Hierarchical Shape Abstraction for Analysis of Free List Memory Allocators

Bin Fang[2,1] and Mihaela Sighireanu[1]

[1] IRIF, University Paris Diderot and CNRS, France
[2] Shanghai Key Laboratory of Trustworthy Computing, ECNU, China

Abstract. We propose a hierarchical abstract domain for the analysis of free list memory allocators that tracks shape and numerical properties about both the heap and the free lists. Our domain is based on Separation Logic extended with predicates that capture the pointer arithmetics constraints for the heap list and the shape of the free list. These predicates are combined using a hierarchical composition operator to specify the overlapping of the heap list by the free list. In addition to expressiveness, this operator leads to a compositional and compact representation of abstract values and simplifies the implementation of the abstract domain. The shape constraints are combined with numerical constraints over integer arrays to track properties about the allocation policies (best-fit, first-fit, etc). Such properties are out of the scope of the existing analyzers. We implemented this domain and we show its effectiveness on several implementations of free list allocators.

1 Introduction

A dynamic memory allocator (DMA) is a piece of software managing a reserved region of the heap. It appears in general purpose libraries (e.g., C standard library) or as part of applications where the dynamic allocation shall be controlled to avoid failure due to memory exhaustion (e.g., embedded critical software). A client program interacts with the DMA by requesting blocks of memory of variable size that it may free at any time. To offer this service, the DMA manages the reserved memory region by partitioning it into arbitrary sized blocks of memory, also called *chunks*. When a chunk is allocated to a client program, the DMA can not relocate it to compact the memory region (like in garbage collectors) and it is unaware about the kind (type or value) of data stored. The set of chunks not in use, also called *free chunks*, is managed using different techniques. In this paper, we focus on *free list allocators* [20,27], that records free chunk in a list. This class of DMA includes textbook examples [20,18] and real-world allocators [21].

The automated analysis of DMA faces several challenges. Although the code of DMA is not long (between one hundred to a thousand LOC), it is highly optimised to provide good performance. Low-level code (e.g., pointer arithmetics, bit fields, calls to system routines like `sbrk`) is used to manage efficiently (i.e., with low additional cost in memory and time) the operations on the chunks in the reserved memory region. At the same time, the free list is manipulated using high level operations over typed memory blocks (values of C structures) by mutating pointer fields without pointer arithmetic. The analyser has to deal efficiently with this *polar usage of the heap* made by the DMA. The invariants maintained by the DMA are complex. The memory region is organised into a *heap list* based on the size information stored in the chunk header such that chunk overlapping and memory leaks are avoided. The start addresses of chunks shall be aligned to some given constant. The free list may have complex shapes (cyclic, acyclic, doubly-linked) and may be sorted by the start address of chunks to ease free chunks coalescing. A precise analysis shall keep track of both numerical and shape properties to infer specifications implying such invariants for the allocation and deallocation methods of the DMA.

These challenges have been addressed partially by several works in the last ten years [24,5,26]. In [24], efficient numerical analyses have been designed to track address alignment and bit-fields. The most important progress has been done by the analysis proposed by Calcagno et al [5]. It is able to track the free list shape and the numerical properties of chunk start addresses due to an abstract domain built on an extension of Separation Logic (SL) [25] with numerical constraints and predicates specifying memory blocks. However, some properties of the heap and free list can not be tracked, e.g., the absence of memory leaks or the ordering of start addresses of free-chunks. Although the analysis in [26] does not concern DMA, it is the first to propose a hierarchical abstraction of the memory to track properties of linked data structures stored in static memory regions. However, this analysis can not track properties like address sorting of the high level data structures (here the free list) stored in the memory region. Furthermore, its link with a logic theory is not clear. Thus, a precise, logic based analysis for the inference of properties of free list DMA is still a challenge.

In this paper, we propose a static analysis that is able to infer the above complex invariants of DMA on both heap list and free list. We define an abstract domain which uses logic formulas to abstract DMA configurations. The logic proposed extends the fragment of symbolic heaps of SL with a hierarchical composition operator, $\Supset$, to specify that the free list covers partially the heap list. This operator provides a hierarchical abstraction of the memory region under the DMA control: the low-level memory manipulations are specified at the level of the heap list and propagated in a way controlled by the abstraction at the level of the free list. The shape specification is combined with a fragment of first order logic on arrays to capture properties of chunks in lists, similar to [3]. This combination is done in an accurate way as regards the logic by including sequences of chunk addresses in the inductive definitions of list segments. The main advantages and contributions of this work are (1) the *high precision of the abstraction* which is able to capture complex properties of free list DMA implementations, (2) the *strong logical basis* allowing to infer invariants that may be used by other verification methods, and (3) the *modularity* of the abstract domain permitting to reuse existing abstract domains for the analysis of linked lists with integer data.

2 Overview

Figure 1 includes excerpts from our running example, a free list DMA implementation proposed in [1]. The type `HDR` (Figure 1 (a)) defines the information stored by the DMA at the start of chunks. The field `size` stores the full size of the chunk (in blocks of `sizeof(HDR)` bytes) and it is used by the heap list to determine the start of the next chunk. The field `fnx` is valid only for free chunks (i.e., chunks in the free list) and it stores the start address of the next free chunk. To simplify the presentation, we added the ghost field `isfree`, to mark explicitly free chunks. The memory region managed by the DMA is enclosed within the addresses stored by the global variables `_hsta` and `_hend`; they are initialised by `minit` using `sbrk` calls. The start of the free list is stored in `frhd`. An intuitive view of the concrete state of the DMA is shown in Figure 1(d). The busy chunks are represented in grey. The "next chunk" relation in the heap list (defined using the field `size`) is represented by the lower arrows; the upper arrows represent the "next free chunk" relation defined by the `fnx` field. Furthermore, other structural invariants should be maintained after each call of DMA methods: the heap list shall be well formed inside the memory region [`_hsta`,`_hend`), consecutive chunks of the heap list are not both free (*early coalescing* policy), the free list shall include only chunks of the heap list, be acyclic and sorted by the start address of chunks. The allocation method searches a chunk with size bigger than the requested `nbytes`; if the chunk is larger, it is split in two parts such that the last part (the end of the initial chunk) is allocated.

The goal of our analysis is to establish that, if the client uses correctly the DMA methods, these methods (i) preserve the above structural invariants and (ii) are memory safe. In particular, we analyse the DMA methods starting from a client program which initialises the DMA and then calls allocation and deallocation methods (see Section 5) in a correct way.

Heap list abstraction. The concrete memory configurations managed by the DMA are abstracted first using an extension of the *symbolic heap graphs* fragment [9] of SL. The logic fragment is parameterised by a set of predicates which capture the properties of the heap list as follows:

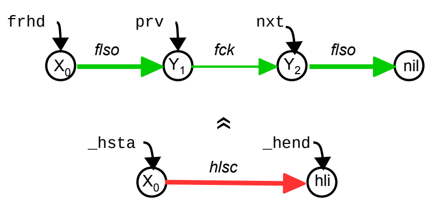
- The predicate $\mathsf{blk}(X;Y)$, introduced in [5], specifies an untyped sequence of bytes between the symbolic addresses X and Y. E.g., the configuration obtained at line 20 of `minit` is abstracted by $\mathsf{blk}($`_hsta`;`_hend`$)$.
- The predicate $\mathsf{chd}(X;Y)$ specifies a memory block $\mathsf{blk}(X;Y)$ storing a value of type `HDR`; the fields of this value are represented by the symbolic variables X`.size`, X`.fnx`, and X`.isfree` respectively.
- The predicate $\mathsf{chk}(X;Y)$ specifies a chunk built from a chunk header $\mathsf{chd}(X;Z)$ followed by a block $\mathsf{blk}(Z;Y)$ such that the full memory occupied, i.e., $Y-X$, has size given by X`.size`$\times$`sizeof(HDR)`.
- A well formed heap list segment starting at address X and ending before Y is specified using the predicate $\mathsf{hls}(X;Y)[W]$. The inductive definition of this predicate (see Table 2) requires that chunks do not overlap or leave memory leaks. The variable W registers the sequence of start addresses of chunks in

```
1  typedef struct hdr_s {
2    struct hdr_s *fnx;
3    size_t size;
4    //@ghost bool isfree;
5  }  HDR;
6
7  static void *_hsta = NULL;
8  static void *_hend = NULL;
9  static HDR *frhd = NULL;
10 static size_t memleft;
11
12 void minit(size_t sz)
13 {
14   size_t align_sz;
15   align_sz = (sz+sizeof(HDR)-1)
16            & ~(sizeof(HDR)-1);
17
18   _hsta = sbrk(align_sz);
19   _hend = sbrk(0);
20
21   frhd = _hsta;
22   frhd->size = align_sz / sizeof(HDR);
23   frhd->fnx = NULL;
24   //@ghost frhd->isfree = true;
25
26   memleft = frhd->size;
27 }
```

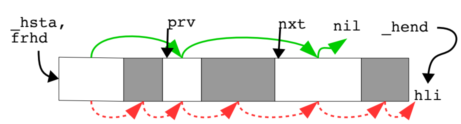
(a) Globals and initialisation

```
28 void* malloc(size_t nbytes)
29 {
30   HDR *nxt, *prv;
31   size_t nunits =
32     (nbytes+sizeof(HDR)-1)/sizeof(HDR) + 1;
33
34   for (prv = NULL, nxt = frhd; nxt;
35        prv = nxt, nxt = nxt->fnx) {
36     if (nxt->size >= nunits) {
37       if (nxt->size > nunits) {
38         nxt->size -= nunits;
39         nxt += nxt->size;
40         nxt->size = nunits;
41       } else {
42         if (prv == NULL)
43           frhd = nxt->fnx;
44         else
45           prv->fnx = nxt->fnx;
46       }
47       memleft -= nunits;
48       //@ghost nxt->isfree = false;
49       return ((void*)(nxt + 1));
50     }
51   }
52   warning("Allocation Failed!");
53   return (NULL);
54 }
```

(b) Allocation



(c) Part of the abstract invariant at line 34



(d) Concrete memory

Fig. 1. Running example with code, concrete memory, and abstract specification

the list segment and it is used to put additional constraints on the fields of these chunks. For DMA with early coalescing of free-chunks (i.e., coalescing at `free`), we abstract the heap list segments by a stronger predicate, hlsc.

These predicates are combined using the *separation conjunction operator* $*$ of SL, which requires disjointness of memory regions specified by its operands. The bottom of Figure 1(c) illustrates the heap list abstraction of the concrete memory provided in Figure 1(d); for readability, the abstraction is represented by its Gaifman graph. The ghost variable hli represents the end of the data segment of the DMA, as returned by `sbrk(0)`.

Hierarchical abstraction of the free list. The first abstraction layer captures the total order of chunks in the heap list. The free list defines a total order over the set of free chunks. The second abstraction layer captures this order using the same SL fragment but over a different set of predicates (see Table 2):

- The predicate $\mathsf{fck}(X;Y)$ specifies a chunk $\mathsf{chk}(X;\ldots)$ starting at X, with X`.fnx` bound to Y and X`.isfree` set to true.
- The predicate $\mathsf{fls}(X;Y)[W]$ specifies a free list segment starting at X, whose last element field `fnx` points to Y; W registers the sequence of start addresses of free chunks in the list segment. The predicate $\mathsf{flso}(X,\ldots)[W]$ abstracts free list segments sorted by the start address of chunks.

The top of Figure 1(c) illustrates the free list abstraction by its Gaifman graph.

Finally, the memory shape abstraction is obtained by composing the two abstraction levels using a new operator, denoted by $\Supset$, which requires that the set of chunks in the free list abstraction is exactly the sub-set of chunks in the heap list whose field `isfree` has value true. Notice that the operator $\Supset$ can not be replaced by the logical conjunction because we are using the intuitive semantics of SL where spatial formulas fully specify the memory configurations. Or the free list abstraction provides only a partial specification of the heap.

Constraints over sequences of chunk addresses. The predicates presented above specify invariants of DMA independent of parameters of DMA methods. To capture allocation policies that depend on these parameters (e.g., the first-fit policy implemented by the `malloc` in Figure 1(b)), we introduce universal constraints over sequences of chunk start addresses W attached to shape atoms, like in [3]. For example, the first-fit policy obtained at line 37 of `malloc`, is specified by:

$$\begin{aligned}\mathsf{hlsc}(X_0;\mathsf{hli})[W_H] \Supset & (\mathsf{fls}(Y_0;Y_2)[W_1] * \mathsf{fck}(Y_2;Y_3) * \mathsf{fls}(Y_3;\mathsf{nil})[W_2]) \\ & \wedge\ Y_2\texttt{.size} \geq \texttt{nunits}\ \wedge\ \forall X \in W_1 \cdot X\texttt{.size} < \texttt{nunits}\end{aligned} \quad (1)$$

where Y_2 is the symbolic address stored in the program variable `nxt`. The general form of universal constraints is $\forall X \in W \cdot A_G \Rightarrow A_U$, where A_G and A_U are arithmetic constraints over X and its fields. To obtain an efficient analysis, we fix A_G and infer A_U. We require that both A_G and A_U belong to a class of constraints supported by some numerical abstract domain (see Section 3).

Static analysis with hierarchical shape abstraction. Overall, the analysis algorithm is a standard shape analysis algorithm. To expose fields constrained or assigned by the program statements, it unfolds predicate definitions. To limit the size of the abstraction, the algorithm normalises formulas to maintain only symbolic addresses that are cut-points, i.e., they are stored in the program variables or are sharing points in lists. This transformation of formulas folds back sub-formulas into more general predicates. The set of normalised shape formulas is bounded, so we define the widening operator only for the sequence constraints.

The hierarchical shape requires to solve a number of specific issues (see Section 5). The unfolding of shape predicates shall be done at the appropriate level of abstraction. For example, a traversal of the free list requires only unfolding and folding at the free list level. The heap list level may abstract chunks which are explicit in the free list level. Thus, we define protocols for the unfolding and folding operations at each level that are sound as regards the hierarchical composition defined by the operator $\Supset$ and with the sequence constraints.

Table 1. Logic syntax

$X, Y \in \mathsf{AVar}$ location variables	$W \in \mathsf{SVar}$ sequence variables
$i, j \in \mathsf{IVar}$ integer variables	$\# \in \{=, \neq, \leq, \geq\}$ comparison operators
$x \in \mathsf{Var}$ logic variable	$\vec{x}, \vec{y} \in \mathsf{Var}^*$ vectors of variables
$X\texttt{.f}$ field access term	t, Δ integer term resp. formula

φ	$::=$	$\Pi \wedge \Sigma \mid \varphi \vee \varphi \mid \exists x \cdot \varphi$	formulas
Π	$::=$	$A \mid \forall X \in W \cdot A \Rightarrow A \mid W = w \mid \Pi \wedge \Pi$	pure formulas
A	$::=$	$X[\texttt{.fnx}] - Y[\texttt{.fnx}] \; \# \; t \mid \Delta \mid A \wedge A$	location and integer constraints
w	$::=$	$\epsilon \mid \texttt{[}X\texttt{]} \mid W \mid w.w$	sequence terms
Σ	$::=$	$\Sigma_H \Supset \Sigma_F$	spatial formulas
Σ_H	$::=$	$\mathsf{emp} \mid \mathsf{blk}(X;Y) \mid \mathsf{chd}(X;Y) \mid \mathsf{chk}(X;Y) \mid X \mapsto x \mid$ $\mathsf{hls}(X;Y)[W] \mid \mathsf{hlsc}(X,i;Y,j)[W] \mid \Sigma_H * \Sigma_H$	heap formulas
Σ_F	$::=$	$\mathsf{emp} \mid \mathsf{fck}(X;Y) \mid \mathsf{fls}(X;Y)[W] \mid \mathsf{flso}(X,x;Y,y)[W] \mid \Sigma_F * \Sigma_F$	free list formulas

3 Logic Fragment Underlying the Abstract Domain

We formalise in this section a fragment of Separation Logic [25] used to define the values of our abstract domain in Section 4.

Syntax. Let AVar be a set of *location variables* representing heap addresses; to simplify the presentation, we consider that AVar contains a special variable nil representing the null address, also denoted by nil. Let SVar be a set of *sequence variables*, interpreted as sequences of heap addresses and IVar be a set of *integer variables*. The full set of *logic variables* is denoted by $\mathsf{Var} = \mathsf{AVar} \cup \mathsf{SVar} \cup \mathsf{IVar}$. The domain of heap addresses is denoted by $\mathbb{A}$ while the domain of values stored in the heap is generically denoted by $\mathbb{V}$, thus $\mathbb{A} \subseteq \mathbb{V}$. To simplify the presentation, we fix `HDR`, the type of chunk headers, and its fields $\{\texttt{size}, \texttt{fnx}, \texttt{isfree}\}$ typed as declared in Figure 1. The syntax of formulas is given in Table 1.

Formulas are in disjunctive normal form. Each disjunct is built from a pure formula Π and a spatial formula Σ. Pure formulas Π characterise the values of logic variables using comparisons between location variables, e.g., $X - Y = 0$, constraints Δ over integer terms, and sequence constraints. We let constraints in Δ unspecified, though we assume that they belong to decidable theories, e.g., linear arithmetic. The integer terms t are built over integer variables and field accesses using classic arithmetic operations and constants. We denote by $\Pi_\forall$ (resp. Π_W, $\Pi_\exists$) the set of sub-formulas of Π built from universal constraints (resp. sequence constraints, quantifier free arithmetic constraints).

A spatial formula has two components: Σ_H specifies the heap list and the locations outside this region; Σ_F specifies only the free list. The operator $\Supset$ ensures that all locations specified by Σ_F are start addresses of free chunks in

Table 2. Derived predicates

$\mathsf{chd}(X;Y)$	$\triangleq \mathsf{blk}(X;Y) \wedge \mathtt{sizeof(HDR)} = Y - X \wedge X \equiv_{\mathtt{sizeof(HDR)}} 0$
$\mathsf{chk}(X;Y)$	$\triangleq \exists Z \cdot \mathsf{chd}(X;Z) * \mathsf{blk}(Z;Y) \wedge X\mathtt{.size} \times \mathtt{sizeof(HDR)} = Y - X$
$\mathsf{fck}(X;Y)$	$\triangleq \exists Z \cdot \mathsf{chk}(X;Z) \wedge X\mathtt{.isfree} = 1 \wedge X\mathtt{.fnx} = Y$
$\mathsf{hls}(X;Y)[W]$	$\triangleq \mathsf{emp} \wedge X = Y \wedge W = \epsilon$
	$\vee\ \exists Z, W' \cdot \mathsf{chk}(X;Z) * \mathsf{hls}(Z;Y)[W'] \wedge W = \mathtt{[}X\mathtt{]}.W'$
$\mathsf{hlsc}(X, f_p; Y, f_\ell)[W]$	$\triangleq \mathsf{emp} \wedge X = Y \wedge W = \epsilon \wedge 0 \leq f_p + f_\ell \leq 1$
	$\vee\ \exists Z, W', f \cdot \mathsf{chk}(X;Z) * \mathsf{hlsc}(Z, f; Y, f_\ell)[W'] \wedge W = \mathtt{[}X\mathtt{]}.W'$
	$\wedge\ f = X\mathtt{.isfree} \wedge 0 \leq X\mathtt{.isfree} + f_p \leq 1$
$\mathsf{fls}(X;Y)[W]$	$\triangleq \mathsf{emp} \wedge X = Y \wedge W = \epsilon$
	$\vee\ \exists Z, W' \cdot \mathsf{fck}(X;Z) * \mathsf{fls}(Z;Y)[W'] \wedge W = \mathtt{[}X\mathtt{]}.W' \wedge X \neq Y$
$\mathsf{flso}(X, x; Y, y)[W]$	$\triangleq \mathsf{emp} \wedge X = Y \wedge W = \epsilon \wedge x - y \leq 0$
	$\vee\ \exists Z, W' \cdot \mathsf{fck}(X;Z) * \mathsf{flso}(Z, X; Y, y)[W']$
	$\wedge\ W = \mathtt{[}X\mathtt{]}.W' \wedge x - X \leq 0$

the heap list. The atom emp holds iff the domain of the heap is empty. The *points-to atom* $X \mapsto x$ specifies a heap built from one memory block at location X storing the value given by x. The *block atom* $\mathsf{blk}(X;Y)$ holds iff the heap contains a block of memory at location X ending before the location Y. The other predicates are derived from blk and defined in Table 2. Notice that the *chunk header* atom $\mathsf{chd}(X;Y)$ does not expose the fields of the block at location X using the points-to operator of SL. This ease the manipulation of heap list level formulas, e.g., the coalescing of block and chunk atoms into a single block.

Semantics. Formulas φ are interpreted over pairs (I, h) where I is an *interpretation* of logic variables and h is a *heap* mapping a location to a non empty sequence of values stored at this location. Formally, $I \in [(\mathsf{AVar} \cup \mathsf{IVar}) \rightharpoonup \mathbb{V}] \cup [\mathsf{SVar} \rightharpoonup \mathbb{V}^*]$ and $h \in [\mathbb{A} \rightharpoonup \mathbb{V}^+]$ such that $\mathsf{nil} \notin \mathrm{dom}(h)$. Let $h(\ell)[i]$ denote the ith element of $h(\ell)$. Without loss of generality, we consider that a value of type `HDR` is a sequence of values indexed by fields. Table 3 provides the most important semantic rules. We following definitions are standard:

$$[\![\varphi]\!] \triangleq \{(I, h) \mid I, h \models \varphi\} \qquad \varphi \Rightarrow \psi \text{ iff } [\![\varphi]\!] \subseteq [\![\psi]\!]$$

Transformation rules. The definitions in Table 2 imply a set of lemmas used to transform formulas in abstract values (in Section 5). The first set of lemmas is obtained by directing predicate definitions in both directions. For example, each definition $P(\ldots) \triangleq \vee_i \varphi_i$ introduces a set of *folding* lemmas $\varphi_i \Rightarrow P(\ldots)$ and an *unfolding* lemma $P(\ldots) \Rightarrow \vee_i \varphi_i$.

The second class of lemmas concerns list segment predicates in Table 2. The inductive definitions of these predicates satisfy the syntactic constraints defined in [13] for *compositional predicates*. Thus, every $P \in \{\mathsf{hls}, \mathsf{hlsc}, \mathsf{fls}, \mathsf{flso}\}$ satisfies the following *segment composition lemma*:

$$P(X, \vec{x}; Y, \vec{y})[W_1] * P(Y, \vec{y}; Z, \vec{z})[W_2] \wedge W = W_1.W_2 \quad \Rightarrow \quad P(X, \vec{x}; Z, \vec{z})[W] \qquad (2)$$

Table 3. Logic semantics: main rules

$I, h \models \Sigma_H \Supset \Sigma_F$	iff	$I, h \models \Sigma_H$ and $\exists h' \subseteq h$ s.t. $I, h' \models \Sigma_F$
		$\forall \ell \in \mathrm{dom}(h') \cdot h'(\ell)[\mathtt{isfree}] = 1$
$I, h \models \mathsf{emp}$	iff	$\mathrm{dom}(h) = \emptyset$
$I, h \models \mathsf{blk}(X; Y)$	iff	$\mathrm{dom}(h) = I(X) \ \wedge\ I(Y) - I(X) = \lvert h(I(X)) \rvert$
$I, h \models X \mapsto x$	iff	$\mathrm{dom}(h) = I(X) \ \wedge\ h(I(X))[0] = I(x)$
$I, h \models \Sigma_1 * \Sigma_2$	iff	$\exists h_1, h_2$ s.t. $h = h_1 \uplus h_2$ and $I, h_i \models \Sigma_i$ for $i = 1, 2$
$I, h \models \forall X \in W \cdot A_1 \Rightarrow A_2$	iff	$I(W) = [a_1, \ldots, a_n]$ s.t.
		$\forall i \in (1..n)\ I[X \mapsto a_i], h \models A_1 \Rightarrow A_2$
where		
$h_1 \subseteq h_2$	iff	$\mathrm{dom}(h_1) \subseteq \mathrm{dom}(h_2)$ and $\forall \ell \in \mathrm{dom}(h_1) \cdot h_1(\ell) = h_2(\ell)$
$h_1 \circledast h_2$	iff	$\forall\, l_1 \in \mathrm{dom}(h_1), l_2 \in \mathrm{dom}(h_2) \cdot\ l_1 \neq l_2 \wedge$
		$\big((l_1..l_1 + \lvert h_1(l_1)\rvert - 1) \cap (l_2..l_2 + \lvert h_2(l_2)\rvert - 1) = \emptyset\big)$
$h = h_1 \uplus h_2$	iff	$h_1 \circledast h_2$, $\mathrm{dom}(h) = \mathrm{dom}(h_1) \uplus \mathrm{dom}(h_2)$, and
		$(h_1 \uplus h_2)(\ell) \triangleq \begin{cases} h_1(\ell) & \text{if } \ell \in \mathrm{dom}(h_1) \\ h_2(\ell) & \text{if } \ell \in \mathrm{dom}(h_2) \end{cases}$

The reverse implication is applied to split non empty list segments. Finally, the block sub-formulas are removed, split, or folded using the following lemmas:

$$\mathsf{blk}(X;Y) \ \wedge\ X \geq Y \quad \Rightarrow \quad \mathsf{emp} \tag{3}$$

$$\mathsf{blk}(X;Y) \ \wedge\ X < Y \quad \Rightarrow \quad \mathsf{blk}(X;Z) \ *\ \mathsf{blk}(Z;Y) \wedge X \leq Z \leq Y \tag{4}$$

$$\mathsf{blk}(X;Y) \ *\ \mathsf{blk}(Y';Z) \ \wedge\ X \leq Y = Y' \leq Z \quad \Rightarrow \quad \mathsf{blk}(X;Z). \tag{5}$$

4 Abstract Domain for Hierarchical Shape Abstraction

We define in this section the join-semilattice $\langle \mathcal{A}, \sqsubseteq, \sqcup \rangle$ used in our analysis. It is parameterised by a numerical join-semilattice $\langle \mathcal{N}, \sqsubseteq^{\mathcal{N}}, \sqcup^{\mathcal{N}} \rangle$.

Concrete states. Let $\mathbb{X}$ be the set of program variables, where hli is a ghost variable of location type. Values in $\mathcal{A}$ represent sets of concrete states $M \in \mathbb{M}$ of the program. A concrete state M encloses an environment $\epsilon \in \mathbb{E} = \mathbb{X} \to \mathbb{A}$ mapping each program variable to its storing location, and a heap $h : \mathbb{A} \rightharpoonup \mathbb{V}^+$ mapping locations to sequences of values. For simplicity, the symbol hli is overloaded to denote the symbolic location stored by hli.

Abstract values. Values in $\mathcal{A}$ are a restricted form of logic formulas. Generally speaking, $\mathcal{A}$ is a *co-fibered product* [6] of an *extended symbolic heap domain* for the spatial part and a *data word* domain [3] for the pure part. More precisely, $\mathcal{A}$ includes a special value for $\top$ and finite mappings of the form:

$$a^\sharp \ ::= \ \{\langle \epsilon_i^\sharp, \Sigma_i(\overrightarrow{x}, \overrightarrow{W}) \rangle \mapsto \Pi_i(\overrightarrow{x}, \overrightarrow{W} \cup \{W_H, W_F\})\}_{i \in I} \tag{6}$$

where $\epsilon_i^\sharp : \mathbb{X} \to \mathsf{Var}$ is an abstract environment mapping program variables to symbolic location variables, Π_i includes arithmetic constraints allowed by $\mathcal{N}$, and the free variables of each formula are detailed. Furthermore, the usage of sequence variables in Σ_i and Π_i is restricted as follows:

R$_1$: A sequence variable is bound to exactly one list segment atom in Σ_i; thus Σ_i defines an injection between list segment atoms and sequence variables.
R$_2$: Π_i contains only the sequence constraints $W_H = w$ and $W_F = w'$, where W_H and W_F are special variables representing the full sequence of start addresses of chunks in the heap resp. free list levels.

In addition, the universal constraints in the pure formulas Π_i are restricted such that, in any formula $\forall X \in W \cdot A_G \Rightarrow A_U$:

R$_3$: A_G and A_U use only terms where X appears inside a field access $X\texttt{.f}$.
R$_4$: A_G has one of the forms $X\texttt{.size}\#i$ or $X\texttt{.isfree} = i$.

These restrictions still permit to specify DMA policies like first-fit (see eq. (1)) and besides enable an efficient inference of universal constraints.

Internal representation. To ease the manipulation of extended spatial formulas $\langle \epsilon^\sharp, \Sigma \rangle$, we use their Gaifman graph representation, like in Figure 1(c): nodes represent symbolic locations variables and labeled edges represent the spatial atoms in Σ or mappings in $\epsilon^\sharp$. The universal formulas are represented by a map binding each pair (W, A_G) built from a sequence variable and some guard A_G to a numerical abstract value.

Concretisation. An abstract value of the form (6) represents a formula $\vee_i \exists \vec{x}, \vec{W} \cdot \Sigma_i \wedge \Pi_i \wedge \epsilon_i^\sharp$ where each binding $(v, x) \in \epsilon_i^\sharp$ is encoded by $v \mapsto x$ (v is the location where is stored the program variable v). The false formula represents $\bot$, which corresponds to the empty mapping. Therefore, we define the concretisation $\gamma : \mathcal{A} \to \mathbb{M}$ as the denotation of the formula represented by the abstract value, i.e., $\gamma(a^\sharp) = [\![a^\sharp]\!]$.

Ordering. The partial order $\sqsubseteq$ is defined using a sound procedure inspired by [4,13]. For any two non trivial abstract values $a^\sharp, b^\sharp \in \mathcal{A}$, $a^\sharp \sqsubseteq b^\sharp$ if for each binding $\langle \epsilon_i^\sharp, \Sigma_i \rangle \mapsto \Pi_i \in a^\sharp$ there exists a binding $\langle \epsilon_j^\sharp, \Sigma_j \rangle \mapsto \Pi_j \in b^\sharp$ such that:

- there is a graph isomorphism between the Gaifman graphs of spatial formula at each level of abstraction from Σ_i to Σ_j; this isomorphism is defined by a bijection $\Psi : \mathrm{img}(\epsilon_i^\sharp) \to \mathrm{img}(\epsilon_j^\sharp)$ between symbolic location variables and a bijection Ω between sequence variables. Thus, $\Sigma_i[\Psi][\Omega] = \Sigma_j$,
- for each sequence constraint $W = w$ in $\Pi_{W,i}$, $\Omega(W) = \Omega(w)$ is a sequence constraint in $\Pi_{W,j}$,
- $\Psi(\Pi_{\exists,i}) \sqsubseteq^{\mathcal{N}} \Pi_{\exists,j}$,
- for each W defined in Σ_i and for each universal constraint $\forall X \in W \cdot A_G \Rightarrow A_U$ in $\Pi_{\forall,i}$, then $\Pi_{\forall,j}$ contains a universal constraint on $W' = \Omega(W)$ of the form $\forall X \in W' \cdot A_G \Rightarrow A'_U$ such that $\Psi(\Pi_{\exists,i} \wedge A_U) \sqsubseteq^{\mathcal{N}} A'_U$.

The following theorem is a consequence of restrictions on the syntax of formulas used in the abstract values.

Theorem 1 **($\sqsubseteq$ soundness).** *If $a^\sharp \sqsubseteq b^\sharp$ then $[\![a^\sharp]\!] \subseteq [\![b^\sharp]\!]$.*

Join. Given two non-trivial abstract values, $a^\sharp$ and $b^\sharp$, their join is computed by joining the pure parts of bindings with isomorphic shape graphs [3]. Formally, for each two bindings $\langle \epsilon_i^\sharp, \Sigma_i \rangle \mapsto \Pi_i \in a^\sharp$ and $\langle \epsilon_j^\sharp, \Sigma_j \rangle \mapsto \Pi_j \in b^\sharp$ such that there is a graph isomorphism defined by Ψ and Ω between $\langle \epsilon_i^\sharp, \Sigma_i \rangle$ and $\langle \epsilon_j^\sharp, \Sigma_j \rangle$, we define their join to be the mapping $\{\langle \epsilon_j^\sharp, \Sigma_j \rangle \mapsto \Pi\}$ where Π is defined by:

- Π includes the sequence constraints of $b^\sharp$, i.e., $\Pi_W \triangleq \Pi_{W,j}$,
- $\Pi_\exists \triangleq \Psi(\Pi_{\exists,i}) \sqcup^{\mathcal{N}} \Pi_{\exists,j}$,
- for each W sequence variable in $\mathrm{dom}(\Omega)$ and for each type of constraint A_G, then $\Pi_\forall$ contains the formula $\forall X \in \Omega(W) \cdot A_G \Rightarrow \Psi(A_{U,i}) \sqcup^{\mathcal{N}} A_{U,j}$ where $A_{U,i}$ (resp. $A_{U,j}$) is the constraint bound to W (resp. $\Omega(W)$) in $\Pi_{\forall,i}$ (resp. $\Pi_{\forall,j}$) for guard A_G or $\top$ if no such constraint exists.

The join of two bindings with non-isomorphic spatial parts is the union of the two bindings. Then, $(a^\sharp \sqcup b^\sharp)$ computes the join of bindings in $a^\sharp$ with each binding in $b^\sharp$. Intuitively, the operator collects the disjuncts of $a^\sharp$ and $b^\sharp$ but replaces the disjuncts with isomorphic spatial parts by one disjunct which maps the spatial part to the join of the pure parts. Two universal constraints are joined when they concern the same sequence variables and guard A_G since $\big((\forall c \in W \cdot A_G \Rightarrow A_1) \ \vee \ (\forall c \in W \cdot A_G \Rightarrow A_2)\big) \ \Rightarrow \ \big(\forall c \in W \cdot A_G \Rightarrow (A_1 \vee A_2)\big)$.

Theorem 2 ($\sqcup$ soundness). *For any* $a^\sharp, b^\sharp \in \mathcal{A}$, $[\![a^\sharp]\!] \cup [\![b^\sharp]\!] \subseteq [\![a^\sharp \sqcup b^\sharp]\!]$.

Cardinality of the abstract domain. The number of mappings in (6) increases during the symbolic execution by the introduction of new existential variables keeping track of the created chunks. Although the analysis stores only values with linear shape of lists (other shapes are signalled as an error state), the number of linear shapes is exponential in the number of nodes, in general. We avoid this memory explosion by eliminating existential variables using the transformation rules that replace sub-formulas by predicates, an operation classically called *predicate folding*. This operation uses lemmas (2)–(5), as discussed in Section 5. Thus, the domain of abstract values is bounded by an exponential on the number of pointer program variables local to DMA methods which is small in general, e.g., ≤ 3 in our benchmark. However, the domain of pure formulas used in the image of abstract values is not bounded because of integer constants. This fact requires to define widening operators for the data word domain used for the pure constraints.

5 Analysis Algorithm

We now describe the specific issues of the static analysis algorithm based on the hierarchical abstract domain presented in Section 4.

5.1 Main principles

The analysis algorithm consists of the following three steps.

```
int main(void) {
  minit(1024);
  void* p = malloc(20);
  malloc(20);
  mfree(p); p = NULL;
  p = malloc(20);
  malloc(20);
  mfree(p); p = NULL;
  return 0;
}
```

Fig. 2. A client program

The first step targets on discovering the properties of the free and heap lists in order to select a suitable set of list segment predicates. It consists of an inter-procedural and non relational *symbolic execution* of a correct client program like the one in Figure 2. The sets of reachable configurations are represented by abstract values of our domain built over the chunk and block atoms only, i.e., atoms using predicates fck, chk, chd, and blk. Thus, the heap and the free lists are completely unfolded. For example, the abstract value computed for the start location of method `malloc` (line 28 in Figure 1) when executing the client program in Figure 2 is built from four disjuncts whose shape part is given in Figure 3. The client programs are chosen to reveal the heap list organisation (including chunk coalescing) and the shape of the free list. We don't employ the most general client or a client using an incorrect sequence of calls to the DMA methods in order to speed-up this step and avoiding configurations leading to error states.

The second step transforms the abstract values computed by the previous step to obtain an abstract value representing a pre-condition of the DMA method that constrains the global variables and the parameters of the method. For this, the variables of the client program (e.g., `p` in Figure 2) are projected out and folding lemmas are applied to obtain list atoms. For example, the transformation of the abstract value in Figure 3 leads to an abstract value with one disjunct whose spatial part is $\mathsf{hlsc}(X_0, 0; \mathsf{hli}, 0) \Supset \mathsf{flso}(X_0, X_0; \mathsf{nil}, \mathsf{hli})$. The resulting pre-condition is not the weakest one, but it is bigger than (as regards $\sqsubseteq$) the abstract value computed by the symbolic execution at this control location.

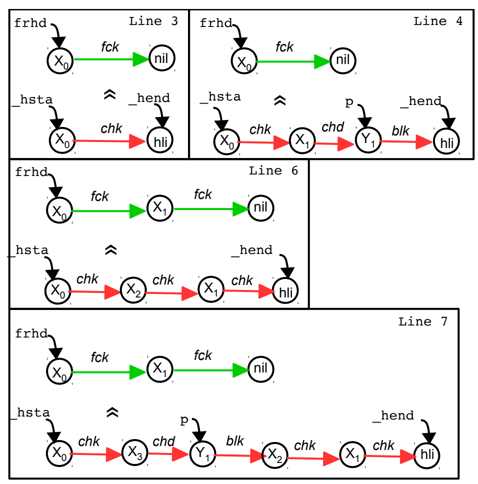


Fig. 3. Spatial formulas at line 28

The third step does forward, non-relational abstract interpretation [8] starting from the computed pre-conditions of each DMA method. The analysis follows the principles of [9,7,10] and uses the widening operator to speed-up the convergence of the fix-point computation for program loops. The original points on abstract transformers concern the transfer of information between abstraction layers in the hierarchical unfolding, splitting, and folding of predicates, as detailed in Section 5.2–5.3. Furthermore, these operations are defined in a modular way, by extending [6] to data word numerical domains. The widening operator uses the widening of data word domain defined in [10].

5.2 Hierarchical unfolding

Abstract transformers compute over-approximations of post-images of atomic conditions and assignments in the program. For the spatial part, the abstract value is transformed such that the program variables read or written by the program operation are constrained using predicates that may capture the effect of the program operation. This transformation is called *predicate unfolding*.

We define the following partial order between predicates $\mathsf{blk} < \mathsf{chd} < \mathsf{chk} < \mathsf{fck} < \mathsf{hls}, \mathsf{hlsc}, \mathsf{fls}, \mathsf{flso}$ which intuitively corresponds to an increasing degree of specialisation. For each program operation s and each pointer variable `x` in s, an atom $P(X; \ldots)$ with $\epsilon^\sharp(\mathtt{x}) = X$ is transformed using lemmas in Section 3 to obtain the atom $Q(X; \ldots)$ such that $Q \leq P$ is the maximal predicate satisfying:

- if s reads the fields of `HDR`, then $Q \leq \mathsf{fck}$,
- if s assigns `x.isfree` or `x.fnx`, then $Q \leq \mathsf{chk}$,
- if s mutates `x` using pointer arithmetics or assigns `x.size`, then $Q \leq \mathsf{chd}$.

We illustrate this procedure on the condition `nxt->size > nunits` at line 37 in Figure 1(b), which reads the field `size`. Applied to the abstract value in Figure 1(c), it requires to unfold the flso predicate from Y_2, to obtain the formula on top of Figure 4. To compute the post-image of the next operation, `nxt->size -= nunits`, the symbolic location Y_2 shall be the root a chd predicate (third case above). Thus, Y_2 is instantiated in the heap list by (i) splitting and then unfolding the hlsc predicate using the segment composition lemma, and (ii) by unfolding chk to obtain the formula at the bottom of Figure 4. The unfolding of chk requires to remove the fck atom from Y_2 in the free list because its definition is not more satisfied at the heap list abstraction level.

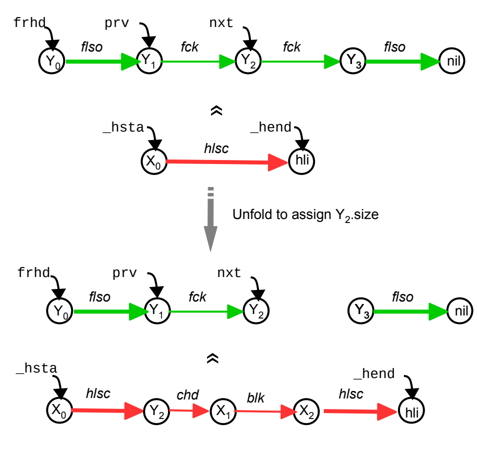


Fig. 4. Hierarchical unfolding at line 38

The next assignment, `nxt += nxt->size`, does not require to transform the predicate rooted in Y_2 because it is already $\leq$ chd. Instead, the transformer adds a new symbolic location Z_1 in the heap list level and constrain it by $Z_1 = Y_2 + Y_2\mathtt{.size} \times \mathtt{sizeof(HDR)}$. If Z_1 goes beyond the limit of the block of the chunk starting at Y_2 (i.e., outside the interval $[X_1, X_2)$ in Figure 4), the analysis signals a chunk breaking. Otherwise, the blk atom from X_1 is split using lemma (4) to insert Z_1; the result is given in the top part of Figure 5.

5.3 Hierarchical folding

To reduce the size of abstract values, the abstract transformers finish their computation on a binding $\langle \epsilon_i^\sharp, \Sigma_i \rangle \mapsto \Pi_i$ by eliminating the symbolic locations which are not cut-points in Σ_i. The elimination uses predicate folding lemmas like (2) or (5) to replace sub-formulas using these variables by one predicate atom. The graph representation eases the computation of sub-formulas matching the left part of a folding lemma.

More precisely, the elimination process has the following steps. First, it searches sequences of sub-formula of the form $\mathsf{chd}(X_0; X_1) * \mathsf{blk}(X_1; X_2) \ldots * \mathsf{blk}(X_{n-1}; X_n)$ where none of X_i $(1 \leq i < n)$ is in $\mathrm{img}(\epsilon^\sharp)$. Such sub-formulas are folded into $\mathsf{chk}(X_0; X_n)$ if the pure part of the abstract value implies $X_0.\mathtt{size} \times \mathtt{sizeof(HDR)} = X_n - X_0$ (see Table 2). We use the variable elimination provided by the numerical domain $\mathcal{N}$ to project out $\{X_1, \ldots, X_{n-1}\}$ from the pure part. Furthermore, if the pure part implies $X_0.\mathtt{isfree} = 1$, then the chunk atom (and its start address) is promoted as fck to the free list level.

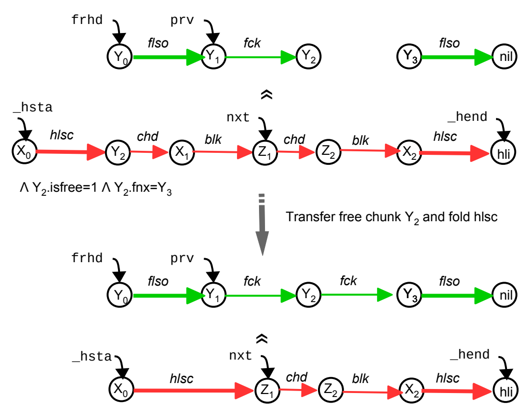


Fig. 5. Hierarchical folding after line 48

This step is illustrated on sub-formulas $\mathsf{chd}(Y_2; X_1) * \mathsf{blk}(X_1; Z_1)$ at the top of Figure 5. The second step folds hlsc list segments by applying their inductive definition and the composition lemma (2). The atoms $\mathsf{chk}(X_0; \ldots)$ for which the free list level contains an atom $\mathsf{fck}(X_0; \ldots)$ may be folded at the heap list level into list segments due to the semantics of $\Supset$. For example, the chunk starting from location Y_2 is folded inside a hlsc segment in the formula at the bottom of Figure 5. Notice that folding of list segments implies the update of sequence and universal constraints like in [10].

6 Experiments

We implemented the abstract domain and the analysis algorithm in Ocaml as a plug-in of the Frama-C platform [19]. We are using several modules of Frama-C, e.g., C parsing, abstract syntax tree transformations, and the fix-point computation. The data word domain uses as numerical join-lattice $\mathcal{N}$ the library of polyhedra with congruence constraints included in Apron [17]. To obtain precise numerical invariants, we transform program statements using bit-vector operations (e.g., line 16 of Figure 1(a)) into statements allowed by the polyhedra domain which over-approximate the original effet.

We applied our analysis on the benchmark of free list DMA in Table 4. (Detailed experimental results are available in [**?**].) DKff and DKbf are imple-

Table 4. Benchmark of DMA

DMA	*LOC*	*List Pred.*	*Time (s)*	$\|a^\sharp\|$	$\|W_H\|/\|W_F\|$	*Invariants*
DKFF [20]	176	hlsc, flso	0.05	25	8/5	first-fit, MIN_SIZE-size
DKBF [20]	130	hlsc, flso	0.05	26	8/6	best-fit, MIN_SIZE-size
LA [1]	181	hlsc, flso	0.07	25	8/5	first-fit, 0-size
DKNF [20]	137	hlsc, flso	0.05	30	8/6	first-fit, MIN_SIZE-size
KR [18]	284	hlsc, flso	2.8	32	8/6	first-fit, 0-size

mentations of Algorithms A and B from Section 2.5 of [20]. These DMA keep an acyclic free list sorted by the start addresses of chunks. The deallocation does coalescing of successive free chunks. The allocation implements a first-fit resp. best-fit policy such that the fitting chunk is not split if the remaining free part is less than `MIN_SIZE` (variant proposed in [20]). This property is expressed by the following sub-formula of the invariant "`MIN_SIZE`-size" (for `MIN_SIZE`=8):

$$\forall X \in W_H \cdot X.\mathtt{size} \geq 8 \tag{7}$$

which is inferred by our analysis. The first-fit policy is implied by an abstract value similar to the one in equation (1) (page 5). The best-fit policy is implied by a value using the constraint:

$$\forall X \in W_i \cdot X.\mathtt{size} \geq \mathtt{rsz} \Rightarrow X.\mathtt{size} \geq Y.\mathtt{size} \tag{8}$$

where `rsz` is the requested size, Y is the symbolic address of the fitting chunk, and W_i represents a list segment around the fitting chunk. LA is our running example in Figure 1; it follows the same principles as DKFF, but get rid of the constraint for chunk splitting. For this case study, our analyser infers the "0-size" invariant, i.e., $\forall X \in W_H \cdot X.\mathtt{size} \geq 4$ (=`sizeof(HDR)`). Notice that the code analysed fixes an obvious problem of the `malloc` method published in [1]. DKNF implements the next-fit policy using the "roving pointer" technique proposed in [20]: a global variable points to the chunk in the free list involved in the last allocation or deallocation; `malloc` searches for a fitting free chunk starting from this pointer. Thus, the next-fit policy is a first-fit from the roving pointer. DKNF is challenging because the roving pointer introduces a case splitting that increases the size (number of disjuncts) in abstract values. The KR allocator [18] keeps a circular singly linked list, circularly sorted by the chunk start addresses; the start of the free list points to the last deallocated block. The circular shape of the list requires to keep track of the free chunk with the biggest start address and this increases the size of the abstract values.

The analysis times reported in Table 4 have been obtained on a 2.53 GHz Intel Core 2 Duo laptop with 4GB of RAM. They correspond to the total time of the three steps of the analysis starting from the client given in Figure 2. The universally quantified invariants inferred for DMA policies are given in the last column. Columns $|a^\sharp|$ and $|W_H|/|W_F|$ provide the maximum number of disjuncts generated for an abstract value resp. the maximum number of predicate atoms in each abstraction level.

7 Related Work and Conclusion

Our analysis infers complex invariants of free list DMA implementations due to the combination of two ingredients: the hierarchical representation of the shape of the memory region managed by the DMA and an abstract domain for the numerical constraints based on universally quantified formulas. The abstract domain has a clear logical definition, which facilitates the use of the inferred invariants by other verification methods.

The proposed abstract domain extends previous works [5,3,10,22,11]. We consider the SL fragment proposed in [5] to analyse programs using pointer arithmetic. We enrich this fragment in both spatial and pure formulas to infer a richer class of invariants. E.g., we add a heap list level to track properties like chunk overlapping and universal constraints to infer first-fit policy invariants.

The split of shape abstraction on levels is inspired by work on overlaid data structures [22,11]. We consider here a specific overlapping schema based on set inclusion which is adequate for the class of DMA we consider. We propose new abstract transformers which do not require user annotations like in [22]. Another hierarchical analysis of shape and numeric properties has been proposed in [26]. They consider the analysis of linked data structures coded in arrays and track the shape of these data structures and not the organisation of the set of free chunks. Their approach is not based on logic and the invariants inferred on the content of list segments are simpler.

Our abstract domain includes a simpler version of the data word domain proposed in [3,10], since the universal constraints quantify only one position in the list. Several abstract domains have been defined to infer invariants over arrays, e.g., [14] for array sizes, [15,16] for array content. These works infer invariants of different kind on array partitions and they can not be applied directly to sequences of addresses. Recently, [23] defined an abstract domain for the analysis of array properties and applies it to the Minix 1.1 DMA, which uses chunks of fixed size. A modular combination of shape and numerical domains has been proposed in [6]. We extend their proposal to combine shape domains with domains on sequences of integers. Precise analyses exist for low level code in C [24] or for binary code [2]. They efficiently track properties about pointer alignment and memory region separations, but can not infer shape properties.

References

1. L. Aldridge. Memory allocation in C. *Embedded Systems Programming*, pages 35–42, August 2008.
2. G. Balakrishnan and T. W. Reps. Recency-abstraction for heap-allocated storage. In *SAS*, volume 4134 of *LNCS*, pages 221–239. Springer, 2006.
3. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI*, pages 578–589. ACM, 2011.
4. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In *ATVA*, volume 7561 of *LNCS*, pages 167–182. Springer, 2012.

5. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, volume 4134 of *LNCS*, pages 182–203. Springer, 2006.
6. B. E. Chang and X. Rival. Modular construction of shape-numeric analyzers. In *Semantics, Abstract Interpretation, and Reasoning about Programs*, volume 129 of *EPTCS*, pages 161–185, 2013.
7. B. E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In *SAS*, volume 4634 of *LNCS*, pages 384–401. Springer, 2007.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.
9. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920, pages 287–302. Springer, 2006.
10. C. Dragoi. *Automated verification of heap-manipulating programs with infinite data*. PhD thesis, University Paris Diderot, 2011.
11. C. Dragoi, C. Enea, and M. Sighireanu. Local shape analysis for overlaid data structures. In *SAS*, volume 7935 of *LNCS*, pages 150–171. Springer, 2013.
12. C. Enea, V. Saveluc, and M. Sighireanu. Compositional invariant checking for overlaid and nested linked lists. In *ESOP*, volume 7792 of *LNCS*, pages 129–148. Springer, 2013.
13. C. Enea, M. Sighireanu, and Z. Wu. On automated lemma generation for separation logic with inductive definitions. In *ATVA*, LNCS, pages 80–96. Springer, 2015.
14. S. Gulwani, T. Lev-Ami, and S. Sagiv. A combination framework for tracking partition sizes. In *POPL*, pages 239–251. ACM, 2009.
15. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246. ACM, 2008.
16. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348. ACM, 2008.
17. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.
18. B. W. Kernighan and D. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.
19. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *FAC*, 27(3):573–609, 2015.
20. D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.
21. D. Lea. `dlmalloc`. ftp://gee.cs.oswego.edu/pub/misc/malloc.c, 2012.
22. O. Lee, H. Yang, and R. Petersen. Program analysis for overlaid data structures. In *CAV*, volume 6806 of *LNCS*, pages 592–608. Springer, 2011.
23. J. Liu and X. Rival. Abstraction of arrays based on non contiguous partitions. In *VMCAI*, volume 8931 of *LNCS*, pages 282–299. Springer, 2015.
24. A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES*, pages 54–63. ACM, 2006.
25. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, LNCS, pages 1–19. Springer, 2001.
26. P. Sotin and X. Rival. Hierarchical shape abstraction of dynamic structures in static blocks. In *APLAS*, volume 7705 of *LNCS*, pages 131–147. Springer, 2012.
27. P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *IWMM*, volume 986 of *LNCS*, pages 1–116. Springer, 1995.

A Answers to reviewers' comments

We thank the reviewers for their interesting and constructive comments. We have fixed up all the typos revealed by the reviewers and we have tried to improve the text using the reviewers' recommandations.

In the following, we discuss the main comments and questions of reviewers and how they have been addressed in the revised version of our paper.

Operator $\Supset$: This operator is used to specify that the set of addresses of chunks in the free list is exactly the set of addresses of free chunks in the heap list. Reviewer 1 was concerned about the lack of generality of this operator. We recognise this fact. However, we defend this choice mainly for efficiency reasons. A more general operator could keep track of more complex relations between sets of addresses, as proposed, e.g., in [12]. However, this expressivity power is to be balanced against the complexity of the abstract domain required to manage such constraints. The particular constraints exhibited by the invariants of free list DMA keep us away from such complexity.

Static analysis versus testing: Reviewer 3 pointed out that our previous presentation of the analysis algorithm may be interpreted as the test of a specific client, which may be done without an expensive static analysis. (S)he asked why we don't use a more general client in order to exploit the power of the static analysis. To avoid such misinterpretation, we enrich Section 5 with more details on the analysis algorithm, thanks to the additional page. We explain that the client program is used only by the first step of our analysis (which is a symbolic execution not a concrete execution like in testing) to discover the set of predicates to be used in the abstract interpretation algorithm. This first step is followed by a static analysis based on abstract interpretation. Notice that the result of our analysis is not only a diagnosis (correct or flawed) like in testing, but also a set of invariants satisfied by the DMA configurations.

Expensive abstract domain: Reviewers 2 and 3 were concerned about the complexity of the proposed abstract domain because of its "highly disjunctive nature" and the use of precise numerical abstract domains. For general programs, they might have right. For our specific application, we do not experience the drawbacks of this high precision. As we commented on paragraph "Cardinality of the abstract domain" (page 10), the DMA implementations we consider employ few variables (pointer or integer). Thus, the number of disjuncts is reduced. Moreover, as explained in the previous paragraph, our analysis is directed by the abstract configurations reachable in clients using correct sequences of calls of the DMA methods, which reduces further the number of disjuncts in abstract values. Finally, the numerical abstract domain used is based on polyhedra with modulo constraints but the last constraints are not fully general. Indeed, we are using only constraints of the form $a \equiv_k 0$ with $k \in \{4, 8\}$.