpha \cdot \iota$ abstracts complete structures described by inductive definition ι, at address α. Segment $\alpha \cdot \iota \mathrel{*\!\!=} \beta \cdot \iota$ abstracts *incomplete* structures, that is a structures starting at address α with a *hole* at address β, i.e. a missing sub-structure at address β. The semantics of inductive and segment

edges is defined by syntactic unfolding of their definitions, using rewrite relation $\rightsquigarrow_{\text{unfold}}$. For instance, the unfolding rules of inductive definition **list** are:

$$\begin{array}{ll}
\alpha \cdot \mathbf{list} & \rightsquigarrow_{\text{unfold}} (\mathbf{emp} \wedge \alpha = 0) \\
\alpha \cdot \mathbf{list} & \rightsquigarrow_{\text{unfold}} (\alpha \cdot \text{next} \mapsto \beta_0 * \alpha \cdot \text{prio} \mapsto \beta_1 * \beta_0 \cdot \mathbf{list})\,. \\
\alpha \cdot \mathbf{list} \ast\!\!= \delta \cdot \mathbf{list} & \rightsquigarrow_{\text{unfold}} (\mathbf{emp} \wedge \alpha = \delta) \\
\alpha \cdot \mathbf{list} \ast\!\!= \delta \cdot \mathbf{list} & \rightsquigarrow_{\text{unfold}} (\alpha \cdot \text{next} \mapsto \beta_0 * \alpha \cdot \text{prio} \mapsto \beta_1 * \beta_0 \cdot \mathbf{list} \ast\!\!= \delta \cdot \mathbf{list}.)\,.
\end{array}$$

Concretization. We can now formalize the concretization $\gamma_{\text{G}} : \mathbb{D}^{\sharp}_{\text{G}} \to \mathcal{P}(\mathbb{M} \times \mathbb{V}\mathbb{a}\mathbb{l})$ of edges and of shape graphs. First, let us consider an edge e, and define its concretization $\gamma_{\text{G}}(e)$.

- If e is $\alpha_{[o_0,o_1[} \mapsto \beta + o_2$, then $(\sigma, \nu) \in \gamma_{\text{G}}(e)$ if and only if:

$$\begin{cases} \mathbf{dom}(\sigma) = [\nu_{\mathcal{L}}(\alpha + o_0), \nu_{\mathcal{L}}(\alpha + o_1)[\\ \mathbf{read}(\sigma, \nu_{\mathcal{L}}(\alpha + o_0), \nu_{\mathcal{L}}(\alpha + o_1)) = \nu_{\mathcal{L}}(\beta + o_2) \end{cases}$$

  where $\nu_{\mathcal{L}}$ denotes the extension of ν to linear expressions over symbolic variables (for instance, $\nu_{\mathcal{L}}(8 + \alpha + 2\beta) = 8 + \nu(\alpha) + 2\nu(\beta)$). This indeed captures the property that this points-to edge covers the range of addresses corresponding to symbolic range $[\alpha + o_0, \alpha + o_1[$ and contains symbolic value $\beta + o_2$.
- If e is either an inductive edge $\alpha \cdot \iota$ or a segment edge $\alpha \cdot \iota \ast\!\!= \beta \cdot \iota$, then its concretization is defined by unfolding; thus $(\sigma, \nu) \in \gamma_{\text{G}}(e)$ if and only if:

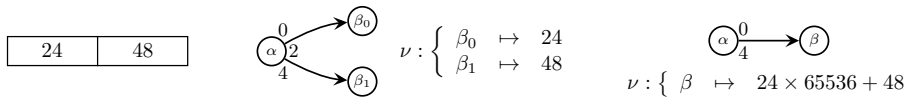
$$\exists G,\ e \rightsquigarrow_{\text{unfold}} G \wedge (\sigma, \nu) \in \gamma_{\text{G}}(G).$$

Concretization γ_{G} calculates the separating conjunction of the concretizations of the edges of shape graphs:

$$\gamma_{\text{G}}(e_0 * e_1 * \ldots * e_k) = \{(\sigma_0 \circledast \sigma_1 \circledast \ldots \circledast \sigma_k, \nu) \mid \forall i,\ (\sigma_i, \nu) \in \gamma_{\text{G}}(e_i)\}.$$

where $\circledast$ is the fusion of functions with disjoint domains ($\sigma_0 \circledast \sigma_1$ is defined if and only if $\mathbf{dom}(\sigma_0) \cap \mathbf{dom}(\sigma_1) = \emptyset$ and then $\mathbf{read}(\sigma_0 \circledast \sigma_1, a, a') = \mathbf{read}(\sigma_i, a, a')$ if $[a, a'[\subseteq \mathbf{dom}(\sigma_i)$). In general, due to inductive and segment edges, the concretization of a shape graph has to be defined as a least-fixpoint.

Examples. In practice, a contiguous concrete memory region (or *block*) may be described by one or more points-to edges from one single node, that denote fragments of that memory region. We call such a set of points-to edges starting from a same source node α a *segmentation* of the block. As a very simple example, Fig. 3 shows two possible segmentations (with two edges in Fig. 3b or with one edge in Fig. 3c) to abstract a concrete array of two unsigned 2-bytes integers shown in Fig. 3a. As a convention, we insert segmentation offsets between points-to edges (offsets 0, 2 and 4 in Fig. 3b) and destination offsets at the end of points-to edges (offsets +0 in Fig. 3b): Fig. 3b represents shape graph $\alpha_{[0,2[} \mapsto \beta_0 + 0 * \alpha_{[2,4[} \mapsto \beta_1 + 0$. Segmentations with linear expressions over symbolic variables as offsets rely on the same principle and will appear in Sect. 4.



(a) Concrete array. (b) Pair of edges. (c) Single edge.

Fig. 3: Segmentations representing an array of two unsigned short integers

3.2 Combination with a Value Domain

The advantage of the notion of shape graphs presented in Sect. 3.1 is that they allow a nice decomposition of the abstract domain: in particular, other forms of properties (such as arithmetic constraints) over symbolic variables can be described in a separate value abstract domain $\mathbb{D}^\sharp_{\mathrm{V}}$, with concretization $\gamma_{\mathrm{V}} : \mathbb{D}^\sharp_{\mathrm{V}} \to \mathcal{P}(\mathbb{V}\mathbb{a}\mathbb{l})$. An abstract memory state is a pair $(G, V) \in \mathbb{D}^\sharp_{\mathrm{G}} \times \mathbb{D}^\sharp_{\mathrm{V}}$, and concretizes into $\{\sigma \mid \exists \nu \in \gamma_{\mathrm{V}}(V),\ (\sigma, \nu) \in \gamma_{\mathrm{G}}(G)\}$.

In most cases, $\mathbb{D}^\sharp_{\mathrm{V}}$ can be chosen among numerical abstractions. For instance, in the case of Fig. 3b, the octagon abstract domain [20] allows to express relation $\beta_0 < \beta_1$ which is satisfied in valuation ν used to concretize that shape graph into the concrete store of Fig. 3a. However, non purely numerical abstractions may be used as well. For instance, in the case of the shape graph of Fig. 3c, symbolic variable β denotes an array of unsigned 2-bytes integers, and array specific abstractions may be used to abstract β; for instance, that array is sorted, so we could choose $\mathbb{D}^\sharp_{\mathrm{V}}$ in order to express array sortedness.

Moreover, a concrete state M also encloses an environment $E \in \mathbb{E} = \mathbb{X} \mapsto \mathbb{A}$ mapping program variables into addresses, thus is a pair $M = (E, \sigma)$. Likewise, an abstract state $M^\sharp \in \mathbb{S}^\sharp$ also includes an abstract environment $E^\sharp \in \mathbb{E}^\sharp = \mathbb{X} \mapsto \mathbb{V}^\sharp$, and the concretization simply asserts the compatibility of the concrete environment with the abstract environment up to the valuation. We write $\mathbb{S}$ for $\mathbb{E} \times \mathbb{M}$ and $\mathbb{S}^\sharp$ for $\mathbb{E}^\sharp \times \mathbb{D}^\sharp_{\mathrm{G}} \times \mathbb{D}^\sharp_{\mathrm{V}}$. The concretization writes down as follows:

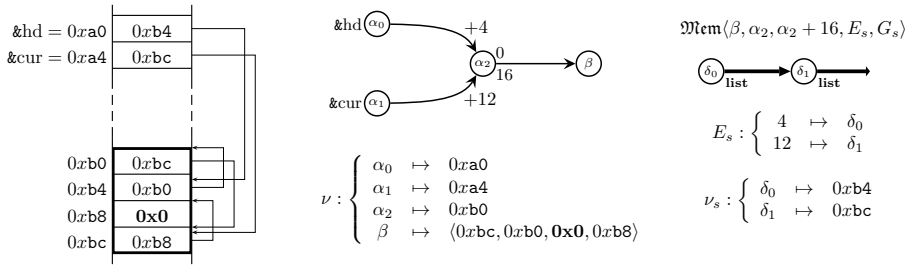
$$(E, \sigma) \in \gamma_{\mathbb{S}}(E^\sharp, G, V) \iff \exists \nu \in \gamma_{\mathrm{V}}(V),\ E = \nu \circ E^\sharp \ \wedge\ (\sigma, \nu) \in \gamma_{\mathrm{G}}(G).$$

3.3 Hierarchical Abstraction

At this stage, we are ready to formalize the final step of our hierarchical abstraction: indeed, we noticed in Sect. 3.2 that symbolic variables denote values (as sequences of bytes), that can be constrained both in the shape graph and in some *underlying* value abstraction; thus, we simply need to let our shape abstraction be a possible instance of the value abstraction.

In order to ensure correct mapping with the main memory, the sub-memory abstraction should carry not only a shape graph, but also a *local environment* describing how sub-memory cells are accessed. Therefore, the general form of a *sub-memory value abstract domain* predicate is:

$$\mathfrak{Mem}\langle \beta, \alpha + o_0, \alpha + o_1, E_s, G_s \rangle$$



(a) Concrete store. (b) Main memory abstraction. (c) Sub-memory.

Fig. 4: Hierarchical abstraction

where:

- $\beta \in \mathbb{V}^\sharp$ denotes the sub-memory contents;
- $[\alpha + o_0, \alpha + o_1[$ denotes the range of addresses covered by the sub-memory (where $\alpha \in \mathbb{V}^\sharp$ is the base address of the block the sub-memory belongs to);
- $G_s \in \mathbb{D}^\sharp_{\mathrm{G}}$ is a shape graph describing the sub-memory;
- $E_s : \mathbb{V}^\sharp_{\mathcal{L}} \to \mathbb{V}^\sharp$ is a partial map from symbolic offsets (linear combination of symbolic variables) relative to α into nodes of sub-shape graph G_s.

In practice, a store may contain several sub-memories, thus an abstract value of $\mathbb{D}^\sharp_{\mathrm{V}}$ consists of a finite set of sub-memory predicates together with a regular abstract element of some other numerical domain to express arithmetic constraints among symbolic variables. We write $\mathbb{D}^\sharp_{V[\mathbf{sub}]}$ (resp., $\gamma_{V[\mathbf{sub}]}$) for the sub-memory abstract domain (resp., concretization). Concretization function $\gamma_{V[\mathbf{sub}]} : \mathbb{D}^\sharp_{V[\mathbf{sub}]} \to \mathcal{P}(\mathbb{Val})$ is formally defined as follows:

$$\begin{aligned} &\nu \in \gamma_{V[\mathbf{sub}]}(\mathfrak{Mem}\langle \beta, \alpha + o_0, \alpha + o_1, E_s, G_s \rangle) \\ &\iff \quad \exists (\sigma_s, \nu_s) \in \gamma_{\mathrm{G}}(G_s), \begin{cases} \mathbf{dom}(\sigma_s) = [\nu_{\mathcal{L}}(\alpha + o_0), \nu_{\mathcal{L}}(\alpha + o_1)[\\ \nu(\beta) = \mathbf{read}(\sigma_s, \nu_{\mathcal{L}}(\alpha + o_0), \nu_{\mathcal{L}}(\alpha + o_1)) \\ \forall l \in \mathbb{V}^\sharp_{\mathcal{L}},\ E_s(l) = \delta \Longrightarrow \nu_{\mathcal{L}}(\alpha + l) = \nu_s(\delta) \end{cases} \end{aligned}$$

As an example, we consider in Fig. 4 the case of an array, used as a sub-memory which contains a list occupying the whole array (for the sake of simplicity, we assume list elements only have a next field). Fig 4a shows a concrete state, where the array has length 4, and all cells are part of a list. All physical addresses are shown and thin edges help visualize pointers. Fig. 4b shows the shape graph which abstracts the main memory together with the valuation used to concretize it into the store of Fig. 4a. Note that symbolic variable β is mapped into the concatenation of four 4-bytes values (we assume a 32 bits architecture), hence a value of length 16 bytes. The associated sub-memory predicate is displayed in Fig. 4c, with its shape graph, its environment and the valuation ν_s used to concretize it appropriately. As G_s summarizes the list into a segment predicate

and an inductive predicate, some physical addresses ($0x$b0 and $0x$b8) do not even appear in ν_s.

In practice, the abstract states manipulated in order to analyze programs such as the code shown in Fig. 1c are more complex, yet the principle is the same as in the example of Fig. 4:

- list elements have additional fields, so that the size of one structure element is 8 bytes or more, and the strides of the pointers in the free-pool region are multiple of that size s;
- the overall size of the free-pool may be much larger, and could actually be kept abstract (i.e., the analysis would only know it is an unsigned number, that would be a multiple of s);
- the sub-memory may not occupy the whole free-pool space (as is the case in the concrete store shown in Fig. 1b), so the free-pool corresponds to a segmentation with several outgoing edges;
- the offsets in the main shape graph and in the sub-environment are non constant linear expressions over symbolic variables.

4 Static Analysis Algorithms in the Hierarchical Abstract Domain

We now describe the static analysis algorithms, which allow to infer precise invariants over both the main memory and the sub-memory for programs such as the insertion routine in Fig. 1c.

4.1 Structure of the analysis

For the most part, the analysis consists of a standard shape analysis following the principles of [6, 5], which can be formalized as a forward abstract interpretation [7]. The concrete semantics $[\![P]\!]$ of program P collects the set of states (ℓ, M) which are reachable from the entry point of P, after any sequence of execution steps: $(\ell, M) \in [\![P]\!]$ if and only if $(\ell_0, M_0) \to (\ell_1, M_1) \to \ldots \to (\ell_n, M_n) \to (\ell, M)$, where ℓ_0 is the entry point of P and $\to$ denotes the transition relation of P. The analysis computes invariants I_ℓ for all control states ℓ, which consist of finite disjunctions of abstract states. The analysis is sound in the following sense:

Theorem 1 (Soundness). *For all $(\ell, M) \in [\![P]\!]$, there exists $M^\sharp \in I_\ell$ such that $M \in \gamma_{\mathsf{S}}(M^\sharp)$.*

To achieve this, we use sound transfer functions to compute abstract post-conditions and sound abstract join and widening operators to over-approximate the effect of control flow joins. The abstract join operator is especially interesting in the sense that it may introduce or merge existing sub-memory predicates, which is why we consider it first (Sect. 4.2). Furthermore, another novelty is the need for all abstraction layers (main memory, sub-memory and other value abstract domains) to exchange information, as one analysis step typically requires some steps of computation be done in all layers (Sect. 4.3).

4.2 Abstract join and management of sub-memory predicates

In the beginning of the analysis, the contents of the memory is unknown, so no information is available in $\mathbb{D}^\sharp_{V[\mathbf{sub}]}$ (the empty set of sub-memory predicates denotes the absence of sub-memory information). As the analysis progresses, sub-memory predicates may be introduced or be combined into new sub-memory predicates. Those operations are performed at control flow join points, by the shape abstract join (which serves both as an abstract union and as a widening).

The abstract join operator takes two inputs $M^\sharp_l = (E^\sharp_l, G_l, V_l)$, $M^\sharp_r = (E^\sharp_r, G_r, V_r)$, and computes an over-approximation $M^\sharp_o = (E^\sharp_o, G_o, V_o)$. To achieve such a result, the shape graph join computes matching partitions of the edges of G_l and G_r and approximates such corresponding sets of edges with edges into G_o. These partitions are described by functions $\Psi_l, \Psi_r : \mathbb{V}^\sharp \to \mathbb{V}^\sharp$, where Ψ_l (resp., Ψ_r) maps nodes of G_o into the nodes of G_l (resp., G_r) that they over-approximate. In the following, we let $\Psi(G)$ (resp., $\Psi(o)$) denote the renaming of all symbolic variables in graph G (resp., in offset o) by applying function Ψ. Then, the computation of G_o takes the form of a sequence of rewriting steps over graph tuples:

$$(G_l, G_r, \mathbf{emp}) = (G^0_l, G^0_r, G^0_o) \overset{\sqcup}{\rightsquigarrow} (G^1_l, G^1_r, G^1_o) \overset{\sqcup}{\rightsquigarrow} \dots$$
$$\dots \overset{\sqcup}{\rightsquigarrow} (G^{k-1}_l, G^{k-1}_r, G^{k-1}_o) \overset{\sqcup}{\rightsquigarrow} (G^k_l, G^k_r, G^k_o) = (\mathbf{emp}, \mathbf{emp}, G_o)$$

where each step i is sound in the sense that

$$\forall s \in \{l, r\},\ \gamma_{\mathbb{S}}(E^\sharp_s, G^i_s * \Psi_s(G^i_o), V_s) \subseteq \gamma_{\mathbb{S}}(E^\sharp_s, G^{i+1}_s * \Psi_s(G^{i+1}_o), V_s)$$

Each step corresponds to a rule, as defined in [5], such as, for instance:

- Rule **(r-pt)** over-approximates a pair of points-to edges with a new one:

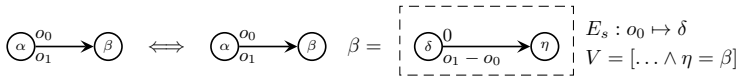
$$\begin{pmatrix} \Psi_l(\alpha)_{[\Psi_l(o), \Psi_l(o')[} \mapsto \Psi_l(\beta), \\ \Psi_r(\alpha)_{[\Psi_r(o), \Psi_r(o')[} \mapsto \Psi_r(\beta), \\ \mathbf{emp} \end{pmatrix} \overset{\sqcup}{\rightsquigarrow} \begin{pmatrix} \mathbf{emp}, \\ \mathbf{emp}, \\ \alpha_{[o, o'[} \mapsto \beta \end{pmatrix}.$$

- Rule **(r-emp-seg)** matches an empty region in G_l with a region of G_r that can be proved a particular case of a segment; one case of **(r-emp-seg)** is:

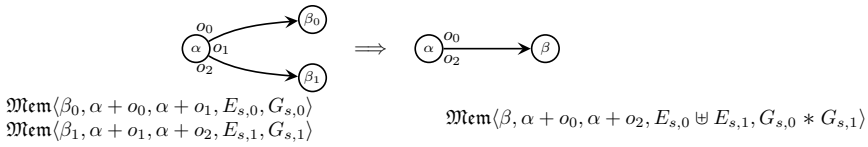
$$\begin{pmatrix} \mathbf{emp}, \\ \Psi_r(\alpha)_{[\Psi_r(o), \Psi_r(o')[} \mapsto \Psi_r(\beta), \\ \mathbf{emp} \end{pmatrix} \overset{\sqcup}{\rightsquigarrow} \begin{pmatrix} \mathbf{emp}, \\ \mathbf{emp}, \\ \alpha \cdot \mathbf{list} \ast\!\!\Leftarrow \beta \cdot \mathbf{list} \end{pmatrix} \text{ if } \Psi_l(\alpha) = \Psi_l(\beta).$$

All rules are shown in [5]. The soundness of each step guarantees the soundness of the result, given value the abstract element $V_o = (\Psi_l)^{-1}(V_l) \nabla_V (\Psi_r)^{-1}(V_r)$, where ∇_V is a widening operator in $\mathbb{D}^\sharp_{\mathrm{V}}$ (symbolic variables of inputs need be renamed using Ψ_l, Ψ_r to make value abstractions consistent), and environment $E^\sharp_o = (\Psi_l)^{-1} \circ E^\sharp_l$. In our setup, with points-to edges of non statically known size (Sect. 3.1) and with sub-memory predicates (Sect. 3.3), additional join rewriting rules need be considered, resulting in the *introduction* and in the *fusion* of sub-memory predicates, as part of ∇_V.

Theorem 2 (Soundness). *For all* $s \in \{l, r\}$, $\gamma_{\mathbb{S}}(E^\sharp_s, G_s, V_s) \subseteq \gamma_{\mathbb{S}}(E^\sharp_o, G_o, V_o)$.



(a) Introduction of a sub-memory predicate.



(b) Fusion of sub-memory predicates.

Fig. 5: Management of sub-memory predicates

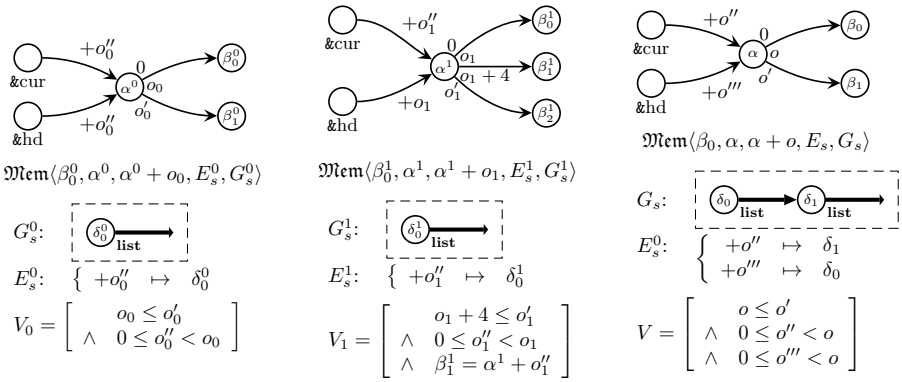
Join over contiguous points-to edges. Unlike the abstract domain of [5], the abstraction shown in Sect. 3 copes with arrays, thus new rewriting rules need be added for the case where matching nodes $\Psi_l(\alpha), \Psi_r(\alpha)$ are the origin of different numbers of points-to edges in both arguments. Thus, the extended algorithm adds a general rule **(r-fusion)** to re-partition such segmentations from a same node, as in array analyses such as [8]. When the left segmentation has one edge and the right segmentation has two, **(r-fusion)** writes down as follows:

$$\begin{pmatrix} \Psi_l(\alpha)_{[\Psi_l(o),\Psi_l(o'')[} \mapsto \Psi_l(\beta), \\ \Psi_r(\alpha)_{[\Psi_r(o),\Psi_r(o')[} \mapsto \beta_0 * \Psi_r(\alpha)_{[\Psi_r(o'),\Psi_r(o'')[} \mapsto \beta_1, \\ \mathbf{emp} \end{pmatrix} \overset{\sqcup}{\rightsquigarrow} \begin{pmatrix} \mathbf{emp}, \\ \mathbf{emp}, \\ \alpha_{[o,o''[} \mapsto \beta \end{pmatrix}$$

where Ψ_r maps β into the *sequence* $\langle \beta_0, \beta_1 \rangle$, i.e. expresses that symbolic variable β should over-approximate values corresponding to the concatenation of the values represented by β_0 and β_1 in G_r. General **(r-fusion)** subsumes **(r-pt)**.

Introduction and fusion of a sub-memory predicate. Whenever a shape graph contains a points-to edge, a sub-memory predicate can be introduced, as shown in Fig. 5a. When applying rule **(r-fusion)**, such sub-memory predicates can be introduced in both join inputs, so as to capture the meaning of points-to edges in both inputs. However, this process generates two sub-memory predicates in the right hand side (and one in the left hand side), thus those sub-memory predicates need be combined together. This operation can be performed as shown in Fig. 5b.

Example of an abstract join. Fig. 6 shows a join similar to those found in the analysis of the code of Fig. 1c. For the sake of clarity, we show only a relevant fragment of the abstract states, and we express the relation between β_1^1 and α^1 in the value abstraction (in the analysis, it is actually represented as a looping points-to edge $\alpha^1_{[o'_1,o'_1+4[} \mapsto \alpha^1 + o''_1$). The abstract state shown in Fig. 6a describes a list stored in a sub-memory and pointed to by cur. Fig. 6b describes the situation after allocating an additional element in the free-pool, pointed to by cur. Join (Fig. 6c) applies rule **(r-fusion)** to the first two edges, introduces



(a) Left input (G_0, V_0). (b) Right input (G_1, V_1). (c) Output (G_o, V_o).

Fig. 6: Abstract join at the second iteration

a sub-memory in G_1, merges that sub-memory with the pre-existing one, and then performs an abstract join in the sub-memory. This sub-memory join then introduces a segment, thanks to rule **(r-emp-seg)**.

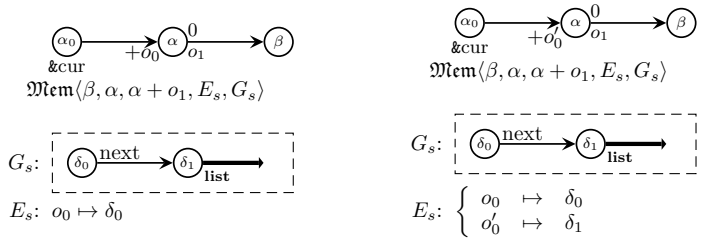
Soundness of the abstract join and termination of the widening can be proved as in [5]. The implementation is also similar (in particular, the algorithm actually infers partitions Ψ_l, Ψ_r as part of the sequence of rewriting steps leading to G_o).

4.3 Abstract transfer functions

Abstract transfer functions compute over-approximated abstract post-conditions for each elementary concrete operations. When analyzing a statement (such as an allocation, an assignment, a test...) between control states ℓ and ℓ', the analyzer should evaluate a transfer function $\mathfrak{transfer}^\sharp_{\ell,\ell'}$. This transfer function should satisfy the soundness condition below:

$$M \in \gamma_{\mathbb{S}}(M^\sharp) \wedge (\ell, M) \to (\ell', M') \Longrightarrow M' \in \gamma_{\mathbb{S}}(\mathfrak{transfer}^\sharp_{\ell,\ell'}(M^\sharp))$$

Analysis of an assignment: In the following, we consider the analysis of the assignment `lv := ex` between ℓ and ℓ', where `lv` is an l-value and `ex` an expression (the other transfer functions are similar, thus we formalize only $\mathfrak{assign}$). In the concrete level, $[\![\texttt{lv}]\!]$ (resp., $[\![\texttt{ex}]\!]$) denotes the semantics of `lv` (resp., `ex`); it maps a memory state into an address (resp., a numeric or pointer value). Then, the concrete transitions corresponding to that assignment are of the form $(E, \sigma) \to (E, \sigma')$, where $\sigma' = \sigma[[\![\texttt{lv}]\!](E, \sigma) \leftarrow [\![\texttt{ex}]\!](E, \sigma)]$. In the abstract level, $[\![\texttt{lv}]\!]^\sharp$ (resp., $[\![\texttt{ex}]\!]^\sharp$) returns a node with offset $\alpha + o$ denoting the address of the cell to modify (resp., a node with offset $\beta' + o'$ denoting the value to assign). When the abstract pre-condition shape graph G contains a points-to edge $\alpha + o \mapsto \beta$, $\mathfrak{assign}$ should simply replace this edge with points-to edge $\alpha + o \mapsto \beta' + o'$. However, some transformations may need be done on G before this trivial $\mathfrak{assign}$ can be applied:



(a) Pre-condition. (b) Post-condition.

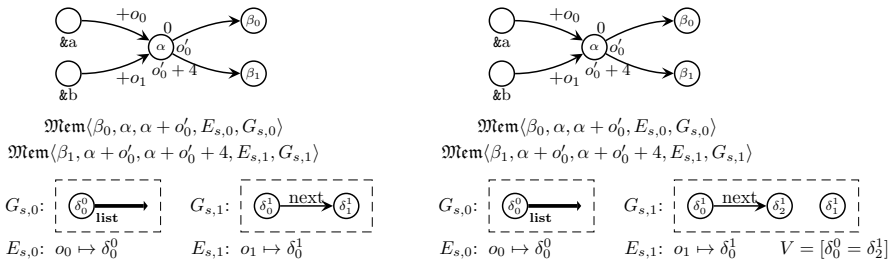
Fig. 7: Analysis of the assignment cur = cur−>next

- When the evaluation of either `lv` or `ex` requires accessing fields which are not materialized, as they are summarized as part of inductive or segment edges, those should be unfolded first [5].
- When `ex` is a non trivial numeric expression, it should be analyzed in the value domain. Let us consider the case where `ex` is $4 * x + 8$. That expression should be transformed by replacing all l-values with nodes corresponding to their addresses, which gives an expression of the form $4 \cdot \delta + 8$. A fresh node β' should be added to G. The assignment $\beta' \leftarrow 4 \cdot \delta + 8$ should be analyzed in the value domain, using sound abstract transfer function $\mathfrak{assign}_{\mathbb{V}^\sharp}$. Then, the $\mathfrak{assign}$ proceeds as above, by switching a points-to edge.

In the following, we extend this operator to the hierarchical abstract domain. We assume abstract pre-condition $(E^\sharp, G, V)$ contains at least one sub-memory $\mathfrak{Mem}\langle\beta, \alpha + o_0, \alpha + o_1, E_s, G_s\rangle$, and describe the analysis assignment `lv := ex`. As cases where the assignment only affects the main memory are unchanged, we consider only the cases where it involves a read or a write into the sub-memory, and show what changes need be done to the classic $\mathfrak{assign}$ operator.

Read in a sub-memory. Let us consider statement cur = cur−>next, with the abstract pre-condition shown in Fig. 7a. Then, l-value cur evaluates into α_0, which is the origin of points-to edge $\alpha_0 \mapsto \alpha + o_0$. The evaluation of r-value cur−>next is more complex, as cur points into the sub-memory, at offset o_0. However, the environment maps o_0 into sub-memory node δ_0, which has a next field, pointing to δ_1. Thus, in the sub-memory, the r-value evaluates to δ_1. The effect of the assignment is thus captured by an update to the main memory edge destination offset (o_0' instead of o_0) and an update to the sub-environment to reflect this new mapping (Fig. 7b). To summarize, sub-memory pointer reads can be handled like normal pointer reads, where the base address is represented by the sub-memory based node.

When the read operation returns a node that does not appear in the sub-memory environment (and thus, cannot be seen from the outside), an equality constraint between that node and an external fresh node should be generated in $\mathbb{V}^\sharp$, so as to capture the effect of the assignment.



(a) Pre-condition. (b) Post-condition.

Fig. 8: Analysis of the assignment b−>next = a

Write in a sub-memory. We now consider the case of an assignment to a structure field *inside* a sub-memory. In Fig. 8, we show an abstract pre-condition and an abstract post-condition computed from it, in the case of assignment b−>next = a where a and b are two pointers into a free-pool. This abstract state arises after introduction of two sub-memories, and before their fusion by a join operator (Sect. 4.2). The effect of the assignment is local to the sub-memory where b points to:

- a new node δ^1_2 is created with the equality constraint that it is equal to δ^0_0;
- sub-memory points-to edge $\delta^1_0 \cdot \text{next} \mapsto \delta^1_1$ is replaced with $\delta^1_0 \cdot \text{next} \mapsto \delta^1_2$.

The second operation is actually performed as part of the evaluation of $\mathfrak{assign}_{\mathbb{V}^\sharp}$ over the sub-memory which content is bound to β_1. Past this step, the invariant attached to that sub-memory includes pointers *leaving* the sub-memory itself (to the other sub-memory). Similar situations arise in the program of Fig. 1c, when updates are done inside the free-pool. Moreover, when the sub-memory cell that need be assigned is part of a folded inductive predicate, classical inductive predicate unfolding techniques [5] apply.

Soundness. Operator $\mathfrak{assign}$ is sound:

Theorem 3 (Soundness). *Let* $M = (E, \sigma) \in \gamma_{\mathbb{S}}(M^\sharp)$*. If we let* $a = [\![\mathtt{lv}]\!](M)$ *and* $v = [\![\mathtt{ex}]\!](M)$*, then* $(E, \sigma[a \leftarrow v]) \in \gamma_{\mathbb{S}}(\mathfrak{assign}(\mathtt{lv}, \mathtt{ex}, M^\sharp))$*.*

5 Prototype and Implementation

We integrated the hierarchical abstraction into the MemCAD analyzer (*Memory Compositional Abstract Domain*, http://www.di.ens.fr/~rival/memcad.html). It was implemented as a functor, which lifts a shape abstract domain into a value abstract domain, which can in turn be fed into the shape abstract domain functor. Numerical abstract invariants ($\mathbb{D}^\sharp_{\mathrm{V}}$) are represented in a numerical domain complying with the Apron [16] interface. The experiments below use convex polyedra [9]. The results obtained when running it on a series of routines that build and manipulate a sub-structure in a free-pool are shown in the table below

(in the third column), and are compared with times to analyze similar routines using regular memory allocation system call **malloc** (second column), which do not require the hierarchical abstraction. Run-times are given in seconds, as observed with an Intel Core i3 CPU laptop running at 2.10 GHz, with 4 Gb of RAM. The analysis is fully automatic and inputs only a generic list inductive definition (Sect. 2) and the unannotated source code. The set of codes considered in the table below includes `running`, our main example (Fig. 1c), as well as other basic operations on the dynamic structure (head and tail insertion, flipping of cells, drop of a cell). Those comprise all typical features of a the user defined allocator based on a static free-pool as found in `running`. While the industrial code of [21] never "deallocates" cells (instead, it sometimes reset the free-pool, before building a new structure in it), we included a `drop` example, where a cell is selected and removed from the linked list, yet cannot be reused; such cells are abstracted from the sub-memory contents (those cells are abstracted into a $\ldots * \mathbf{true}$ predicate [23] i.e., a heap region nothing is known about).

Program	Allocation method		Description
	malloc	free-pool array	
`running`	0.195	0.520	The running example
`head`	0.019	0.034	List, head-insertion
`tail`	0.027	0.050	List, tail-insertion
`traversal`	0.056	0.107	List, tail-insertion then traversal
`flip`	0.139	0.323	List, flipping two cells after selection
`drop`	0.104	0.289	List, dropping a cell after selection
`integers`	NA	0.016	Initialization of an array to zeros

In all those examples, MemCAD infers a precise abstract description of all dynamic structures in the free-pool or in the main memory, and proves memory safety. We observe a 2X to 3X slowdown in the analyses of codes using a free-pool. The difference is justified by the extra burden of maintaining the sub-memory predicates together with the array segmentation and side numerical predicates over offsets. While noticeable, this slowdown is very reasonable, as the properties which are inferred are strong and memory safety is proved. Last, we remarked that the current implementation of the MemCAD analyzer does not feature a very efficient management of symbolic disjunctions of abstract states; addressing that separate issue would improve timings significantly.

6 Related Works and Conclusion

Our hierarchical abstract domain allows to design memory abstractions in a modular way, which relates to the layout of data-structures used in programs. This modularity makes the analysis design simpler while preserving its characteristics (precision and performance). Our abstract domain is parametric in the data of an underlying value abstraction (so that more complex abstract domains could be used in order to deal with values), and in the data of a set of structure inductive definitions (our test case uses only lists, but the analysis would be very

similar if doubly-linked lists or trees were built inside the free-pool instead of singly linked lists). Furthermore, our proposal integrates much of the power of array analyses such as [13, 15, 8] into a shape analysis framework [6, 5]. This was made possible by the structure of the abstraction proposed in [5], which allows a nice combination with a value abstraction. While that value abstraction was initially set to be a numerical abstraction, our hierarchical abstract domain shows that much more complex structures can be devolved to an underlying domain. This allows a very modular design for the static analyses. We notice that the notion of array partition of [8] plays a similar role as the partition used in abstract join [5] of shape graphs. Like [8], our analysis does not require a pre-analysis to discover array partitions, following the principles of the shape join of [5]. Composite structures are a common issue in the shape analysis field [19, 11, 10]. An important contribution of our proposal is to decompose the abstract domain into smaller domains, which are easier to implement and to reason about.

Gulwani et al. [14] enhance shape abstract domains with numerical information so as to reason about the size of arrays stored in linked list elements. Their analysis does not allow to reason about the structure contents whereas our approach allows to delegate such a description to a generic value domain (which may store shape information, when the arrays store complex structures). In [4], Calcagno et al. address the safety of general memory allocators, like the C **malloc**, using an ad-hoc abstract domain based on separation logic, which embeds both shape and numerical information. Their work addresses a separate set of cases than ours, as their approach could not deal with our user-defined pseudo-allocator whereas MemCAD does not handle their examples at this point. As this analysis considers the allocator separately from the code, it can abstract away the contents of memory block. The analysis of [18] targets overlaid dynamic structures, which is also a completely separate issue than that of our application specific memory allocator, and relies on very different techniques.

The most important future work is the integration of our shape abstraction into an analyzer such as Astrée [3], which would effectively improve the analysis of embedded applications such as those considered in [22, 21]. This represents a considerable amount of work as no standard interface has been set up so far for memory abstractions, unlike numerical abstractions. We believe our work actually achieves a step in that direction, as the design of the hierarchical abstraction imposed a careful assessment of abstract domain component interfaces. Other future works include the support of non-contiguous segmentations, which would allow the analysis of a wide family of memory allocators. Last, our framework supports the composition of more than two levels of hierarchical abstractions, e.g., to analyze lists of elements containing arrays, that are stored inside large static zones, thus we could consider such examples.

Acknowledgments. We thank the members of the MemCAD group for discussions and the reviewers for suggestions that helped improving this paper.

References

1. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.
2. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace (I@A 2010)*, 2010.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, 2003.
4. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, 2006.
5. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.
6. E. Chang, X. Rival, and G. Necula. Shape analysis with structural invariant checkers. In *SAS*, 2007.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
8. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, 2011.
9. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.
10. I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *POPL*, 2011.
11. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
12. DO-178C: Software considerations in airborne systems and equipment certification. Technical report, Radio Technical Commission on Aviation, 2011.
13. D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *POPL*, 2005.
14. S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *POPL*, 2009.
15. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, 2008.
16. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, 2009.
17. V. Laviron, E. Chang, and X. Rival. Separating shape graphs. In *ESOP*, 2010.
18. O. Lee, H. Yang, and R. Petersen. Program analysis for overlaid data structures. In *CAV*, 2011.
19. M. Marron, D. Stefanovic, M. V. Hermenegildo, and D. Kapur. Heap analysis in the presence of collection libraries. In *PASTE*, 2007.
20. A. Miné. The octagon abstract domain. *HOSC*, 19(1), 2006.
21. A. Miné. Static analysis of run-time errors in embedded critical parallel c programs. In *ESOP*, 2011.
22. D. Monniaux. Verification of device drivers and intelligent controllers: a case study. In *EMSOFT*, 2007.
23. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
24. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.