Under consideration for publication in Formal Aspects of Computing

Hierarchical Shape Abstraction for Analysis of Dynamic Memory Allocators

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

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

Abstract. Static analysis of dynamic memory allocators is a challenging task because of the complexity of data structures used. This paper addresses this challenge for a class of memory allocators, the free list based memory allocators. It proposes an abstract domain that tracks shape and numerical properties about both the heap and the free lists used by these allocators. 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 an analyzer based on abstract interpretation to infer DMA invariants and we show its effectiveness on several implementations of dynamic memory allocators.

Keywords: Dynamic Memory Allocators; Shape Analysis; Separation Logic

1. Introduction

Dynamic memory allocators (DMA) intend to increase effectiveness of memory management and are widely used in kernel part of operating systems or in libraries of general programming language. Their main task is to provide quick access to the heap while avoiding unsafe accesses and memory leakage. The ubiquity of dynamic memory in nowadays software makes crucial to guarantee the correctness of DMA.

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 a memory region reserved in the heap 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

Correspondence and offprint requests to: bfang@irif.fr

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* [26, 36], that records free chunk in a list. This class of DMA includes textbook examples [26, 24] and real-world allocators [27].

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. For example, 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. Also, the start addresses of chunks shall be aligned to some given constant. Moreover, 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 [31, 7, 34]. In [31], 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 [7]. 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) [32, 33] 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 [34] 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 linked lists 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 the heap list and the free list. We define an abstract domain which uses logic formulas to abstract DMA configurations. The logic proposed, called SLH, extends the fragment of symbolic heaps of SL [4] with a hierarchical composition operator, $\Subset$, 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 [5]. 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.

Introductory example We demonstrate the core ideas of our method on the C code presented in Figure 1 which is extracted from a DMA in our benchmark, the Aldridge's allocator [2], called LA in the following.

The code declares first an internal data type, `HDR`, used to build both the heap and the free list as follows. The field `size` stores the full size of the chunk (in blocks of `sizeof(HDR)` bytes). In the heap list, this information is used to obtain the start address of the next chunk by adding to the start address of the current chunk its size in bytes. The field `fnx` stores the start address of the next free chunk and it is used to form the singly linked list of free chunks, i.e., the free list. We added the ghost field *isfree* in this data type to mark explicitly free chunks and to simplify the presentation of our method. Lines 7–10 declare several globals variables: `_hsta` and `_hend` represent the first address of the entire memory block and the address right after the end of memory block respectively, `frhd` stores the address of the head of the free list, and `memleft` counts the number of free bytes in the memory region.

The method `minit` initializes these global variables and makes a reservation for a memory region such that it may store the requested `sz` bytes plus a header value. The memory is reserved due to the call of the system routine `sbrk`, that extends the data segment of the requesting process by the input value and returns the address representing the old limit of this segment. In the initial state, the heap list and the free list start at the same address, the beginning of the memory region reserved, `_hsta`. They contain only one chunk, which is set as free.

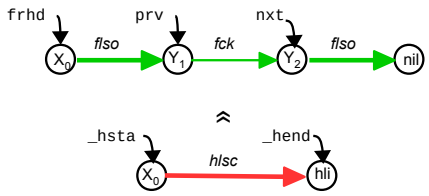
The method `malloc` tries to fulfil a request for allocating `nbytes` bytes. For this, it searches a free chunk whose body has size at least `nbytes` using the loop at lines 34–51 which traverses the free list and stops at

```
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 }
```

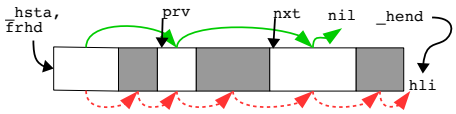
(a) Globals and initialisation

```
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 where green (resp. red) arrows represent the successor relation for the free (resp. heap) list

Fig. 1. Running example with code

the first free chunk satisfying this constraint; this way of choosing the free chunk is called the *first-fit policy*. If the free chunk is much larger, then it is split in two parts and the second part, i.e., at the end of the initial chunk, is allocated.

After several calls of allocation and deallocation methods, the memory region will be split into several chunks including free and busy chunks. An intuitive view of the concrete state of the DMA at line 36 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. Notice that the free list is sorted by the start address of free chunks in this example. This fact eases the coalescing of successive free chunks. Indeed, LA implements the *early coalescing policy* which prevents to store in the heap list two successive free chunks. Therefore, the deallocation method of LA (not shown here) merges continuous free chunks into one bigger free chunk and updates both lists accordingly.

Our method abstracts set of states of the DMA using sets of formulas, each formula being a conjunction of predicate atoms. Figure 1(c) gives a graph representation for such a formula that specifies the concrete state represented in Figure 1(d). The heap list satisfying the early coalescing is specified by the atom $\mathsf{hlsc}(X_0, \mathsf{hli})$ where hlsc is an inductively defined predicate (formally introduced in Section 2). The value X_0 and hli are stored in variables `_hsta` resp. `_hend`, which is represented by arrows sourcing these variables. The free list is abstracted by three atoms building the upper graph starting from X_0 also. The atom $\mathsf{flso}(X_0, Y_1)$ specifies the free list segment from the start of the list `frhd` to the location stored in `prv`, represented by the logic variable Y_1. The atom $\mathsf{fck}(Y_1, Y_2)$ specifies a free chunk at location Y_1 which stores in his *fnx* fiel the location

values	$v \in \mathbb{V}$	addresses	$\alpha \in \mathbb{A}$, $\mathbb{A} \subseteq \mathbb{V}$
fields	$f \in \mathbb{F}$	variables	$x \in \mathbb{X}$
types	$T \in \mathbb{T}$	typing	$\tau : \mathbb{X} \cup \mathbb{F} \to \mathbb{T}$
memories	$m \in \mathbb{M} \mathrel{\hat{=}} \mathbb{E} \times \mathbb{H}$		
environments	$\xi \in \mathbb{E} \mathrel{\hat{=}} [\mathbb{X} \to \mathbb{A}]$	stores	$\sigma \in \mathbb{H} \mathrel{\hat{=}} [\mathbb{A} \to \mathbb{V}]$

Fig. 2. Memory model

Y_2, stored by variable `nxt`. The last atom $\mathsf{flso}(Y_2, \mathsf{nil})$ specifies another free list segment, suffix of the free list until null. The predicates used in these atoms are specified inductively using an extension of separation logic formally introduced in Section 2.

Both graphs specify fully (for the lower graph) or partially (for the upper graph) the same concrete memory region. The upper part highlights only the free list, but all the chunks in the free lists are also chunks of the heap list specified by the lower graph. To compose these two abstractions of the memory region, we introduce a new operator, the hierarchical composition "$\Subset$", which allows to relate the two levels of abstraction while keeping separated properties related with each kind of list used. For example, we are able to express the early coalescing property of the heap lists without interfering with the free list, which is not concerned about this policy. The formula obtained are used as abstract values in order analysis algorithm to represent program configurations, as presented in Section 4. The separation of concerns provided by the hierarchical composition is used by the analysis we propose in order to focus only on the abstraction level required by the statements to be analysed. For example, the loop traversing the free list at lines 34–35 requires to reason only at the free list level. The details on this analysis are provided in Section 5.

Contribution To summarise, we propose a static analysis of free list memory allocators based on the hierarchical representation of the allocator configurations which has the following main advantages:

- the *high precision of the abstraction* which allows us to capture complex properties of these allocators like coalescing policies for freed chunks or complex shapes of the free list,
- the *strong logical basis* provides means for the used of the inferred invariants in other verification methods, for example deductive methods, and
- the *modularity* of the abstract domain that combines existing abstract domains for the analysis of linked lists with integer data.

Outline The paper is organized as follows. Section 2 formalizes the logic SLH whose formulas are used as values of the abstract domain. Section 3 defines the syntax and the semantics of the programming language analysed. Section 4 defines the abstract domain used by hierarchical shape analysis of DMA. Section 5 presents the static analysis algorithm based on our abstract domain and gives the detailed abstract transformers implementing the abstract semantics of the programming language. We report on the experimental results of this work in Section 6. The related work and the conclusion are commented in Section 7.

2. Specification Logic

This section introduces the SLH logic used in our analysis method to abstract sets of DMA configurations. SLH extends the fragment of Separation Logic [32, 33] proposed by Calcagno et al. [7] with several features in order to capture precisely the properties of configurations and policies exhibited by the free list DMA.

2.1. Memory Model

A formula of SLH specifies a set of configurations of a DMA, i.e., the values stored by each variable in the DMA code and by each byte in the memory region managed by the DMA. More precisely, we consider a model of program configurations, formally defined in Figure 2, that abstracts some low level details of program states as follows. The absence of recursive functions in the DMA code allows us to model the program stack

$X, Y, \mathsf{nil}, \mathsf{hli} \in \mathsf{AVar}$ location variables
$W \in \mathsf{SVar}$ sequence variables
$i, j \in \mathsf{IVar}$ integer variables
$x \in \mathsf{Var}$ logic variable
$X\texttt{.f}$ field access term, $\texttt{f} \in \mathbb{F}$
k integer constant
$\# \in \{=, \neq, \leq, \geq\}$ comparison operators
o, Δ integer operator resp. formula

φ	::=	$\exists \vec{x} \cdot \Pi \wedge \Sigma \mid \varphi \vee \varphi$	formulas
Π	::=	$A \mid \forall X \in W \cdot A \Rightarrow A \mid W = w \mid \Pi \wedge \Pi$	pure formulas
t	::=	$k \mid i \mid X\texttt{.f} \mid o(t_1, \ldots, t_n)$	integer terms
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	::=	$\Sigma_H \circledast \Sigma_F$	spatial formulas
Σ_H	::=	$\mathsf{emp} \mid X \mapsto x \mid \mathsf{blk}(X;Y) \mid \mathsf{chd}(X;Y) \mid \mathsf{chk}(X;Y) \mid$ $\mathsf{hls}(X;Y)[W] \mid \mathsf{hlsc}(X,i;Y,j)[W] \mid \Sigma_H * \Sigma_H$	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

Fig. 3. Syntax of SLH

by a mapping ξ, called environment, that associates each program variable to the unique address where its value is stored. The set of all environment mappings is denoted by $\mathbb{E}$. The memory region managed by the DMA is modelled by a mapping σ, called store, that maps each address of the memory region to a value. The set of all store mappings is denoted by $\mathbb{H}$. A memory model m is built from an environment and a store.

In the following, we suppose that the domain $\mathbb{V}$ of values stored in the memory is $\mathbb{N}$, the natural numbers, equipped with the usual comparison, arithmetic, and bitwise operations. For the sub-domain of addresses, $\mathbb{A}$, the addition is done only between an address and a non address value. In $\mathbb{A}$, we consider that nil is a special address representing the null address. Also, the stored values are typed using types in $\mathbb{T}$ that represent either subsets of naturals (e.g., `size_t`), addresses (e.g., `HDR*`), or tuples of such types (e.g., `HDR`). The type `HDR` is predefined and its elements are labeled by fields, that belong to a finite set $\mathbb{F}$, and are typed using the typing function τ. Without loss of generality, we consider that `HDR` contains at least the three fields `size`, `fnx`, and `isfree` typed like in Figure 1(a).

We denote by S a set of memory states, i.e., a subset of $\mathbb{M}$. Also, we denote by $\sigma[\alpha \leftarrow v]$ the store obtained by the update of σ at address α with value v.

2.2. Syntax

The syntax of SLH is defined in Figure 3. Logic variables in AVar are interpreted over addresses in $\mathbb{A}$. To simplify the presentation, we employ the same name nil for the location variable always mapped to the address nil. Also, the address variable hli denotes the end of the memory region managed by the DMA. Sequence variables in SVar are interpreted over words of addresses in $\mathbb{A} \setminus \{\mathsf{nil}\}$. The term $X\texttt{.f}$ denotes the value stored at the address $X + \texttt{f}$, i.e., the value of the C expression `*(X+f)`. More integer terms are obtained using operations over integers. These operations and the constraints built over integer terms are a parameter our logic; in the analysis framework we built, this parameter is instantiated with binary arithmetic operations and linear constraints. Formulas are in disjunctive normal form. Each disjunct is an existentially quantified formula built from a pure formula Π and a spatial formula Σ.

Pure formulas Π specify constraints over location, integer, and sequence variables using comparison operations, integer constraints Δ, and sequence constraints. We restrict the constraints over location variables to difference constraints. The sequence constraints are of two forms. The universal constraints $\forall X \in W \cdot A_G \Rightarrow A_U$ constrain the addresses and the values stored at these addresses in the word of addresses bound to W.

$$\mathsf{chd}(X;Y) \triangleq \mathsf{blk}(X;Y) \wedge \texttt{sizeof(HDR)} = Y - X \wedge X \equiv_{\texttt{sizeof(HDR)}} 0$$

$$\mathsf{chk}(X;Y) \triangleq \exists Z \cdot \mathsf{chd}(X;Z) * \mathsf{blk}(Z;Y) \wedge X\texttt{.size} \times \texttt{sizeof(HDR)} = Y - X$$

$$\mathsf{fck}(X;Y) \triangleq \exists Z \cdot \mathsf{chk}(X;Z) \wedge X\texttt{.isfree} = 1 \wedge X\texttt{.fnx} = Y$$

$$\mathsf{hls}(X;Y)[W] \triangleq \mathsf{emp} \wedge X = Y \wedge W = \epsilon \vee \exists Z, W' \cdot \mathsf{chk}(X;Z) * \mathsf{hls}(Z;Y)[W'] \wedge W = \texttt{[}X\texttt{]}.W'$$

$$\mathsf{hlsc}(X, f_p; Y, f_\ell)[W] \triangleq \mathsf{emp} \wedge X = Y \wedge W = \epsilon \wedge 0 \leq f_p + f_\ell \leq 1 \vee \exists Z, W', f \cdot \mathsf{chk}(X;Z) * \mathsf{hlsc}(Z, f; Y, f_\ell)[W'] \wedge W = \texttt{[}X\texttt{]}.W' \wedge f = X\texttt{.isfree} \wedge 0 \leq X\texttt{.isfree} + f_p \leq 1$$

$$\mathsf{fls}(X;Y)[W] \triangleq \mathsf{emp} \wedge X = Y \wedge W = \epsilon \vee \exists Z, W' \cdot \mathsf{fck}(X;Z) * \mathsf{fls}(Z;Y)[W'] \wedge W = \texttt{[}X\texttt{]}.W' \wedge X \neq Y$$

$$\mathsf{flso}(X, x; Y, y)[W] \triangleq \mathsf{emp} \wedge X = Y \wedge W = \epsilon \wedge x - y \leq 0 \vee \exists Z, W' \cdot \mathsf{fck}(X;Z) * \mathsf{flso}(Z, X; Y, y)[W'] \wedge W = \texttt{[}X\texttt{]}.W' \wedge x - X \leq 0$$

Fig. 4. Derived predicates

To obtain an efficient analysis, we use only a fixed form of constraints for A_G and A_U, mainly the arithmetic constraints used in numerical abstract domain. The sequences variables are defined using equations $W = w$ where the sequence term w is either an empty sequence ϵ, a singleton address $[X]$, or the concatenation of two sequences $w \cdot w'$. As usual, ϵ is the neutral element for concatenation. 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).

Spatial formulas Σ specify the content of the heap using two views that are constrained by the hierarchical composition operator $\Subset$. The left operand of $\Subset$, Σ_H, specifies the heap list maintained by the DMA in the memory region and the locations outside this region. The right operand of $\Subset$, Σ_F, specifies only the free list. The operator $\Subset$ requires that the set of chunks in the free list view is exactly the set of chunks in the heap list whose field `isfree` has value true (see Figure **??** and the next section). Notice that the operator $\Subset$ 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.

Heap list formulas include the classic formulas of Separation Logic for the *empty heap* emp, the *points-to atom* $X \mapsto x$ specifying a heap built from one memory block at location X storing the value given by x, and the *separating conjunction* of two disjoint heaps. In addition, we introduce the *block atom* $\mathsf{blk}(X;Y)$ proposed in [7] to specify a heap that contains 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 $\mathsf{blk}(\texttt{\_hsta};\texttt{\_hend})$. The other atoms use predicates derived from blk and defined by SLH formulas in Figure 4. An atom $\mathsf{chd}(X;Y)$ specifies a memory block $\mathsf{blk}(X;Y)$ storing a value of type `HDR`; the values stored in the fields are obtained using the terms $X\texttt{.size}$, $X\texttt{.fnx}$, and $X\texttt{.isfree}$ respectively. An atom $\mathsf{chk}(X;Y)$ 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 $X\texttt{.size}\times\texttt{sizeof(HDR)}$. An atom $\mathsf{hls}(X;Y)[W]$ specifies a well formed heap list segment starting at address X and ending before Y. The predicate hls is defined inductively using a disjunction that contains the base case of the empty list and the case of a chunk stating in X and followed by a heap list. The variable W registers the sequence of start addresses of chunks in the list segment and it is used to put additional constraints on the fields of these chunks. The atom $\mathsf{hlsc}(X, i; Y, j)[W]$ specifies heap list obtained in DMA with early coalescing of free chunks (i.e., coalescing at `free`), where two adjacent chunks can not be both free. The parameters i and j are the free status of the before the first resp. last chunk in the list.

$I,h \models \Sigma_H \Rsh \Sigma_F$	iff	$I,h \models \Sigma_H$ and $\exists h' \subseteq h$ s.t. $I,h' \models \Sigma_F$ $\forall \ell \in \mathrm{dom}(h') \cdot h'(\ell)[\texttt{isfree}] = 1$
$I,h \models \mathsf{emp}$	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 $(h_1 \uplus h_2)(\ell) \triangleq \begin{cases} h_1(\ell) & \text{if } \ell \in \mathrm{dom}(h_1) \\ h_2(\ell) & \text{if } \ell \in \mathrm{dom}(h_2) \end{cases}$

Fig. 5. Logic semantics: main rules

Free list formulas 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 Figure 4):

- The predicate $\mathsf{fck}(X;Y)$ specifies a chunk $\mathsf{chk}(X;\ldots)$ starting at X, with $X\texttt{.fnx}$ bound to Y and $X\texttt{.isfree}$ 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.

2.3. 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. Figure 5 provides the most important semantic rules. We following definitions are standard:

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

2.4. Examples of Specification

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 [5]. For example, the first-fit policy obtained at line 37 of `malloc`, is specified by:

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

where Y_2 is the symbolic address stored in the program variable `nxt`.

2.5. Employability for Program Analysis and Verification

2.5.1. Decidability

[1.satisfiability:
2.entailment:
]

2.5.2. Transformation rules

The definitions in Figure 4 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 Figure 4. The inductive definitions of these predicates satisfy the syntactic constraints defined in [19] for *compositional predicates*. Thus, every $P \in \{\mathsf{hls}, \mathsf{hlsc}, \mathsf{fls}, \mathsf{flso}\}$ satisfies the following *segment composition lemma*:

Lemma 1. $P(X, \vec{x}; Y, \vec{y})[W_1] * P(Y, \vec{y}; Z, \vec{z})[W_2] \wedge W = W_1.W_2 \quad \Rightarrow \quad P(X, \vec{x}; Z, \vec{z})[W]$

Lemma 1 states two separated inductive list segments can be folded as a summary list segment if these two list segments are connected by separation operator $*$. The reverse implication is applied to split non empty list segments into two or more small list segments.

For predicates of block, the sub-formulas are removed, split, or folded using the following lemmas:

Lemma 2. $\mathsf{blk}(X;Y) \ \wedge \ X \geq Y \quad \Rightarrow \quad \mathsf{emp}$

Lemma 3. $\mathsf{blk}(X;Y) \ \wedge \ X < Y \quad \Rightarrow \quad \mathsf{blk}(X;Z) \ * \ \mathsf{blk}(Z;Y) \wedge X \leq Z \leq Y$

Lemma 4. $\mathsf{blk}(X;Y) \ * \ \mathsf{blk}(Y';Z) \ \wedge \ X \leq Y = Y' \leq Z \quad \Rightarrow \quad \mathsf{blk}(X;Z).$

Lemma 2 is used to transform incorrect block into empty case. Lemma 4 is the reverse of lemma 3. By using these transformation rules, the abstract values partially presented by the predicate can be converted to the appropriate predicates when the precision and efficiency are taken into account during the static analysis.

3. Concrete Semantics

In this section, we describe the programming language we considered in this work and introduce the concrete program semantics. We define a graph representation of concrete memory state and its transformation rules. The last part of this section describes the preliminaries on program analysis based on abstract interpretation.

3.1. Programming language

The programs we consider is a class of typed imperative programs manipulating memory regions. The programs include structure types and integers, pointers and pointer arithmetics, pointer casting, bit fields and bit-wise operations, standard system routines for data segment manipulation (`sbrk` and `brk`). String manipulation, arrays, pointers to functions, float arithmetics, concurrency constructs, are not present. Moreover, recursive functions are not used. We formalise this observation by fixing in Figure 6 a standard imperative programming which capture the features used in the C code analysed. Notice that we are analyzing C code and not this demonstration language.

3.2. Syntax of programs

A program is a sequence of functions declarations and statements. Statements include a do-nothing statement, sequencing, non-deterministic branching, and looping. Standard `if` and `while` statements are defined as usual

statements	s	$::=$	$c \mid \texttt{skip} \mid s_1; s_2 \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c\ \texttt{done}$
commands	c	$::=$	$\ell\texttt{:=}e \mid \ell\texttt{:=sbrk}(e)$
locations	ℓ	$::=$	$x \mid \ell\,.\,f \mid \texttt{*}e$
expressions	e	$::=$	$v \mid \ell \mid \texttt{\&}\ell \mid \oplus(e_1, \ldots, e_n)$
bexpressions	b	$::=$	$\neg b \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \mid e_1 \odot e_2$
operators	$\oplus$	$\supset$	$\{+, -, \times, /, \&, =\}$
comparisons	$\odot$	$\supset$	$\{<, \leq, >, \geq, \neq, ==\}$

Fig. 6. Program syntax

$$
\begin{array}{rcl}
\mathcal{E}[\![e]\!] & : & \mathbb{M} \to \mathbb{V} \\
\mathcal{E}[\![\ell]\!](\xi,\sigma) & \triangleq & \sigma(\mathcal{L}[\![\ell]\!](\xi,\sigma)) \\
\mathcal{E}[\![\texttt{\&}\ell]\!] & \triangleq & \mathcal{L}[\![\ell]\!] \\
\\
\mathcal{L}[\![\ell]\!] & : & \mathbb{M} \to \mathbb{A} \\
\mathcal{L}[\![x]\!](\xi,\sigma) & \triangleq & \xi(x) \\
\mathcal{L}[\![\texttt{*}e]\!](\xi,\sigma) & \triangleq & \mathcal{E}[\![e]\!] \\
\mathcal{L}[\![\ell\,.\,f]\!](\xi,\sigma) & \triangleq & \mathcal{L}[\![\ell]\!](\xi,\sigma) + f \\
\mathcal{L}[\![e_1 \oplus e_2]\!](\xi,\sigma) & \triangleq & \mathcal{L}[\![e_1]\!](\xi,\sigma) \oplus \mathcal{L}[\![e_2]\!](\xi,\sigma)
\end{array}
$$

$$
\frac{\alpha = \mathcal{L}[\![\ell]\!](\xi,\sigma) \quad v = \mathcal{E}[\![e]\!](\xi,\sigma)}{(\xi,\sigma) \to_{\ell\texttt{:=}e} (\xi, \sigma(\alpha)(t) \leftarrow v)}
$$

$$
\frac{\alpha = \mathcal{L}[\![\ell]\!](\xi,\sigma) \quad v = \mathcal{E}[\![e]\!](\xi,\sigma) \quad \beta = \texttt{sbrk}(v)}{(\xi,\sigma) \to_{\ell\texttt{:=sbrk}(e)} (\xi, \sigma(\alpha)(t) \leftarrow \beta)}
$$

Fig. 7. Operational semantics

from these statements. Commands include assignment, data segment extension and guards. An assignment is specified by a location expression ℓ that names a memory address to update and an expression e that is evaluated to yield the new contents for the cell. The type t gives the typing information of the content written at location ℓ and is mainly used to encode C casting. Data segment extension is done using the system method `sbrk`; moreover, it returns the address (of type `void*`) after the last included in the extended data segment. Notice that this address is page aligned, i.e., multiple of 8. Types are either structured types, pointer types or basic types (left unspecified). Structured types are defined by a list of fields. The first field in the list determines the alignment constraints for the addresses at which the values of this type may be stored. Operators include binary arithmetic, bitwise (and, or, shift), and comparison operations. The bitwise operations are used to encode masking or extraction of least/most significant bits. The absence of recursive procedure calls allows us to consider intra-procedural analysis only and use inlining for procedure calls. However, our analysis may be extended to an inter-procedural analysis using, e.g., [5]. Fields are treated as numerical offsets, so C field access $\ell\,.\,f$ is the address $a + f$ where a is the address denoted by location expression ℓ. A typing function τ maps fields and variables to their declaration type; it is extended naturally to location expressions.

3.3. Concrete program semantics

Operational semantics The operational semantics of our programming languag and its two rules e are given in Figure 6. They represent the semantics of two kinds of statements, normal assignment and data segment extension function `sbrk`. The evaluations of locations and expressions are denoted by $\mathcal{L}[\![\cdot]\!]$ and $\mathcal{E}[\![\cdot]\!]$. The function $\mathcal{L}[\![\cdot]\!]$ (resp. $\mathcal{E}[\![\cdot]\!]$) maps a location expression ℓ (resp. expression e) in a context of a given concrete memory state to an address (resp. value). The evaluation of field $\ell.f$ is to evaluate location ℓ then do address computation. The evaluation of deference $\mathcal{E}[\![\texttt{\&}\ell]\!]$ is obtained by evaluation of location ℓ, i.e. $\mathcal{L}[\![\ell]\!]$. And the evaluation of deference $\mathcal{L}[\![\texttt{*}e]\!]$ is obtained by evaluation of expression e, i.e. $\mathcal{E}[\![e]\!]$.

Example. At line 45 of Figure 1(b), for assignment c : `prv->fnx` = `nxt->fnx`, we assume the concrete state of program before line 45 is same as Figure 1(d) shows and is denoted by (ξ, σ_c). The evaluation of this

$$
\begin{array}{lll}
\mathbf{post}[\![c]\!]S \triangleq & \text{match } c \text{ with:} & \\
& \mid \texttt{skip} & \rightarrow \quad S \\
& \mid \ell \texttt{ :=} e & \rightarrow \quad \{(\xi, \mathcal{L}[\![\ell]\!](\xi,\sigma) \leftarrow \mathcal{E}[\![e]\!](\xi,\sigma)) \mid (\xi,\sigma) \in S\} \\
& \mid \ s_1; s_2 & \rightarrow \quad \mathbf{post}[\![c_1]\!](\mathbf{post}[\![c_2]\!]S) \\
& \mid \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 & \rightarrow \quad \mathbf{post}\ [\![c_1]\!]S \cap \mathcal{B}[\![e]\!]) \cup \mathbf{post}\ [\![c_1]\!]S \cap \mathcal{B}[\![\neg e]\!]) \\
& \mid \texttt{while } e \texttt{ do } c \texttt{ done} & \rightarrow \quad \mathbf{lfp}^{\subseteq}(\lambda S_0.S \cup \mathbf{post}[\![c_1]\!]S_0 \cap \mathcal{B}[\![e]\!])) \cap \mathcal{B}[\![\neg e]\!]
\end{array}
$$

Fig. 8. Concrete postconditions

assignment proceeds as follows. We first evaluate the fight-hand side:

$$
\begin{aligned}
\mathcal{E}[\![\texttt{nxt->fnx}]\!](\xi,\sigma_c) &= \sigma(\mathcal{L}[\![\texttt{nxt->fnx}]\!](\xi,\sigma_c)) = \sigma(\mathcal{L}[\![\texttt{nxt}]\!](\xi,\sigma_c) + \texttt{fnx}) \\
&= \sigma(\xi(\texttt{nxt}) + \texttt{fnx}) = \xi(\mathsf{nil})
\end{aligned}
$$

Then, we do location evaluation of left-hand side of the assignment:

$$
\begin{aligned}
\mathcal{L}[\![\texttt{prv->fnx}]\!](\xi,\sigma_c) &= \mathcal{L}[\![\texttt{fnx}]\!](\xi,\sigma_c) + \texttt{fnx} = \xi(\texttt{prv}) + \texttt{fnx} \\
&= X + \texttt{fnx} = Y
\end{aligned}
$$

We assume the memory cell address to which variable `prv` points is X and the address of field `fnx` of `prv` is Y. Then, the value at address Y is updated with the value address of nil. Thus, the green arrow out from X will point to $\mathsf{nil} : \sigma[Y \leftarrow \xi(\mathsf{nil})]$.

Postcondition operators The postcondition operators of program are shown in Figure 8. $\texttt{Post}[\![c]\!]S$ represents the set of states reachable from a set of states S after a program command c. The least-fixed point of $\mathbf{f} : \mathbb{M} \rightarrow \mathbb{M}$ is denoted by **lfp**. For evaluation of condition tests command, the evaluation of boolean expressions was left out since it is standard. $\mathcal{B}[\![e]\!] \in \mathbb{M}$ defines the set of concrete states in which e is true. The effect of transition relation are obtained by the operational semantics.

3.4. Graph representation of memory state

Graph definition For sake of readability, we define graph representation for concrete memory state and define rules for graph transformations below. Each formula of SLH represents a set of memory states has a corresponding graph representation of its spatial formulas. As shown in Figure 1(d), each memory chunk is represented by a graph node. The identifier of a node is the start address of the memory chunk. The edges of the graph have three types. They represent the neighbor relation of heap list (e.g. next chunk, shown in red arrows), the points-to relation of free list (e.g. next free chunk, shown in green arrows) and concrete environment (e.g. shown in black arrows mapping program variables to memory addresses). Edges connecting nodes are labeled with atom predicates. The bold edges represent memory region described by summary predicates, e.g., hls, and the thin edges represent atomic predicates, e.g.. chk.

Definition 1. The shape of memory state is represented by $G = (N, P, L, V)$ where:

- N is a set of nodes which includes the distinguished node nil and hli,
- $P : N \rightharpoonup_* N$ is a partial function mapping node to node and is labeled by an atom predicate,
- $L : N \rightharpoonup \mathbb{A}^+$ is a partial function associating nodes to a sequence of addresses, this sequence is denoted by a sequence variable W,
- $V : \mathbb{X} \rightharpoonup N$ is a partial function associating program variables to nodes.

Transformations rules During the processes of allocation and deallocation, some statements involving pointer arithmetics will create or delete chunks. Thus, the shape of memory will also be updated. We define postcondition operator $\mathbf{post}[\![c]\!](G)$ for heap transformations of command c in Figure 9. Rule $\mathrm{R}_{\mathrm{sbrk}}$ is defined for calling extension function `sbrk` and it specifies that a new node will be added into the grapG. We denoted by $G.N$ the set of nodes of graph G. Rule $\mathrm{R}_{\mathrm{split}}$ is defined for assignment with pointer arithmetics which will split a chunk into two pieces. A new node n is generated at a location inside the chunk n_1. The third rule $\mathrm{R}_{\mathrm{ptr}}$ is used for assignment maping a pointer variable to a location of heap.

$$\frac{n \notin G.N \quad n = \mathtt{sbrk}(\mathcal{E}[\![e]\!]G) \quad L(n_1) = \mathsf{hli}}{\mathbf{post}[\![\ell \ :=\mathtt{sbrk}(e)]\!](G) = \left\langle \begin{array}{l} G.N \cup \{n\}, G.P[n_1 \mapsto n, n \mapsto \mathsf{hli} + n], \\ G.V[\ell \mapsto n], G.L[n \mapsto (\mathsf{hli}..\mathsf{hli} + n)] \end{array} \right\rangle} \ \mathrm{R_{sbrk}}$$

$$\frac{n \notin G.N \quad n = \mathcal{E}[\![e]\!]G \quad n \in L(n_1)}{\mathbf{post}[\![\ell \ :=e]\!](G) = \left\langle \begin{array}{l} G.N \cup \{n\}, G.P[n_1 \mapsto n, n \mapsto n_2], \\ G.L[n_1 \mapsto (n_1..n), n \mapsto (n..n_2)], G.V[\ell \mapsto n] \end{array} \right\rangle} \ \mathrm{R_{split}}$$

$$\frac{n \in G.N \quad n = \mathcal{E}[\![e]\!]G \quad n \in L(n_1)}{\mathbf{post}[\![\ell := e]\!](G) = \langle G.N, G.P, G.L, G.V[\ell \mapsto n] \rangle} \ \mathrm{R_{ptr}}$$

Fig. 9. Postcondition of assignments for graph

3.5. Preliminaries on program analysis

Abstract interpretation The static analysis based on abstract interpretation framework [12] relies on 1) designing an approximate abstract domain or abstraction whose elements are abstract properties, 2) establishing the correspondence between concrete domain and abstract domain, e.g. Galois connections. Given two lattices $C = \langle L_1, \subseteq \rangle$ and $A = \langle L_2, \sqsubseteq \rangle$ ($\subseteq$ resp. $\sqsubseteq$ are order relations on L_1 resp. L_2), we can define a pair of monotone functions (maps) $(\alpha : L_1 \to L_2, \gamma : L_2 \to L_1)$ and the following properties is held:

$$\forall c \in L_1, a \in L_2 \cdot \alpha(c) \sqsubseteq a \iff c \subseteq \gamma(a)$$

A pair of such monotone functions is a *Galois connection*. We call α the *abstraction* function and γ the *concretization* function. Lattice A is an *abstract domain* for C if there exists a Galois connection between C and A, and C is called the *concrete domain* for A. If C and A connected by Galois connection, we denote a set of concrete transformers by $\mathcal{F}_C : L_1 \to L_1$ and a set of abstract transformers by $\mathcal{F}_A^\sharp : L_2 \to L_2$ includes a function $f^\sharp$ for each $f \in \mathcal{F}_C$. The abstract transformers $f^\sharp$ is *sound* if for any $a \in L_2$, the property $f(\gamma(a)) \subseteq \gamma(f^\sharp(a))$ holds. If $\alpha(f(\gamma(a))) = f^\sharp(a)$ then $f^\sharp$ is a best abstraction, for any $a \in L_2$.

Numeric abstract domain For describing different numerical properties of numerical program variables, many and various abstract domains have been separately proposed. The intervals domain introduced in [13] can infer variables bounds. The octagons domain presented in [30] can describe invariants of the form $\pm v_1 \pm v_2 \leqslant c$ (v_1 and v_2 are numerical variables and c is a numeric constant). Intervals and octagons are two restrictions of polyhedra [15]. Each numerical domain focuses on one kind of properties and has its own expressiveness and computational complexity. The open library Apron [23] provides implementations of lots of numerical abstract domains, and it gives a uniform, rich and domain-independent API.

Combine abstract domains To infer complex properties of program, the abstract interpretation framework supplies several ways to obtain a new abstract domain from existing domains by product operators [10]. Sometimes, the communication between component domain is necessary and component domains may depend on each other. For example, [14] introduces reduced product of abstract domains, [35, 8] introduce cofibered domain, and [11] gives open product framework. In this paper, we use cofibered-product to combine a shape domain and a numerical domain.

4. Abstract Domain for Hierarchical Shape Abstraction

In this section, we present abstract domain for hierarchical abstraction. The elements of abstract domain are based on formulas of our logic introduced above. We introduce basic components used in abstract domain and the abstract value. This section also describes the main abstract operations.

4.1. Basic components

Since our logical formulas consist of two main parts, pure part and spatial part, we design domain in a modular way. The abstract domain used in our work is a combination of existing numerical domain and

$$\frac{n \notin G^\sharp.N \quad n = \mathtt{sbrk}(\mathcal{E}[\![e]\!]G^\sharp) \quad L(n_1) = \mathsf{hli}}{\mathbf{post}^\sharp[\![\ell\ \mathtt{:=sbrk}(e)]\!](G^\sharp) = \left\langle \begin{matrix} G^\sharp.N \cup \{n\}, G^\sharp.P[n_1 \mapsto n, n \mapsto \mathsf{hli}+n], \\ G^\sharp.V[\ell \mapsto n], G^\sharp.L[n \mapsto (\mathsf{hli}..\mathsf{hli}+n)] \end{matrix} \right\rangle} \ \mathrm{R}_{\mathrm{sbrk}}$$

$$\frac{n \notin G^\sharp.N \quad n = \mathcal{E}[\![e]\!]G^\sharp \quad n \in L(n_1)}{\mathbf{post}^\sharp[\![\ell\ \mathtt{:=}e]\!](G^\sharp) = \left\langle \begin{matrix} G^\sharp.N \cup \{n\}, G^\sharp.P[n_1 \mapsto n, n \mapsto n_2], \\ G^\sharp.L[n_1 \mapsto (n_1..n), n \mapsto (n..n_2)], G^\sharp.V[\ell \mapsto n] \end{matrix} \right\rangle} \ \mathrm{R}_{\mathrm{split}}$$

$$\frac{n \in G^\sharp.N \quad n = \mathcal{E}[\![e]\!]G^\sharp \quad n \in L(n_1)}{\mathbf{post}^\sharp[\![\ell \mathtt{:=}e]\!](G^\sharp) = \left\langle G^\sharp.N, G^\sharp.P, G^\sharp.L, G^\sharp.V[\ell \mapsto n] \right\rangle} \ \mathrm{R}_{\mathrm{ptr}}$$

Fig. 10. Postcondition of assignments for abstract graph

our shape domain. The numerical domain is corresponding to the pure part of logical formulas. The shape domain is corresponding to spatial formulas and it is a co-fibered domain combining symbolic heap with data words domains.

4.1.1. Numerical abstract domain

In this paper, some relational numerical domain are required to define abstract value. For example, we need relation such as $X[.fnx] - Y[.fnx] = x$ in the pure part of abstract value, therefore the polyhedra domain[15] is used in the analyzer. Thus, our abstract domain is parameterized by numerical domains. The cost of the analysis depends mainly on the cost of the operation in the underlying numerical domain. The numerical abstract domain used in our domain to specify arithmetic constraints is denoted by $A_{\mathcal{N}} = \langle \mathcal{N}, \sqsubseteq^{\mathcal{N}}, \sqcap^{\mathcal{N}}, \sqcup^{\mathcal{N}}, \top^{\mathcal{N}}, \bot^{\mathcal{N}} \rangle$.

4.1.2. Data words domain

The symbolic variable SVar, represents the set of sequence variables (data word variables) register the start addresses of chunks. We denote by $[]$ (resp. $[X]$) the empty word (resp. the word with only one element X). The binary concatenation operator for words is denoted by the dot, e.g. $W_1.W_2$. The data words domain over SVar is denoted by $\mathbb{DW}(\mathsf{SVar})$ and it is the lattice of sets of maps $[\mathsf{SVar} \rightharpoonup \mathbb{V}^+]$. We define domain $\mathcal{D}^\sharp_{\mathbb{W}} = (\mathcal{A}^{\mathbb{W}}, \sqsubseteq^{\mathbb{W}}, \sqcap^{\mathbb{W}}, \sqcup^{\mathbb{W}}, \top^{\mathbb{W}}, \bot^{\mathbb{W}})$ as an abstract domain for $\mathbb{DW}(\mathsf{SVar})$. It is parameterized by numerical abstract domain $A_{\mathcal{N}}$. The elements of $\mathcal{D}^\sharp_{\mathbb{W}}$ include universally quantified formulas with free variables in Var of the form $\forall v.A_G \Rightarrow A_U$. Here, A_U is defined as an object of the numerical abstract domain. A set of transformers for data words domains are defined similarly in [5]. By using data words domain, we can represent sequence constraints on words associated to the chunks.

4.1.3. Extended symbolic heap domain

We define the *extended symbolic heap domain* $\mathcal{H}^\sharp_{\mathbb{S}} = \langle \mathbb{E}^\sharp \times \Sigma, \sqsubseteq^{\mathbb{S}}, \sqcap^{\mathbb{S}}, \sqcup^{\mathbb{S}}, \top^{\mathbb{S}}, \bot^{\mathbb{S}} \rangle$ to specify symbolic shape of heap with its concretisation $\gamma_{\mathbb{S}} : \mathbb{E}^\sharp \times \Sigma \to \mathbb{M}$. The *abstract environment* is denoted by $\mathbb{E}^\sharp$. The set of abstract memory states $\mathbb{M}^\sharp$ is represented by $\mathbb{E}^\sharp \times \Sigma$ and an abstract memory value $m^\sharp$ is a pairs $\langle \epsilon^\sharp, \Sigma \rangle$, consist of a set of mappings and spatial formulas. The abstraction for heap H is represented by the Gaifman graph $G^\sharp$ with 1) edges labeled by spatial atoms in Σ or mappings in $\epsilon^\sharp$ and 2) nodes representing symbolic location. The abstract environment $\mathbb{E}^\sharp$ is a function mapping program variables to nodes of graph $G^\sharp$, i.e., $\mathbb{E}^\sharp = \mathbb{X} \to \mathbb{V}^\sharp$. As shown in Figure 1, Figure 1(c) is abstract state of concrete shape of memory state Figure 1(d). The heap list is abstracted by the summary predicate of Σ. We define the corresponding abstract postcondition transformer $\mathbf{post}^\sharp[\![c]\!](G^\sharp)$ for abstract heap.

4.2. Abstract domain

In this paper, a domain used in our analysis is denoted by $\mathcal{A}^\sharp_{\mathbb{M}} = \langle \mathcal{A}^\sharp, \sqsubseteq, \sqcup \rangle$, where $\sqcup$ is its join operator. It is parameterized by the numerical domain $A_{\mathcal{N}} = \langle \mathcal{N}, \sqsubseteq^{\mathcal{N}}, \sqcup^{\mathcal{N}} \rangle$. Values in $\mathcal{A}^\sharp_{\mathbb{M}}$ are a restricted form of

logic formulas. Generally speaking, $\mathcal{A}^\sharp_{\mathbb{M}} : [\mathcal{H}^\sharp_{\mathbb{S}} \rightrightarrows \mathcal{D}^\sharp_{\mathbb{W}}]$ is a *co-fibered product* [8] of the extended symbolic heap domain ($\mathcal{H}^\sharp_{\mathbb{S}}$) for the spatial part and the data word ($\mathcal{D}^\sharp_{\mathbb{W}}$) domain [5] for the pure part. The numerical constrains and data words constraints used in $\mathcal{D}^\sharp_{\mathbb{W}}$ specify the precise content's information for shape of extend symbolic heap.

4.2.1. Abstract values

More precisely, $\mathcal{A}^\sharp_{\mathbb{M}}$ includes special values for $\top, \bot$ and finite mappings of the form:

$$a^\sharp \ ::= \ \{\langle \epsilon^\sharp_i, \Sigma_i(\overrightarrow{x}, \overrightarrow{W})\rangle \mapsto \Pi_i(\overrightarrow{x}, \overrightarrow{W} \cup \{W_H, W_F\})\}_{i\in I} \tag{2}$$

where $\epsilon^\sharp_i : \mathbb{X} \to \mathsf{Var}$ 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.

4.2.2. Concretisation

An abstract value of the form (2) represents a formula $\vee_i \exists \vec{x}, \vec{W} \cdot \Sigma_i \wedge \Pi_i \wedge \epsilon^\sharp_i$ where each binding $(v, x) \in \epsilon^\sharp_i$ 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 $\gamma_{\mathbb{M}} : \mathcal{A}^\sharp \to \mathbb{M}$ as the denotation of the formula represented by the abstract value, i.e., $\gamma_{\mathbb{M}}(a^\sharp) = [\![a^\sharp]\!]$.

4.3. Lattice operators

4.3.1. Ordering

The partial order $\sqsubseteq$ is defined using a sound procedure inspired by [6, 19]. For any two non trivial abstract values $a^\sharp, b^\sharp \in \mathcal{A}^\sharp$, $a^\sharp \sqsubseteq b^\sharp$ if for each binding $\langle \epsilon^\sharp_i, \Sigma_i\rangle \mapsto \Pi_i \in a^\sharp$ there exists a binding $\langle \epsilon^\sharp_j, \Sigma_j\rangle \mapsto \Pi_j \in b^\sharp$ 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 $\Psi : \mathrm{img}(\epsilon^\sharp_i) \to \mathrm{img}(\epsilon^\sharp_j)$ 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 $[\![a^\sharp]\!] \subseteq [\![b^\sharp]\!]$.

4.3.2. 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 [5]. 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 $a^\sharp, b^\sharp \in \mathcal{A}^\sharp$, $[\![a^\sharp]\!] \cup [\![b^\sharp]\!] \subseteq [\![a^\sharp \sqcup b^\sharp]\!]$.

4.4. Cardinality and widening

4.4.1. Cardinality

The number of mappings in (2) 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. Keeping small the number of addresses used in the shape part of each level is very important for the efficiency of the analysis. Moreover, to some of these addresses are bound feature variables which blow-up with a constant factor the variables of the pure part.

We make the compromise to keep in the shape parts only k addresses which are used in the shape atoms but are not aliased by program variables, also called *anonymous addresses*. k is a parameter of the analysis and it is usually small, e.g., 2 or 3. It especially influences the quality of values in the $\mathcal{D}^\sharp_{\mathbb{W}}$ domain, i.e., the universally quantified formulas. An abstract memory $m^\sharp$ is *k-normalised* if all its conjuncts contains shape graphs with less than k anonymous addresses bound by predicate atoms.

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 (1)–(4), as discussed in Section 5. The application of the lemma follows a heuristics that searches to replace sets of atoms including an address referenced by a program variable and some anonymous addresses by an unique predicate. If a memory value can not be normalised, it is replaced soundly by $\top$. At the H-list level, the failure to fold abstract values may correspond to a memory leak and a warning is issued. Also, the normalisation may lead to an empty pure constraint, in which case the disjunct representing the abstract value is removed. An empty list of disjuncts is equivalent to $\bot$.

4.4.2. Widening

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. 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.

```
1  int main(void) {
2        minit(1024);
3        void* p = malloc(20);
4        malloc(20);
5        mfree(p); p = NULL;
6        p = malloc(20);
7        malloc(20);
8        mfree(p); p = NULL;
9        return 0;
10 }
```

Fig. 11: A client program

It consists of an inter-procedural and non relational *symbolic execution* of a correct client program like the one in Figure 11. 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 11 is built from four disjuncts whose shape part is given in Figure 12.

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 11) are projected out and folding lemmas are applied to obtain list atoms. For example, the transformation of the abstract value in Figure 12 leads to an abstract value with one disjunct whose spatial part is $\mathsf{hlsc}(X_0, 0; \mathsf{hli}, 0) \Subset \mathsf{flso}(X_0, X_0; \mathsf{nil}, \mathsf{hli})$. 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.

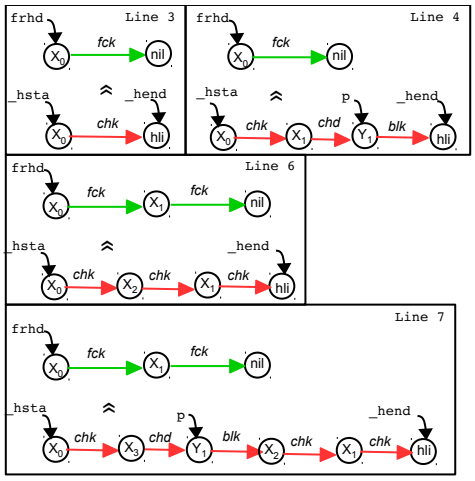


Fig. 12: Spatial formulas at line 28

The third step does forward, non-relational abstract interpretation [12] starting from the computed pre-conditions of each DMA method. The analysis follows the principles of [16, 9, 17] 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 5.3–5.4. Furthermore, these operations are defined in a modular way, by extending [8] to data word numerical domains. The widening operator uses the widening of data word domain defined in [17].

$$
\begin{array}{lcl}
\mathbf{post}^\sharp[\![\texttt{skip}]\!](\xi,h)^\sharp & \triangleq & (\xi,h)^\sharp \\
\mathbf{post}^\sharp[\![\ell \ :=_t \ e]\!](\xi,h)^\sharp & \triangleq & \bigcup_{(\xi',h')^\sharp \in (\xi,h)^\sharp} \{\xi', \mathbf{post}^\sharp[\![\ell \ :=_t \ e]\!](\xi',h')\} \\
\mathbf{post}^\sharp[\![s_1; s_2]\!](\xi,h)^\sharp & \triangleq & \mathbf{post}^\sharp[\![c_1]\!](\mathbf{post}^\sharp[\![c_2]\!](\xi,h)^\sharp) \\
\mathbf{post}^\sharp[\![\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2]\!](\xi,h)^\sharp & \triangleq & \mathbf{post}^\sharp\ [\![c_1]\!]((\xi,h)^\sharp \sqcap \mathcal{B}[\![b]\!]) \sqcup \mathbf{post}^\sharp\ [\![c_1]\!]((\xi,h)^\sharp \sqcap \mathcal{B}[\![\neg b]\!]) \\
\mathbf{post}^\sharp[\![\texttt{while } e \texttt{ do } c \texttt{ done}]\!](\xi,h)^\sharp & \triangleq & \mathbf{lfp}^{\subseteq}(\lambda(\xi,h)^\sharp_0.(\xi,h)^\sharp \sqcup \mathbf{post}^\sharp[\![c_1]\!]((\xi,h)^\sharp_0 \sqcap \mathcal{B}[\![b]\!])) \sqcap \mathcal{B}[\![\neg b]\!]
\end{array}
$$

Fig. 13. Abstract postconditions.[Todo:to review the table]

5.2. Abstract transfers functions

5.2.1. Abstract transformers

Assignments. [TODO: to explain the analysis of assignments]

We define sound abstract transformers for statements in the class of programs we consider. Figure 13 presents how program commands are interpreted using our hierarchical abstract domain to yield the over-approximation of the set of reachable states. The semantics of command c is given by a continuous function $[\![c]\!] : \mathcal{A}^\sharp \to \mathcal{A}^\sharp$. We define the function $\texttt{assign}_{\mathbb{M}} : \mathcal{A}^\sharp(\ell, e) \to \mathcal{A}^\sharp$ which computes sound over-approximated abstract value for assignment ℓ := e whose left hand side is a location expression. The content of memory pointed by ℓ should be updated to evaluated value of e. The transformer is performed by using evaluation of location expression and expression on abstract memory state $m^\sharp$.

[TODO: to add an example with pre-condition and post-condition for an assignment.]

Theorem 3 (assign soundness). $\forall\ m \in \gamma_{\mathbb{M}}(m^\sharp).\ \mathbf{post}[\![\ell \ := e]\!](m) \in \gamma_{\mathbb{M}}(\texttt{assign}_{\mathbb{M}}(m^\sharp(\ell, e))).$

Guards. [TODO: to explain the meet $\sqcap$ operation with booleans]

5.3. Hierarchical unfolding

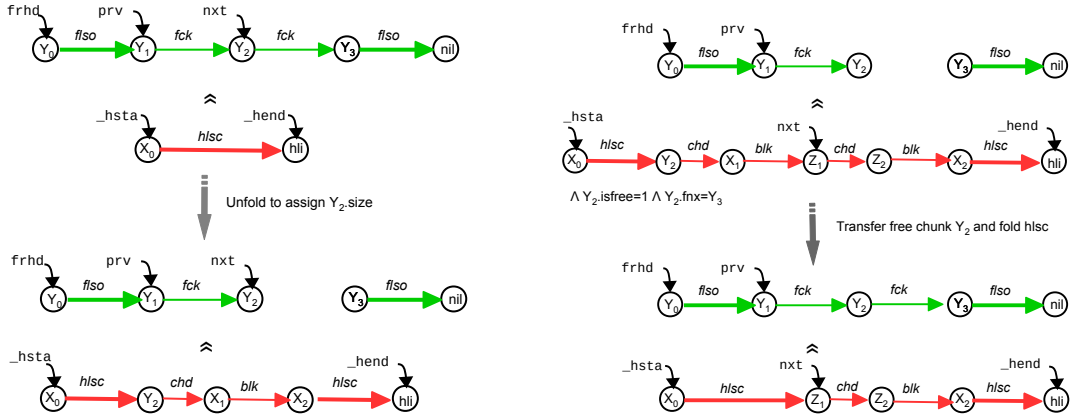
Abstract transformers compute over-approximations of post-images of atomic conditions and assignments in the program. For the spatial part, the abstract value is transformed such that the program variables read or written by the program operation are constrained using predicates that may capture the 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 2 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 14(a). 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 14(a). 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.

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 $Z_1 = Y_2 + Y_2$.size $\times$ 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 14), the analysis signals a chunk breaking. Otherwise, the blk atom from X_1 is split using lemma (3) to insert Z_1; the result is given in the top part of Figure 14(b).



(a) Hierarchical unfolding at line 38 (b) Hierarchical folding after line 48

Fig. 14. Hierarchical unfolding and folding

5.4. 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 (1) or (4) 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 $X_0\texttt{.size} \times \texttt{sizeof(HDR)} = X_n - X_0$ (see Table **??**). 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 $X_0\texttt{.isfree} = 1$, then the chunk atom (and its start address) is promoted as fck to the free list level.

This step is illustrated on sub-formulas $\mathsf{chd}(Y_2; X_1) * \mathsf{blk}(X_1; Z_1)$ at the top of Figure 14(b). The second step folds hlsc list segments by applying their inductive definition and the composition lemma (1). 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 $\ni$. For example, the chunk starting from location Y_2 is folded inside a hlsc segment in the formula at the bottom of Figure 14(b). Notice that folding of list segments implies the update of sequence and universal constraints like in [17].

6. Experiments

We implemented the abstract domain and the analysis algorithm in Ocaml as a plug-in of the Frama-C platform [25]. 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 [23]. 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 15. (Detailed experimental results are available in [1].) DKff and DKbf are implementations of Algorithms A and B from Section 2.5 of [26]. 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 [26]). This property is expressed by the following sub-formula of the invariant "`MIN_SIZE`-size" (for `MIN_SIZE`=8):

$$\forall X \in W_H \cdot X\texttt{.size} \geq 8 \tag{3}$$

which is inferred by our analysis. The first-fit policy is implied by an abstract value similar to the one in

DMA		*LOC*	*List Pred.*	*Time (s)*	$\lvert a^\sharp \rvert$	$\lvert W_H \rvert / \lvert W_F \rvert$	*Invariants*
DKFF	[26]	176	hlsc, flso	0.05	25	8/5	first-fit, `MIN_SIZE`-size
DKBF	[26]	130	hlsc, flso	0.05	26	8/6	best-fit, `MIN_SIZE`-size
LA	[2]	181	hlsc, flso	0.07	25	8/5	first-fit, 0-size
DKNF	[26]	137	hlsc, flso	0.05	30	8/6	first-fit, `MIN_SIZE`-size
KR	[24]	284	hlsc, flso	2.8	32	8/6	first-fit, 0-size

Fig. 15. Benchmark of DMA

equation (1) (page 7). The best-fit policy is implied by a value using the constraint:

$$\forall X \in W_i \cdot X\texttt{.size} \geq \texttt{rsz} \Rightarrow X\texttt{.size} \geq Y\texttt{.size} \tag{4}$$

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., $\forall X \in W_H \cdot X\texttt{.size} \geq 4$ (=`sizeof(HDR)`). Notice that the code analysed fixes an obvious problem of the `malloc` method published in [2]. DKNF implements the next-fit policy using the "roving pointer" technique proposed in [26]: 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 [24] 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 15 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 11. The universally quantified invariants inferred for DMA policies are given in the last column. Columns $\lvert a^\sharp \rvert$ and $\lvert W_H \rvert / \lvert W_F \rvert$ 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 [7, 5, 17, 28, 18]. We consider the SL fragment proposed in [7] 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 [28, 18]. 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 [28]. Another hierarchical analysis of shape and numeric properties has been proposed in [34]. 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 [5, 17], since the universal constraints quantify only one position in the list. Several abstract domains have been defined to infer invariants over arrays, e.g., [20] for array sizes, [21, 22] 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, [29] 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 [8]. We extend their proposal to combine shape domains with domains on sequences of integers. Precise analyses exist for low level code in C [31] or for binary code [3]. They efficiently track properties about pointer alignment and memory region separations, but can not infer shape properties.

References

[1] Celia extensions. http://www.irif.fr/~sighirea/celia/plus.html.

[2] Leslie Aldridge. Memory allocation in C. *Embedded Systems Programming*, pages 35–42, August 2008.

[3] Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In *SAS*, volume 4134 of *LNCS*, pages 221–239. Springer, 2006.

[4] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, volume 3328, pages 97–109. Springer, 2005.

[5] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI*, pages 578–589. ACM, 2011.

[6] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In *ATVA*, volume 7561 of *LNCS*, pages 167–182. Springer, 2012.

[7] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, volume 4134 of *LNCS*, pages 182–203. Springer, 2006.

[8] Bor-Yuh Evan Chang and Xavier Rival. Modular construction of shape-numeric analyzers. In *Semantics, Abstract Interpretation, and Reasoning about Programs*, volume 129 of *EPTCS*, pages 161–185, 2013.

[9] Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In *SAS*, volume 4634 of *LNCS*, pages 384–401. Springer, 2007.

[10] Agostino Cortesi, Giulia Costantini, and Pietro Ferrara. A survey on product operators in abstract interpretation. *arXiv preprint arXiv:1309.5146*, 2013.

[11] Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. Combinations of abstract domains for logic programming: Open product and generic pattern construction. *Science of Computer Programming*, 38(1-3):27–71, 2000.

[12] Patrick Cousot and Radhia 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.

[13] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *ACM SIGPLAN Notices*, volume 12, pages 77–94. ACM, 1977.

[14] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282. ACM, 1979.

[15] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.

[16] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920, pages 287–302. Springer, 2006.

[17] Cezara Dragoi. *Automated verification of heap-manipulating programs with infinite data.* PhD thesis, University Paris Diderot, 2011.

[18] Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Local shape analysis for overlaid data structures. In *SAS*, volume 7935 of *LNCS*, pages 150–171. Springer, 2013.

[19] Constantin Enea, Mihaela Sighireanu, and Zhilin Wu. On automated lemma generation for separation logic with inductive definitions. In *ATVA*, LNCS, pages 80–96. Springer, 2015.

[20] S. Gulwani, T. Lev-Ami, and S. Sagiv. A combination framework for tracking partition sizes. In *POPL*, pages 239–251. ACM, 2009.

[21] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246. ACM, 2008.

[22] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348. ACM, 2008.

[23] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[24] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition.* Prentice-Hall, 1988.

[25] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *FAC*, 27(3):573–609, 2015.

[26] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition.* Addison-Wesley, 1973.

[27] Doug Lea. `dlmalloc`. ftp://gee.cs.oswego.edu/pub/misc/malloc.c, 2012.

[28] Oukseh Lee, Hongseok Yang, and Rasmus Petersen. Program analysis for overlaid data structures. In *CAV*, volume 6806 of *LNCS*, pages 592–608. Springer, 2011.

[29] Jiangchao Liu and Xavier Rival. Abstraction of arrays based on non contiguous partitions. In *VMCAI*, volume 8931 of *LNCS*, pages 282–299. Springer, 2015.

[30] Antoine Miné. The octagon abstract domain. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 310–319. IEEE, 2001.

[31] Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES*, pages 54–63. ACM, 2006.
[32] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, LNCS, pages 1–19. Springer, 2001.
[33] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
[34] Pascal Sotin and Xavier Rival. Hierarchical shape abstraction of dynamic structures in static blocks. In *APLAS*, volume 7705 of *LNCS*, pages 131–147. Springer, 2012.
[35] Arnaud Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *SAS*, volume 1145 of *LNCS*, pages 366–382. Springer, 1996.
[36] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *IWMM*, volume 986 of *LNCS*, pages 1–116. Springer, 1995.