Lightweight Shape Analysis based on Physical Types

Olivier Nicole, Matthieu Lemerre, Xavier Rival

Lightweight Shape Analysis based on Physical Types

Olivier Nicole[1 2 3], Matthieu Lemerre[1], and Xavier Rival[2 3]

[1] Software Safety and Security Laboratory, CEA List, France
[2] Département d'informatique de l'ENS, CNRS, PSL University, France
[3] Inria, France

Abstract. To understand and detect possible errors in programs manipulating memory, static analyses of various levels of precision have been introduced, yet it remains hard to capture both information about the byte-level layout and precise global structural invariants. Classical pointer analyses struggle with the latter, whereas advanced shape analyses incur a higher computational cost. In this paper, we propose a new memory analysis by abstract interpretation that summarizes the heap by means of a type invariant, using a novel kind of *physical types*, which express the byte-level layout of values in memory. In terms of precision and expressiveness, our abstraction aims at a middle point between typical pointer analyses and shape analyses, hence the *lightweight shape analysis* name. We pair this summarizing abstraction with a *retained and staged points-to predicates* abstraction which refines information about the memory regions that are in use, hereby allowing *strong updates* without introducing disjunctions. We show that this combination of abstractions suffices to verify spatial memory safety and non-trivial structural invariants in the presence of low-level constructs such as pointer arithmetic and dynamic memory allocation, on both C and binary code.

1 Introduction

Memory errors have long been a very important concern for programmers, due to the potential safety and security issues that they raise. In particular, programs that perform low-level pointer and memory operations are particularly tedious to reason about in languages like C/C++ or assembly. For instance, such patterns are very common in system software, which makes its correct implementation challenging.

Many verification techniques aimed at verifying the correctness of memory manipulating programs have been developed. In particular, several families of automatic and conservative static analysis focus on such errors. Pointer analyses [35] based on abstractions of aliasing relations [1] or access paths [10] infer basic conservative relations between pointer values and can tackle basic memory errors. However, they are of limited expressiveness, which implies they cannot establish safety when doing so requires reasoning over structural invariance. On the other hand, shape analyses based on three valued logics like TVLA [33] or on separation logics [31] such as Infer [11] or Xisa [6] attempt to establish precise structural

invariants such as the existence of some list or tree data-structures. Such analyses can cope with the verification of memory safety in presence of sophisticated structures, yet they are typically less scalable than basic pointer analyses and also less resilient to a local precision loss in the sense that losing precision over a fragment of the memory often entails no information can be recovered about that region. Another limitation is that such analyses are difficult to apply to low-level code, like low-level C or binary code, even though some abstractions have been adapted to deal with some forms of pointer arithmetics [18,20]. Few analyses have been aimed for a precision level that sits in between those two large classes, like graph heap models [27], but these do not cope with a low-level memory description.

In this paper, we are interested in memory abstractions expressive enough to verify *type safety*, i.e. the *preservation of structural invariants expressed by types*, in non-trivial linked data-structure manipulations in both high- and low-level code (such as assembly or low-level C). This type safety entails *spatial memory safety*, namely that each memory access is done on an address that was previously allocated (and thus that null or out-of-bound pointer dereferences are impossible). We also seek for a high level of automation (i.e., by avoiding the requirement of complex handwritten program annotations) and of efficiency.

To achieve this, we propose a novel memory abstraction that is inspired by the classical notion of types, but applies to the physical representation of data-structures (Section 4). Our abstract domain (Section 5) represents the heap in a flow-insensitive way, which is less expressive than shape analyses, but allows a simpler representation of abstract states and simpler, more efficient analysis operations (Section 6). Combined with two independent extensions of the domain to track "retained" and "staged" points-to predicates (Section 7), we show that the combination naturally deals with both C and binary code manipulating dynamic data structures (Section 8).

2 Overview example

We demonstrate the main features of our analysis on a low-level implementation of a classical union-find structure inspired by Kennedy [17]. The representation combines the union-find structure based on chains of pointers to class representatives in reverse tree shapes with doubly linked-lists for efficient iteration over the elements of an equivalence class. The whole code is presented in Figure 1. It is written in C for the sake of readability, but we are interested in analysis techniques that would also cope with the corresponding assembly code just as well. Structures `uf` and `dll` respectively represent the union find and doubly linked list structures. Following a pattern common in low-level and system code [3], the structure `node` comprises both sub-structures `uf` and `dll`. Function `uf_find` returns the representative of the class of an element and halves [36] the paths to the root to speed up subsequent calls. Functions `dll_union` and `uf_union` respectively merge doubly linked-lists and union-finds. Last, `merge` merges two `node` structures and `make` creates a new node.

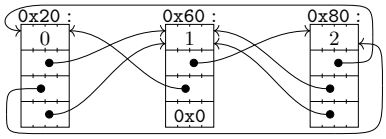
Figure 2(a) displays an example concrete state, with a class made of three nodes (and where the node at address `0x60` is the representative). Such states contain a

```
1  typedef struct uf {
2    struct uf* parent;
3  } uf;
4  typedef struct dll {
5    struct dll *prev; /* != null. */
6    struct dll *next; /* != null. */
7  } dll;
8  typedef unsigned int node_kind;
9  typedef struct node {
10   node_kind kind; /* kind <= 5. */
11   struct dll dll;
12   struct uf uf;
13 } node;
14 uf *uf_find(uf *x) {
15   while(x->parent != 0) {
16     uf *parent = x->parent;
17     if(parent->parent == 0)
18       return parent;
19     x->parent = parent->parent;
20     x = parent->parent;
21   }
22   return x;
23 }
24 void dll_union(dll *x, dll *y) {
25   y->prev->next = x->next;
26   x->next->prev = y->prev;
27   x->next = y; y->prev = x;
28 }
29 void uf_union(uf *x, uf *y) {
30   uf *rootx = uf_find(x);
31   uf *rooty = uf_find(y);
32   if(rootx != rooty)
33     rootx->parent = rooty;
34 }
35 void merge(node *x, node *y) {
36   dll_union(&x->dll, &y->dll);
37   uf_union(&x->uf, &y->uf);
38 }
39 node *make(node_kind kind) {
40   node *n = malloc(sizeof(node));
41   n->kind = kind;
42   n->dll.next = &n->dll;
43   n->dll.prev = &n->dll;
44   n->uf.parent = NULL;
45   return n;
46 }
```

Fig. 1. An algorithm for union-find and listing elements in a partition.

very high degree of sharing due to the interleaved union-find and doubly-linked list structures. Moreover, these structures are unbounded. Therefore, pointer analysis techniques would require tricky and ad hoc adaptations regarding sensitivity to be precise, so as to divide heaps in regions of pointers with similar properties; these techniques are too imprecise to verify type or memory safety for C or assembly. In the same time, shared data structures such as union-find are notoriously hard to handle for shape analysis abstractions and we are not aware of any successful shape analysis based verification for a structure similar to that of Figure 1.

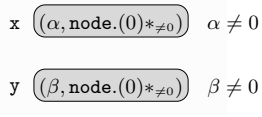
Our key contribution is to propose an abstract interpretation framework based on a semantic interpretation of physical types, that simultaneously verifies the preservation of type-based structural invariants, and uses these invariants to perform and improve the precision of the analysis. This contrasts with the usual method where syntactic type checking and type-based pointer analyses are separate analyses, each insufficiently precise to verify type safety for low-level languages like C or binary. The type-based structural invariant implies dividing the heap into partitions, which are attached flow-insensitive information, allowing for efficient static analysis operations. To further improve precision, our analysis is strengthened by flow-sensitive points-to predicates, whose effect is comparable to materialization in shape analysis, but the memory summary is provided by the type-based structural invariants. In this section, we informally present the basic predicates of our analysis.



(a) Concrete state.

$$\begin{aligned}\mathcal{M} : \texttt{node\_kind} &\mapsto \{x : \texttt{word}_4 \mid x \leq 5\}\\ \texttt{uf} &\mapsto \texttt{uf}.(0)*\\ \texttt{dll} &\mapsto \texttt{dll}.(0)*_{\neq 0} \times \texttt{dll}.(0)*_{\neq 0}\\ \texttt{node} &\mapsto \texttt{node\_kind} \times \texttt{dll} \times \texttt{uf}\end{aligned}$$

(b) Physical types.



(c) Analysis of `merge`.

$\forall \nu, \forall (\sigma, h, \mathcal{L}, \Gamma)$ well-typed state, $\forall v$ value :

$$v \in (\!|\texttt{node}.(0)*_{\neq 0}|\!)_{\mathcal{L},\nu} \implies v+4 \in (\!|\texttt{node}.(4)*_{\neq 0}|\!)_{\mathcal{L},\nu} \quad (1)$$

$$(\!|\texttt{node}.(4)*_{\neq 0}|\!)_{\mathcal{L},\nu} \subseteq (\!|\texttt{dll}.(0)*_{\neq 0}|\!)_{\mathcal{L},\nu} \quad (2)$$

$$v \in (\!|\texttt{dll}.(0)*_{\neq 0}|\!)_{\mathcal{L},\nu} \implies h[v..v+4] \in (\!|\texttt{dll}.(0)*_{\neq 0}|\!)_{\mathcal{L},\nu} \quad (3)$$

$$(\!|\texttt{uf}.(0)*_{\neq 0}|\!)_{\mathcal{L},\nu} \cap (\!|\texttt{dll}.(0)*_{\neq 0}|\!)_{\mathcal{L},\nu} = \emptyset \quad (4)$$

(d) Some structural invariants entailed by $\mathcal{M}$

Fig. 2. Concrete and abstract states based on physical types.

Let us examine the types and structural invariants on our example code. The types are given in Figure 2(b). They must be provided by the analysis user, and possibly derived in part from the C types, although they express stronger invariants. Note that our analysis is independent from C typing rules; in particular C is not type-safe, while we can verify type-safety on both C and compiled programs. Intuitively, $\texttt{dll}.(0)*_{\neq 0}$ denotes a non-null pointer to the base address of another, well-formed `dll` instance. In the case of `uf`, the parent pointer may be null, hence the subscript $*_{\neq 0}$ is absent. Finally, type `node_kind` is a type refined with a predicate restraining its possible values: it corresponds to 4-byte bit vectors whose unsigned value is lesser than 5. Thus, these types can be more precise than C types, although C types can be translated to our type language. But they are less precise than shape invariants, as they cannot represent the relation between different elements of a same type: our `dll` structure could e.g. represent a binary tree with leaves pointing to the root.

These types entail structural invariants, some of which are presented in Fig. 2(d), that a well-typed state must fulfill. These invariants relate types, interpreted as sets of values: $(\!|t|\!)_{\mathcal{L},\nu}$ represents the set of values for type t. Equation (1) relates adjacent addresses; Equation (2) describes a subtyping relationship; Equation (3) relates the type of an address with its contents; and Equation (4) describes a partitioning of the heap in distinct regions. Note that the correctness of these invariants implies that the memory layout of the heap must be compatible with these equations (as is Figure 2(a)), which is why the interpretation of types depends on the heap layout $\mathcal{L}$.

We now turn our attention to our abstract domain. The abstract state shown in Figure 2(c) represents the initial state when execution of the `merge` function begins (this function requires that it is given non-null pointers to `node` as arguments). Each variable is associated to both an abstract type describing possible values stored in the variable, and to a symbolic variable used to attach numerical constraints to this value. For instance, variable `x` is bound to physical type $\texttt{node}.(0)*_{\neq 0}$, meaning that its value belongs to $(\!|\texttt{node}.(0)*_{\neq 0}|\!)_{\mathcal{L},\nu}$; furthermore it is bound to symbolic variable α which is constrained to be not null. Combined with structural invariants of Equations (1),(2) and (3), we can verify that `x+4` (the low-level counterpart of

Fig. 3. Abstract state before line 19.

`&x->dll`) points to a valid address, that can be safely casted as type $\texttt{dll}.(0)*_{\neq 0}$, and that reading from this address will return a value that also has type $\texttt{dll}.(0)*_{\neq 0}$. Eventually, using these invariants we can verify that all memory accesses performed by the call to `dll_union` are valid, but also that each statement preserves these structural invariants.

However, this approach does not suffice when considering more complex functions, like `uf_find`. First, we remark that the function may run correctly only when argument `x` is non-null due to the dereference at line 15, although the `uf` physical type does not require pointers to parent be non-null. Therefore, the verification of this function will use semantic information coming from the numerical abstract domain. Next, we observe that to prove the validity of the access to `parent->parent` at line 17, the analysis needs to establish that `parent` is non-null, by observing that it is equal to `x->parent`, which is non-null due to the condition at line 15. Such reasoning cannot be performed solely using a combination of types and numerical predicates, because the type-based invariants cannot attach different information to different heap objects of the same type. Therefore, we augment variable-type predicates with additional boxes, also defined with a symbolic variable and a physical type, but that corresponds to some selected heap addresses. Only boxes that are reachable from a variable finite chain of points-to predicates may be retained this way. Figure 3 shows the abstract state at line 19 that enables to proves the `parent->parent` access. In the following, we call such predicates *retained points-to predicates*. Such predicates are obtained by retaining information about recent memory writes, loads, or condition tests and need to be abstracted away as soon as they cannot be proved to be preserved. Indeed, when the analysis encounters a memory write, it drops all such boxes for which the absence of aliasing cannot be established with the current information; some aliasing information (e.g. Equation(4)) comes from the partitioning of the heap. This process will be referred to as *blurring* as it carries some similarity with the blurring encountered in some shape analyses. Note that the retained points-to predicates offer a very lightweight way to keep some memory cells represented precisely without resorting to unfolding/focusing which is generally more costly (but also more powerful in the logical point of view), as retaining a heap address or blurring it does not require modifying the summarized heap representation. Physical types coupled with retained points-to predicates allow to verify memory safety and typing preservation for the four functions `dll_union`, `uf_find`, `uf_union` and `merge`.

Finally, we consider function `make`. For the sake of simplicity, we assume that `malloc` always returns a non-null pointer. We note that variable `n` does not point to a valid node object until the very end of the function, thus attempting to prove it satisfies physical type $\texttt{node}.(0)*_{\neq 0}$ before that point will fail. In general, some code patterns like memory allocation or byte-per-byte copy temporarily do not preserve the structural invariants described by our types. To alleviate this, we augment our

$$
\begin{array}{lll}
\mathit{stmt} ::= x := \mathit{expr} & (x \in \mathbb{X}) & \\
\quad | \quad *_\ell \mathit{expr} := \mathit{expr} & (\ell \in \mathbb{N}) & \\
\quad | \quad x := \mathbf{malloc}_t(\mathit{expr}) & (x \in \mathbb{X}, t \in \mathbb{T}) & \\
\quad | \quad \mathbf{skip} \quad | \quad \mathit{stmt}; \mathit{stmt} & & \\
\quad | \quad \mathbf{if}\ \mathit{expr}\ \mathbf{then}\ \mathit{stmt}\ \mathbf{else}\ \mathit{stmt}\ \mathbf{end} & & \\
\quad | \quad \mathbf{while}\ \mathit{expr}\ \mathbf{do}\ \mathit{stmt}\ \mathbf{done} & &
\end{array}
\qquad
\begin{array}{lll}
\mathit{expr} ::= c & (c \in \mathbb{V}) & \\
\quad | \quad x & (x \in \mathbb{X}) & \\
\quad | \quad \mathit{expr} \diamond \mathit{expr} & & \\
\quad\quad (\diamond \in \{+, -, \times, /, \leq, <, & & \\
\quad\quad\quad =, \neq, \&, |, \cdots\}) & & \\
\quad | \quad *_\ell \mathit{expr} & (\ell \in \mathbb{N}) &
\end{array}
$$

Fig. 4. Language WHILE$_{\text{MEM}}$

abstraction with a notion of *staged points-to predicates* that represent precisely the effect of sequences of store instructions such as the body of `make`, allowing to delay their abstraction into types at a later point.

The abstractions sketched so far may also be applied to binary code provided type information can be recovered from, e.g., debugging information. In the rest of the paper, we describe more precisely physical types in Section 4 whereas retained points-to predicates and buffered write predicates are formalized in Section 7.

3 Language and semantics

Although our analysis was implemented both for C and binary code, we adopt a simple imperative language for the sake of presentation. As the grammar in Figure 4 shows, WHILE$_{\text{MEM}}$ features basic assignments, usual arithmetic expressions, memory allocation, and standard control flow commands. Memory locations include a finite set of variables $\mathbb{X}$ and addresses $\mathbb{A}$ that can be computed using usual pointer arithmetic operations. The analysis is parameterized by the choice of an *application binary interface* (or ABI) that fixes endianness, basic types sizes and alignments. In the following, we assume a little-endian ABI is fixed and let $\mathcal{W}$ denote the size of words. Memory access patterns of C can be translated into WHILE$_{\text{MEM}}$; for instance, assuming a pointer size of 4 bytes, `x->prev` turns into $*_4(\mathtt{x} + 4)$. We leave out functions, that our analysis handles in a context sensitive manner. We assume that instances of **malloc** are marked with a type t, though we define the set of types in Section 4.

The values manipulated by WHILE$_{\text{MEM}}$ are bit vectors, i.e., non-negative integers as fixed-size sequences of bytes, so the set of values $\mathbb{V}$ is defined by:

$$\mathbb{V} = \{(\ell, v) \mid \ell \in \mathbb{N}, v \in [0, 2^{8\ell} - 1]\}$$

If $n > 0$, we let $\mathbb{V}_n$ denote the set of bit vectors of length n. We extend the binary operator notation to bit vectors of the same byte length, i.e., $(\ell, v_1) \diamond (\ell, v_2)$ means $(\ell,\ v_1 \diamond v_2)$. The concatenation of any two bit vectors x and y is denoted $x :: y$ and is defined by $(\ell_1, v_1) :: (\ell_2, v_2) = (l_1 + l_2,\ v_1 + 2^{8\ell_1} v_2)$. The set of addresses $\mathbb{A}$ is a subset of $\mathbb{V}_{\mathcal{W}}$. As usual, we let stores map variables to their contents (thus, $\mathbb{\Sigma} = \mathbb{X} \to \mathbb{V}$) and heaps be partial functions from addresses to their contents ($\mathbb{H} = \mathbb{A} \rightharpoonup \mathbb{V}_1$). Moreover, the set of states is $\mathbb{S} = \mathbb{\Sigma} \times \mathbb{H}$.

Given a heap $h \in \mathbb{H}$, $a \in \mathbb{A}$, and $\ell \in \mathbb{N}$, we let $h[a..a{+}\ell]$ denote the reading of a cell of size ℓ at address a. It is defined by $h[a..a{+}\ell] = h(a) :: h(a{+}1) :: \cdots :: h(a{+}\ell{-}1)$.

We denote by $\sigma[x \leftarrow v]$ the store σ with x now mapped to v, and by $h[a..a+\ell \leftarrow v]$ the heap h with values at addresses a (included) to $a+\ell$ (excluded) replaced with the bytes from v. Finally, dropping a range of mappings from a heap is noted $h[a..a+\ell \leftarrow \bot]$.

The semantics of the language is given by a transition relation $\rightarrow \in (\mathit{stmt} \times \mathbb{S}) \times (\mathit{stmt} \times \mathbb{S})$ whose definition is standard (and given in the appendices of the paper [30]). We let Ω denote the state after a run-time error (such as division by zero or null pointer dereference), and $\mathcal{E}[\![e]\!] : \mathbb{S} \rightarrow \mathbb{V} \times \{\Omega\}$ denote expression evaluation. Last, to express the soundness of the analysis, we define a *collecting semantics* as follows. Given a program p, the semantics $[\![p]\!] : \mathcal{P}(\mathbb{S}) \rightarrow \mathcal{P}(\mathbb{S})$ maps a set of input states into a set of output states and is such that $(\sigma', h') \in [\![p]\!](S)$ if and only if there exists $(\sigma, h) \in S$, and a sequence of transitions $(p, (\sigma, h)) \rightarrow (p_1, (\sigma_1, h_1)) \rightarrow \ldots \rightarrow (p_n, (\sigma_n, h_n)) \rightarrow (\mathbf{skip}, (\sigma', h'))$.

4 Physical representation types

In this section, we formalize *physical representation types* (or, for short, *physical types*) and a typed semantics, that serve as a basis for our analysis. The core idea here is to define a notion of well-typed state which will be used as the base invariant representing the summarized regions of memory.

Definition. As shown in Section 2, physical representation types are aimed at describing the memory layout of memory regions using predicates inspired by the standard types, but extended with additional properties. Therefore, the set of physical types comprise standard types for the representation of not only base values, but also structures and arrays. Moreover, they attach to each pointer variable not only the type of the structure that is pointed but also the offset in the block and information about the possible nullness of the pointer.

In order to describe additional constraints such as array indexes, physical types may be refined [15,32] with numerical constraints, that may bind not only the corresponding value, but also existentially quantified symbolic variables (representing e.g. the unknown size of an array). To this effect, we let $\mathbb{V}^\sharp = \{\alpha_0, \alpha_1, \ldots\}$ denote a countable set of *symbolic variables*. Moreover, the concretization of types needs to reason over the actual value of symbolic variables. Such a realization of symbolic variables to values is called a *valuation* and is usually noted $\nu : \mathbb{V}^\sharp \rightarrow \mathbb{V}$.

Finally, the analysis is parameterized by a fixed set of *type names* $\mathcal{N}$, and a mapping $\mathcal{M} \in \mathcal{N} \rightarrow \mathbb{T}$ binding type names to types. Type names have two uses: first they break cycles in the definition of recursive types; second they distinguish types otherwise structurally equal (i.e. it allows the type system to be nominal), and in particular pointers to two structurally equal types with different names will not alias. For instance, in Section 2, we considered recursive types `dll` and `uf`, and Figure 2(b) gives an example of a mapping $\mathcal{M}$.

Definition 1 (Physical representation types). *The set* $\mathbb{T}$ *of* physical representation types *is defined by the grammar in Figure 5.*

$$
\begin{array}{rcll}
\mathit{predexpr}(x) & ::= & x & \text{(constrained type variable)} \\
& | & c \quad \text{(constant } c \in \mathbb{V}\text{)} \quad | \quad \alpha & \text{(symbolic variable } \alpha \in \mathbb{V}^\sharp\text{)} \\
& | & \mathit{predexpr}(x) \diamond \mathit{predexpr}(x) & \text{(binary op., } \diamond \in \{+, -, \times, \cdots\}\text{)} \\
\mathit{pred}(x) & ::= & \mathit{predexpr}(x) \bowtie \mathit{predexpr}(x) & \text{(comparison, } \bowtie \in \{\leq, <, =, \neq\}\text{)} \\
& | & \neg \mathit{pred}(x) \quad | \quad \mathit{pred}(x) \wedge \mathit{pred}(x) & \\
\mathbb{T} \ni t & ::= & \texttt{word}_n & \text{(base type of size } n \text{ bytes)} \\
& | & \texttt{n} & \text{(named type with type name } \texttt{n} \in \mathcal{N}\text{)} \\
& | & t_a* & \text{(possibly null pointer)} \\
& | & t_a*_{\neq 0} & \text{(non-null pointer)} \\
& | & t \times t & \text{(product type)} \\
& | & \{x : t \mid \mathit{pred}(x)\} & \text{(type with a refinement predicate)} \\
& | & t[s] & \text{(array type, } s \in \mathbb{N} \cup \mathbb{V}^\sharp\text{)} \\
\mathbb{T}_A \ni t_a & ::= & t.(k) & \text{(address type with offset, } k \in \mathbb{N}\text{)}
\end{array}
$$

Fig. 5. Definition of physical representation types.

Note that, a type refined by a predicate makes use of a local variable x that denotes the value of this type and is meant to be constrained in the matching $\mathit{pred}(x)$ predicate, which is why grammar entries $\mathit{predexpr}(x)$ and $\mathit{pred}(x)$ take a variable as parameter. An *address type* $t.(k) \in \mathbb{T}_A$ represents the k-th byte in a value of type t. Finally, the pointer types t_a* and $t_a*_{\neq 0}$ respectively account for the possibly null and definitely non-null cases. Thus, $t.(k)*_{\neq 0}$ should be interpreted as the address of the k-th byte of a value of type t and $t.(k)*$ represents the same set of values, with the addition of the value 0.

Example 1 (Doubly linked-lists and structures). Based on Definition 1, the fact that a `dll` object boils down to a pair of non-null `dll` pointers can be expressed by the type $\texttt{dll}.(0)*_{\neq 0} \times \texttt{dll}.(0)*_{\neq 0}$. We also remark that padding bytes added in structures to preserve field alignments can be added using $\ldots \times \texttt{word}_k$.

Before we can formally define the denotation of types, we need to introduce a few notions. As usual in languages like C, we can compute the size of the representation of a type. Since arrays may not be of a statically known size, the size may depend on the actual value of symbolic variables, hence it needs to be parameterized by a valuation ν. Then, size is computed by the function $\text{size}_\nu : \mathbb{T} \to \mathbb{N}$ defined by:

$$
\begin{array}{rl}
\text{size}_\nu(\texttt{word}_n) = n & \qquad \text{size}_\nu(t_1 \times t_2) = \text{size}_\nu(t_1) + \text{size}_\nu(t_2) \\
\text{size}_\nu(t_a*) = \text{size}_\nu(t_a*_{\neq 0}) = \mathcal{W} & \qquad \text{size}_\nu(t[s]) = \begin{cases} s \cdot \text{size}_\nu(t) & \text{if } s \in \mathbb{N} \\ \nu(s) \cdot \text{size}_\nu(t) & \text{if } s \in \mathbb{V}^\sharp \end{cases} \\
\text{size}_\nu(\{x : t \mid p(x)\}) = \text{size}_\nu(t) &
\end{array}
$$

Memory labeling. Physical types are aimed at describing not only variables like standard types do, but also memory locations. To formalize this, we introduce *labelings* as mappings from addresses to physical types.

Definition 2 (Labeling). *A* labeling *is a function $\mathcal{L} : \mathbb{A} \to \mathbb{T}_A$ such that each tagging of a region with a type is whole and contiguous, i.e., for all types $t \in \mathbb{T}$, for all addresses $a \in \mathbb{A}$, if we let $n = \text{size}_\nu(t)$, and if there exists $k \in [0, n-1]$ such that $\mathcal{L}(a+k) = t.(k)$, then:*

$$\mathcal{L}(a) = t.(0) \wedge \mathcal{L}(a+1) = t.(1) \wedge \ldots \wedge \mathcal{L}(a+n-1) = t.(n-1)$$

We extend this notion by letting labelings return a type: if $\mathcal{L}(a) = t$, then it should satisfy the above property. Moreover, we let $\mathbb{L}$ denote the set of labelings. Intuitively, $\mathcal{L}(a) = t$ means both that a points to a value of type t, and that a has type $t.(0)*_{\neq 0}$.

Example 2 (Labeling). We consider the state of Figure 2(a). In this case, the relations below form a valid labeling of the memory:

$$\begin{array}{llllll} \mathcal{L}: & \texttt{0x20} \mapsto \texttt{node}.(0) & \texttt{0x21} \mapsto \texttt{node}.(1) & \ldots & \texttt{0x2c} \mapsto \texttt{node}.(15) \\ & \texttt{0x60} \mapsto \texttt{node}.(0) & \texttt{0x61} \mapsto \texttt{node}.(1) & \ldots & \texttt{0x6c} \mapsto \texttt{node}.(15) \\ & \texttt{0x80} \mapsto \texttt{node}.(0) & \texttt{0x81} \mapsto \texttt{node}.(1) & \ldots & \texttt{0x8c} \mapsto \texttt{node}.(15) \end{array}$$

Subtyping relation. In Example 2, the labeling $\mathcal{L}$ conveys that the type of address `0x24` is `node`.(4). But we could also view this offset as the base address of a doubly-linked list and give it type `dll`.(0), since `node` contains a `dll` at offset 4. However, the former is more precise: all memory cells that contain a `node`.(4) contain a `dll`.(0), but the converse is not true. This remark motivates the definition of a physical form of subtyping relation. Intuitively, the above remark should be noted `node`.(4) $\preceq$ `dll`.(0). More generally, $t.(n) \preceq u.(m)$ means that t "contains" a u somewhere in its structure.

Definition 3 (Subtyping between address types). *The relation $\preceq \in \mathbb{T}_A \times \mathbb{T}_A$ is defined inductively according to the rules below:*

$$\frac{}{t.(k) \preceq t.(k)} \qquad \frac{t = \mathcal{M}(\texttt{n}) \quad 0 \leq k < \text{size}_\nu(t)}{\texttt{n}.(k) \preceq t.(k)} \qquad \frac{t.(k) \preceq u.(l) \quad u.(l) \preceq v.(m)}{t.(k) \preceq v.(m)}$$

$$\frac{0 \leq k < \text{size}_\nu(t_1)}{(t_1 \times t_2).(k) \preceq t_1.(k)} \qquad \frac{0 \leq k < \text{size}_\nu(t_2)}{(t_1 \times t_2).(\text{size}_\nu(t_1) + k) \preceq t_2.(k)}$$

$$\frac{0 \leq q < s \quad 0 \leq k < \text{size}_\nu(t)}{t[s].(q \cdot \text{size}_\nu(t) + k) \preceq t.(k)}$$

Interpretation of types. We now give the meaning of types in terms of an interpretation function that maps each type into a set of values. Unlike classical notions of types, the interpretation of a physical type depends on the data of a labeling $\mathcal{L}$ to resolve field pointers to other structures and on a valuation $\nu : \mathbb{V}^\sharp \to \mathbb{V}$ in order to check that side predicates are satisfied. In the following, we let $\text{eval}_\nu : \textit{pred} \times \mathbb{V} \to \mathbb{B}$ be the function that maps a predicate to its boolean value for valuation ν (the definition of eval_ν is classical thus omitted).

Definition 4 (Interpretation of types). *Given labeling $\mathcal{L}$ and valuation ν the* interpretation *function $(\!|\cdot|\!)_{\mathcal{L},\nu} : \mathbb{T} \to \mathcal{P}(\mathbb{N})$ is defined by:*

$$\begin{array}{rlrl} (\!|\texttt{word}_n|\!)_{\mathcal{L},\nu} &= \mathbb{V}_n & (\!|t_a*_{\neq 0}|\!)_{\mathcal{L},\nu} &= \{a \in \mathbb{A} \mid \mathcal{L}(a) \preceq t_a\} \\ (\!|t_1 \times t_2|\!)_{\mathcal{L},\nu} &= \{v_1 :: v_2 \mid \forall i,\ v_i \in (\!|t_i|\!)_{\mathcal{L},\nu}\} & (\!|t_a*|\!)_{\mathcal{L},\nu} &= (\!|t_a*_{\neq 0}|\!)_{\mathcal{L},\nu} \cup \{0\} \\ (\!|\{x : t \mid p(x)\}|\!)_{\mathcal{L},\nu} &= \{v \in (\!|t|\!)_{\mathcal{L},\nu} \mid \text{eval}_\nu(p, v) = \textbf{true}\} & & \\ (\!|t[s]|\!)_{\mathcal{L},\nu} &= \{v_0 :: v_1 :: \cdots :: v_{s-1} \mid v_0, \ldots, v_{s-1} \in (\!|t|\!)_{\mathcal{L},\nu}\} & & \end{array}$$

As shown below, the interpretation is monotone with respect to subtyping, which is consistent with Liskov's substitution principle [23], which means that all properties of addresses of type τ also hold for addresses of type υ, where $\upsilon \preceq \tau$:

Lemma 1 (Monotonicity). *Let $t.(n)$ and $u.(m)$ be two address types such that $t.(n) \preceq u.(m)$. Then $(\!| t.(n)*_{\neq 0} |\!)_{\mathcal{L},\nu} \subseteq (\!| u.(m)*_{\neq 0} |\!)_{\mathcal{L},\nu}$.*

Example 3. In Figure 2(a), and using the labeling of Example 2, $(\!| \texttt{node}.(4)*_{\neq 0} |\!)_{\mathcal{L},\nu}$ and $(\!| \texttt{dll}.(0)*_{\neq 0} |\!)_{\mathcal{L},\nu}$ both denote the set of addresses $\{\texttt{0x24}, \texttt{0x64}, \texttt{0x84}\}$.

In the analysis, the notion of subtyping and its properties with respect to interpretation have two applications: first, they allow verifying memory safety and the preservation of structural invariants by checking subtyping is preserved by memory updates; second, they also allow to over-approximate aliasing relations as we demonstrate now.

Definition 5 (Set of addresses covered by a type). *Let $\mathcal{L} \in \mathbb{L}$, $\nu : \mathbb{V}^\sharp \to \mathbb{V}$, and t be a type. Then, the* set of addresses covered by type t *is:*

$$\mathrm{addr}_{\mathcal{L},\nu}(t) ::= \{a \in \mathbb{A} \mid \exists i,\ 0 \leq i < \mathrm{size}_\nu(t) \wedge \mathcal{L}(a) \preceq t.(i)\}.$$

Definition 6 (Type containment). *Let $\nu : \mathbb{V}^\sharp \to \mathbb{V}$ be a valuation and $t, u \in \mathbb{T}$ be types. We say that "t* contains *u" if and only if:*

$$\exists i \in [0, \mathrm{size}_\nu(t))\ ,\ \forall k \in [0, \mathrm{size}_\nu(u))\ ,\ t.(i+k) \preceq u.(k).$$

Theorem 1 (Physical types and aliasing). *Let $t, u \in \mathbb{T}$. Then, either t and u cover disjoint regions, or one contains the other, i.e., if $\mathrm{addr}_{\mathcal{L},\nu}(t) \cap \mathrm{addr}_{\mathcal{L},\nu}(u) \neq \emptyset$, then t contains u or u contains t.*

Proof. This comes from the fact that $\preceq$ is a tree relation.

This result entails that physical types can be used to compute must-not alias information. As an example, in Figure 2(a), if we consider types `uf` and `dll`, neither of them contains the other, thus their addresses are disjoint.

States in a typing environment. In the following paragraphs, we define a typed semantics for WHILE$_{\text{MEM}}$. This semantics is conservative in the sense that it rejects some programs and executions that could be defined in the semantics of Section 3. In this second semantics, states are extended with typing information. Its goal is to serve as a step towards the verification of preservation of physical types. More precisely, a state should enclose not only a store and a heap, but also a labeling and a map from variables to types. Furthermore, such a state is well typed if the heap is consistent with the labeling and the variable values are consistent with their types.

Definition 7 (Well-typed state). *A* state *is a 4-tuple $(\sigma, h, \mathcal{L}, \Gamma)$, where $\sigma \in \mathbb{\Sigma}$, $h \in \mathbb{H}$, $\mathcal{L}$ is a labeling, and $\Gamma : \mathbb{X} \to \mathbb{T}$ maps variables to types. We write $\mathbb{S}_t$ for the set of such 4-tuples.*

Moreover, state $(\sigma, h, \mathcal{L}, \Gamma)$ is well typed *if and only if:*

1. *The labeling is consistent with the heap: for all address $a \in \mathbb{A}$, if there exists a type t such that $\mathcal{L}(a) = t.(0)$, then $h[a..a + \mathrm{size}_\nu(t)] \in (\!| t |\!)_{\mathcal{L},\nu}$;*
2. *Variables are well-typed: for all variable $x \in \mathbb{X}$, $\sigma(x) \in (\!| \Gamma(x) |\!)_{\mathcal{L},\nu}$.*

$$\frac{c \in \mathbb{V}_n}{(\sigma,h,\mathcal{L},\Gamma) \vdash c : \mathtt{word}_n} \qquad \frac{c \in \mathbb{A}}{(\sigma,h,\mathcal{L},\Gamma) \vdash c : \mathcal{L}(c)*_{\neq 0}} \qquad \frac{}{(\sigma,h,\mathcal{L},\Gamma) \vdash x : \Gamma(x)}$$

$$\frac{(\sigma,h,\mathcal{L},\Gamma) \vdash e : t.(i) *_{\neq 0} \quad t.(i) \preceq u.(j)}{(\sigma,h,\mathcal{L},\Gamma) \vdash e : u.(j)*_{\neq 0}} \qquad \frac{(\sigma,h,\mathcal{L},\Gamma) \vdash e : t.(i) * \quad t.(i) \preceq u.(j)}{(\sigma,h,\mathcal{L},\Gamma) \vdash e : u.(j)*}$$

$$\frac{(\sigma,h,\mathcal{L},\Gamma) \vdash e : t.(0) *_{\neq 0} \quad \mathrm{size}_\nu(t) = \ell}{(\sigma,h,\mathcal{L},\Gamma) \vdash *_\ell e : t} \qquad \frac{(\sigma,h,\mathcal{L},\Gamma) \vdash e_1 : \mathtt{word}_n \quad s \vdash e_2 : \mathtt{word}_n}{(\sigma,h,\mathcal{L},\Gamma) \vdash e_1 \diamond e_2 : \mathtt{word}_n}$$

$$\frac{(\sigma,h,\mathcal{L},\Gamma) \vdash e_1 : t.(o) *_{\neq 0} \quad (\sigma,h,\mathcal{L},\Gamma) \vdash e_2 : \mathtt{word}_{\mathcal{W}} \quad \mathcal{E}[\![e_2]\!](\sigma,h) = v_2 \quad 0 \le o + v_2 < \mathrm{size}_\nu(t)}{(\sigma,h,\mathcal{L},\Gamma) \vdash e_1 + e_2 : t.((o+v_2))*_{\neq 0}}$$

$$\frac{(\sigma,h,\mathcal{L},\Gamma) \vdash e : t \quad \mathrm{eval}_\nu(p, \mathcal{E}[\![e]\!](\sigma,h)) \text{ holds}}{(\sigma,h,\mathcal{L},\Gamma) \vdash e : \{x : t \mid p(x)\}} \qquad \frac{(\sigma,h,\mathcal{L},\Gamma) \vdash e : t.(0) * \quad \mathcal{E}[\![e]\!](\sigma,h) \neq 0}{(\sigma,h,\mathcal{L},\Gamma) \vdash e : t.(0)*_{\neq 0}}$$

Fig. 6. Typing rules for WHILE$_{\text{MEM}}$ expressions.

Typed semantics of expressions. Typing of expressions aims at proving that the evaluation of an expression will either return a value consistent with the type or a runtime error. Unlike classical type systems, we do not use physical types to prevent runtime errors directly; instead, we let the analysis discharge the verification of memory safety as a second step, after types have been computed. Given a store σ, a heap h, a labeling $\mathcal{L}$, a typing of variables Γ, an expression e, and a type t, we write $(\sigma,h,\mathcal{L},\Gamma) \vdash e : t$ when expression e can be given type t in the typing state $(\sigma,h,\mathcal{L},\Gamma)$. The typing of expressions are given in Figure 6. Intuitively, the type of addresses (resp., variables) is resolved by $\mathcal{L}$ (resp., Γ). Rules for base values and binary operators are classical. Memory reads and pointer arithmetics are typed using corresponding offset calculation over physical types. Subtyping allows replacing a type to a container type at pointer dereference points. Finally, types of expressions can be refined by the values these expressions evaluate to. This typing is sound in the following sense:

Theorem 2 (Soundness of typing of expressions). *Let an expression e, a valuation $\nu \in \mathbb{V}^\sharp \to \mathbb{V}$, a typing state $(\sigma,h,\mathcal{L},\Gamma)$ and a type $t \in \mathbb{T}$. Then, if $(\sigma,h,\mathcal{L},\Gamma)$ is well typed under ν, if $(\sigma,h,\mathcal{L},\Gamma) \vdash e : t$, and if $\mathcal{E}[\![e]\!](\sigma,h) = v$, then either $v = \Omega$ or $v \in (\!|\, t \,|\!)_{\mathcal{L},\nu}$.*

Note that an expression may be given several types in a same state due not only to subtyping but also to pointer arithmetics. For instance, if 8 has type $t.(0)*$, and 16 has type $u.(0)*$, then $8 + 16$ is at the same time of type $t.(16)*$ and $u.(8)*$.

Typed semantics of statements. The typed semantics of instructions is defined by a relation $\rightarrow_t \in (\mathit{stmt} \times \mathbb{S}_t) \times (\mathit{stmt} \times \mathbb{S}_t)$. It is mostly similar to the untyped semantics, but rules involving memory writes differ. Figure 7 displays the rules regarding memory writes. The rule for assignment not only updates the store but also the typing environment Γ. We note that this semantics is non-deterministic since the type of an expression is not unique in general. This semantics enjoys the type preservation property:

Theorem 3 (Preservation of typing of states). *Let a valuation $\nu : \mathbb{V}^\sharp \to \mathbb{V}$, and a typing state $(\sigma_0,h_0,\mathcal{L}_0,\Gamma_0)$, well typed under ν, such that $(\sigma_0,h_0,\mathcal{L}_0,\Gamma_0) \rightarrow_t (\sigma_1,h_1,\mathcal{L}_1,\Gamma_1)$. Then, $(\sigma_1,h_1,\mathcal{L}_1,\Gamma_1)$ is well typed under ν.*

$$\frac{(\sigma, h, \mathcal{L}, \Gamma) \vdash e : t \quad \mathcal{E}[\![e]\!](\sigma, h) = v}{\big(x := e,\ (\sigma, h, \mathcal{L}, \Gamma)\big) \to_t \big(\textbf{skip},\ (\sigma[x \leftarrow v], h, \mathcal{L}, \Gamma[x \leftarrow t])\big)}$$

$$\frac{\begin{array}{c}(\sigma, h, \mathcal{L}, \Gamma) \vdash e_1 : t.(0)*_{\neq 0} \\ (\sigma, h, \mathcal{L}, \Gamma) \vdash e_2 : t \qquad \text{size}_\nu(t) = \ell \qquad \mathcal{E}[\![e_1]\!](\sigma, h) = (\mathcal{W}, a) \qquad \mathcal{E}[\![e_2]\!](\sigma, h) = (\ell, v)\end{array}}{\big(*_\ell\, e_1 := e_2,\ (\sigma, h, \mathcal{L}, \Gamma)\big) \to_t \big(\textbf{skip}, (\sigma, h[a..a + \ell \leftarrow v], \mathcal{L}, \Gamma)\big)}$$

Fig. 7. Selected transition rules for programs.

Therefore, as we consider executions starting in a well-typed state only, Theorem 3 entails that well-typedness is an invariant. This semantics is not computable in general.

Last, we note that the typed semantics is more restrictive than the untyped one:

Theorem 4 (Semantic comparison). *If the typing states* $(\sigma_0, h_0, \mathcal{L}_0, \Gamma_0)$ *and* $(\sigma_1, h_1, \mathcal{L}_1, \Gamma_1)$ *are such that* $(p_0, (\sigma_0, h_0, \mathcal{L}_0, \Gamma_0)) \to_t (p_1, (\sigma_1, h_1, \mathcal{L}_1, \Gamma_1))$*, then* $(p_0, (\sigma_0, h_0)) \to (p_1, (\sigma_1, h_1))$*.*

Intuitively, the typed semantics is more restrictive than the untyped semantics in two ways: first, it considers only well-typed initial states only; second, it considers ill-typed memory writes as blocking, even though such a write may be part of a program fragment that overall preserves invariants. Finally, note that `malloc` calls cannot be readily incorporated in the typed semantics; this is solved in Section 7.

5 Type-based shape domain

We now set up the type-based shape abstract domain which serves as a basis for our analysis by defining its abstract elements and concretization function. This abstract domain combines type information with numerical constraints. Types constrain the regions pointed to by variables and may contain symbolic variables denoting numerical values. To cope with numerical constraints, our abstract domain is parameterized by a numerical domain such as that of intervals [9] or any other abstract domain. Thus, we assume such an abstract domain $\mathbb{D}^\sharp_{\text{num}}$ is fixed, together with a concretization function $\gamma_N : \mathbb{D}^\sharp_{\text{num}} \to (\mathbb{V}^\sharp \to \mathbb{V})$.

Abstract types. First, an abstract type defines a set of types where all symbolic variables are mapped into a numerical value. Due to the dependency on the association of symbolic variables to numerical values, its concretization returns pairs including a valuation $\nu : \mathbb{V}^\sharp \to \mathbb{V}$. It boils down to either a physical type or either of the $\bot, \top$ constant elements.

Definition 8 (Abstract types). *The set of* abstract types $\mathbb{T}^\sharp$ *is defined by the grammar below together with its concretization* $\gamma_T : \mathbb{T}^\sharp \to \mathcal{P}(\mathbb{T} \times (\mathbb{V}^\sharp \to \mathbb{V}))$*:*

$$\begin{array}{lll|lll}
\mathbb{T}^\sharp \ni t^\sharp ::= & \bot & & \gamma_T : \bot & \longmapsto & \emptyset \\
\quad | & t.(\alpha)* & & \top & \longmapsto & \mathbb{T} \times (\mathbb{V}^\sharp \to \mathbb{V}) \\
\quad | & t.(\alpha)*_{\neq 0} & & t.(\alpha)* & \longmapsto & \{(t.(\nu(\alpha))*,\ \nu) \mid 0 \leq \nu(\alpha) < \text{size}_\nu(t)\} \\
\quad | & \top & & t.(\alpha)*_{\neq 0} & \longmapsto & \{(t.(\nu(\alpha))*_{\neq 0},\ \nu) \mid 0 < \nu(\alpha) < \text{size}_\nu(t)\}
\end{array}$$

Example 4 (Array type). As an example, the abstract type and numerical constraints below abstract the information that one would attach to a pointer to some array of 10 integers:

- abstract type $\texttt{int}[10].(\alpha)$ states that we are looking at an address somewhere into such an array;
- numerical constraints $\alpha \in [0, 39] \wedge \exists k \in \mathbb{N},\ \alpha = 4k$ (expressible in a reduced product of intervals and congruences) refines the above abstract type by filtering out misaligned pointers.

Note that an address into an array of statically unknown length would write down $\texttt{int}[\alpha'].(\alpha)$, with matching numerical constraints.

Type-based shape abstraction. At this point, we can formalize the type based shape domain as follows, by letting each variable be abstracted by an abstract type. In order to also express constraints over the contents of variables, this abstraction also needs to attach to each variable a symbolic variable denoting its value.

Definition 9 (Type-based shape domain). *We let the* type-based shape domain $\mathbb{H}^\sharp$ *denote the set of pairs* $(\sigma^\sharp, \Gamma^\sharp)$ *pairs called* abstract stores*, where:*

- $\sigma^\sharp : \mathbb{X} \to \mathbb{V}^\sharp$ *is a mapping from variables to symbolic variables,*
- *and* $\Gamma^\sharp \in \mathbb{X} \to \mathbb{T}^\sharp$ *is a mapping from variables to abstract types.*

Moreover, the concretization for $\mathbb{H}^\sharp$ *is the function* $\gamma_H : \mathbb{H}^\sharp \to \mathcal{P}(\mathbb{S}_t \times (\mathbb{V}^\sharp \to \mathbb{V}))$ *defined by:*

$$\gamma_H\left(\Gamma^\sharp\right) = \left\{ \left((\sigma, h, \mathcal{L}, \Gamma),\ \nu\right) \;\middle|\; \begin{array}{l} (\sigma, h, \mathcal{L}, \Gamma) \text{ is well typed under } \nu \\ \text{and } \forall x \in \mathbb{X},\ \left(\Gamma(x), \nu\right) \in \gamma_T(\Gamma^\sharp(x)) \\ \text{and } \forall x \in \mathbb{X},\ \sigma(x) = \nu(\sigma^\sharp(x)) \end{array} \right\}$$

Definition 9 does not provide representable abstract states quite yet. Indeed, we still need to reason over the possible numerical values denoted by symbolic variables. The numerical abstract domain allows completing this last step.

Definition 10 (Combined shape abstraction). *The* combined shape-numeric abstract domain $\mathbb{S}^\sharp$ *and its concretization* $\gamma_S : \mathbb{S}^\sharp \to \mathcal{P}(\mathbb{S}_t \times (\mathbb{V}^\sharp \to \mathbb{V}))$ *are defined as follows:*

$$\mathbb{S}^\sharp ::= \mathbb{H}^\sharp \times \mathbb{D}^\sharp_{\text{num}} \qquad \gamma_S(h^\sharp, \nu^\sharp) = \{(s, \nu) \in \gamma_H(h^\sharp) \mid \nu \in \gamma_N(\nu^\sharp)\}$$

Figure 2(c) provides an example of an abstract state in this combined abstraction.

6 Static analysis

Our static analysis is a standard, forward, abstract interpretation-based static analysis [9]. We focus on important operations,like verifying that stores preserve type invariants, or the lattice operations. Both of these operations rely on a procedure called *abstract type checking*.

Structure of the interpreter. The analysis of $\text{WHILE}_{\text{MEM}}$ expressions and statements is respectively performed by the two functions $\mathcal{E}[\![\cdot]\!]^\sharp : \mathit{expr} \times \mathbb{S}^\sharp \to (\mathbb{V}^\sharp \times \mathbb{T}^\sharp) \times \mathbb{D}^\sharp_{\text{num}}$ and $[\![\cdot]\!]^\sharp : \mathit{stmt} \times \mathbb{S}^\sharp \to \mathbb{S}^\sharp$. The evaluation of expressions manipulates *abstract values* $\mathbb{V}^\sharp_t \stackrel{\text{def}}{=} \mathbb{V}^\sharp \times \mathbb{T}^\sharp$, which are the counterpart of concrete values in the concrete semantics. An abstract value $\alpha_{t^\sharp}$ is just a pair $(\alpha, t^\sharp) \in \mathbb{V}^\sharp_t$ with a symbolic variable α and an abstract type $t^\sharp$ respectively describing *all* the possible values and *some* possible types for an expression. The evaluation of symbolic variables is standard [5] (each node in the expression tree creates a fresh symbolic variable and updates the numerical domain accordingly), and the computation of abstract types follows closely the concrete typing rules given in Figure 6.

Abstract type checking. Abstract type checking verifies that, given the numerical constraints of $\nu^\sharp$, casting an abstract value $\alpha_{t^\sharp}$ into type $u^\sharp$ is safe (this is written as $\alpha : t^\sharp \overset{\nu^\sharp}{\rightsquigarrow} u^\sharp$). In most type checks (that we call *upcasts*), this is done by checking that $t^\sharp \preceq^\sharp_{\nu^\sharp} u^\sharp$, where $\preceq^\sharp$ is an ordering between abstract types which derives from the subtyping relation ($\preceq$) between concrete address types:

Theorem 5 (Soundness of abstract subtyping). *Let $\nu^\sharp \in \mathbb{D}^\sharp_{\text{num}}$, $t^\sharp, u^\sharp \in \mathbb{T}^\sharp$, $t.(i)$ and $u.(j) \in \mathbb{T}_A$. If $t^\sharp \preceq^\sharp_{\nu^\sharp} u^\sharp$, then*

$$\forall \nu \in \gamma_N(\nu^\sharp),\ (t.(i), \nu)* \in \gamma_T(t^\sharp) \wedge (u.(j), \nu)* \in \gamma_T(u^\sharp) \Longrightarrow t.(i) \preceq u.(j).$$

Example 5. (Upcasting after pointer arithmetics) Following Definition 3, $\texttt{int}[10].(\alpha)* \preceq^\sharp_{\upsilon^\sharp} \texttt{int}.(0)*$ holds when both numerical constraints $\alpha \in [0, 39] \wedge \exists k \in \mathbb{N},\ \alpha = 4k$ hold. Thus the abstract type given in Example 4 can be safely casted into an $\texttt{int}*$. Note that querying the numerical abstract domain is necessary to check the safety of this cast.

Some other type checks, like verifying that it is safe to transform a $t*$ pointer to a $t*_{\neq 0}$ pointer, or checking that the predicate of a refinement type holds, are *downcast* operations: to verify the safety of the cast, we examine not only the types, but also the numerical properties of the value being type checked:

Example 6. (Downcasting to a $*_{\neq 0}$ pointer) In Figure 1, the `merge` function gives `&x->dll`, whose type is $\texttt{node}.(\alpha)*$ with $\alpha = 4$, as an argument to `dll_union`, which eventually gets written into `y->prev`, a memory location that may contain values in the abstract type $\texttt{dll}.(\beta)*_{\neq 0}$ with $\beta = 0$: this requires to perform a type check. Casting to $\texttt{dll}.(\beta)*$ can be done using the abstract subtyping relation $\texttt{node}.(\alpha)* \preceq^\sharp_{\nu^\sharp} \texttt{dll}.(\beta)*$, but to cast to $\texttt{dll}.(\beta)*_{\neq 0}$, we must additionally check that the value `x->dll` (i.e. `&x` + offset `4`) cannot be 0; this makes use of the fact that pointer arithmetics inside a valid object cannot wrap around.

The soundness of the abstract type checking operation and its proof are established using the interpretation of the types:

Theorem 6 (Soundness of abstract type checking). *Let $(h^\sharp, \nu^\sharp) \in \mathbb{S}^\sharp$. Let $\alpha \in \mathbb{V}^\sharp$ be a symbolic variable and $t^\sharp, u^\sharp$ be two abstract types. If $\alpha : t^\sharp \overset{\nu^\sharp}{\rightsquigarrow} u^\sharp$ then*

$$\forall ((\sigma, h, \mathcal{L}, \Gamma), \nu) \in \gamma_S(h^\sharp, \nu^\sharp),\ \nu(\alpha) \in (\!|\, t^\sharp \,|\!)_{\mathcal{L},\nu} \Longrightarrow \nu(\alpha) \in (\!|\, u^\sharp \,|\!)_{\mathcal{L},\nu}.$$

Interpretation of base statements. As shown in Example 6, the analysis must ensure that all memory updates preserve the typing of states (Theorem 3), and abstract type checking is a central operation for doing this. The interpretation of memory updates is done as follows:

$$\begin{aligned}
[\![*_\ell e_1 := e_2]\!]^\sharp(h^\sharp, \nu^\sharp) &\stackrel{\text{def}}{=} (h^\sharp, \nu_2^\sharp) \text{ if } \alpha : t^\sharp \overset{\nu_2^\sharp}{\rightsquigarrow} u^\sharp \text{ where } u^\sharp \text{ is a non-null pointer type} \\
&\quad \text{and } (\alpha, t^\sharp, \nu_1^\sharp) = \mathcal{E}[\![e_1]\!]^\sharp(h^\sharp, \nu^\sharp) \text{ and } (\beta, w^\sharp, \nu_2^\sharp) = \mathcal{E}[\![e_2]\!]^\sharp(h^\sharp, \nu_1^\sharp) \\
&\quad \text{and } w^\sharp \text{ is the type of the values pointed by } (\alpha, u^\sharp) \\
[\![*_\ell e_1 := e_2]\!]^\sharp(h^\sharp, \nu^\sharp) &\stackrel{\text{def}}{=} \top \qquad \text{otherwise}
\end{aligned}$$

Note that memory updates do not modify the representation of the abstract heap, and is thus a fast operation. The evaluation of assignments is also fast, since it only needs to evaluate an expression and to record the abstract value in the abstract store:

$$\begin{aligned}
[\![x := e]\!]^\sharp((\sigma^\sharp, \Gamma^\sharp), \nu^\sharp) &\stackrel{\text{def}}{=} (\sigma^\sharp[x \leftarrow \alpha], \Gamma^\sharp[x \leftarrow t^\sharp], \nu_1^\sharp) \\
&\qquad \text{where } (\alpha, t^\sharp, \nu_1^\sharp) = \mathcal{E}[\![e]\!]^\sharp((\sigma^\sharp, \Gamma^\sharp), \nu^\sharp)
\end{aligned}$$

Lattice operations. The analysis of condition and loop commands is based on approximations for concrete unions and on conservative inclusion checks $\sqsubseteq_{\Phi,\mathbb{S}^\sharp}$ to test whether a post-fixpoint is reached [9]. The latter relies on abstract type checking. This operation consists in type-checking every $\textsc{While}_{\text{MEM}}$ variable, in addition to verifying the inclusion of the numerical constraints:

$$\begin{aligned}
((\sigma_1^\sharp, \Gamma_1^\sharp), \nu_1^\sharp) \sqsubseteq_{\Phi,\mathbb{S}^\sharp} ((\sigma_2^\sharp, \Gamma_2^\sharp), \nu_2^\sharp) &\stackrel{\text{def}}{=} \\
\nu_1^\sharp \sqsubseteq_{\Phi,\mathbb{D}_{\text{num}}^\sharp} \nu_2^\sharp \quad &\text{and} \quad \forall x \in \mathbb{X} : \ (\sigma_1^\sharp(x) : \Gamma_1^\sharp(x) \overset{\nu_1^\sharp}{\rightsquigarrow} \Gamma_2^\sharp(x))
\end{aligned}$$

where $\Phi : \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ is a renaming function that handles the fact that each abstract state refers to different variables [5]) (here it can be defined as $\forall x \in \mathbb{X} : \Phi(\sigma_1^\sharp(x)) = \sigma_2^\sharp(x)$).

Theorem 7 (Soundness of inclusion). *Let $s_1^\sharp, s_2^\sharp \in \mathbb{S}^\sharp$. Then:*

$$s_1^\sharp \sqsubseteq_{\Phi,\mathbb{S}^\sharp} s_2^\sharp \implies \forall (s, \nu) \in \gamma_S(s_1^\sharp), (s, \nu \circ \Phi) \in \gamma_S(s_2^\sharp)$$

The *join* operation can be deduced from the definition of $\sqsubseteq_{\Phi,\mathbb{S}^\sharp}$. These lattice operations are necessary to define the interpretation of `while` and `if` statements (which is standard). The interpretation of other statements is also standard.

Theorem 8 (Soundness of the abstract semantics). *Let $s^\sharp \in \mathbb{S}^\sharp$ be an abstract state and $c \in$ stmt be a statement. Then, $\gamma_S([\![c]\!]^\sharp(s^\sharp)) \supseteq [\![c]\!](\gamma_S(s^\sharp))$.*

7 Retained and staged points-to predicates

The type-based shape abstraction suffers from two important limitations. First, the heap is represented only in a summarized form by the type constraints, and there is no way to retain additional information about the contents of the heap. Second, all stores to memory must preserve the type invariants—situations where the type

$$\mathbb{P}^\sharp \stackrel{\text{def}}{=} \mathbb{V}_t^\sharp \rightharpoonup \mathbb{N} \times \mathbb{V}_t^\sharp \qquad\qquad \mathbb{U}^\sharp \stackrel{\text{def}}{=} \mathbb{S}^\sharp \times \mathbb{P}^\sharp \times \mathbb{P}^\sharp$$

$$\gamma_P : \mathbb{P}^\sharp \to \mathcal{P}\left(\mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V})\right)$$
$$\gamma_P(p^\sharp) = \{(h,\nu) \mid \forall(\alpha_t,\beta_t) \in \mathbb{V}_t^\sharp, \forall \ell \in \mathbb{N} : p^\sharp(\alpha_t) = (\ell,\beta_t) \implies h[\nu(\alpha)..\nu(\alpha)+\ell] = \nu(\beta)\}$$

$$\gamma_U : \mathbb{U}^\sharp \to \mathcal{P}\left(\mathbb{S} \times (\mathbb{V}^\sharp \to \mathbb{V})\right)$$
$$\gamma_U(s^\sharp, r^\sharp, p^\sharp) = \{\left((\sigma, h' \triangleright h),\ \nu\right) \mid \exists \Gamma, \mathcal{L}, h, h' : \left((\sigma,\Gamma,h,\mathcal{L}),\nu\right) \in \gamma_S(s^\sharp) \text{ and } (h,\nu) \in \gamma_P(r^\sharp) \text{ and } (h',\nu) \in \gamma_P(p^\sharp)\ \}$$

where $h' \triangleright h : \text{dom}(h') \cup \text{dom}(h) \to \mathbb{V}$ is defined as $(h' \triangleright h)(a) \stackrel{\text{def}}{=} h'(a)$ if $a \in \text{dom}(h')$; $(h' \triangleright h)(a) \stackrel{\text{def}}{=} h(a)$ otherwise

Fig. 8. Extending the base domain ($s^\sharp$) with retained ($r^\sharp$) and staged ($p^\sharp$) points-to predicates

invariants are temporarily violated are not handled. This happens when data is allocated but not yet initialized (as in function `make` in Figure 1), when updating a value with an invariant that spans multiple words, and in other situations.

We solve both problems by tracking some points-to predicates and attaching specific properties to them. The meaning of a points-to predicate $\alpha_{t^\sharp} \mapsto_\ell \beta_{u^\sharp}$ is, that for all possible valuations ν, the value (of size ℓ) stored in the heap at address $\nu(\alpha)$ is $\nu(\beta)$ and that α satisfies the abstract type $t^\sharp$ and β the abstract type $u^\sharp$. Points-to predicates are represented using a simple map p mapping a symbolic variable to another variable and size, and is concretized by considering all the possible values for each symbolic variable. In the following, we define and track the so-called *retained* and *staged* points-to predicates. Their combination is formally defined in Figure 8 (where each points-to predicate $\alpha_{t^\sharp} \mapsto_\ell \beta_{u^\sharp}$ is represented by bindings $\alpha_{t^\sharp} \mapsto (\ell, \beta_{u^\sharp})$ of a function $p^\sharp \in \mathbb{P}^\sharp$).

Retained points-to predicates. The type-based shape domain remembers flow-sensitive information only about the store, as the heap is represented only using the type invariants. We use *retained* points-to predicates $\alpha_{t^\sharp} \mapsto_\ell \beta_{u^\sharp}$ to store flow-sensitive information about the heap: they provide symbolic variables, like β, to represent values stored in the heap, so that they can be attached numerical and type information. In practice, retained points-to predicates achieve an effect comparable to materialization in shape analyses[4]. The concretization of these predicates is done by standard intersection.

Example 7 (Retained points-to predicate). Consider the abstract state (Figure 3) at line 19 of Figure 1. The binding from $(\beta, \texttt{uf}.(0)*)$ to $(\delta, \texttt{uf}.(0)*)$ (represented by an arrow) has been added by the read `parent->parent` at line 17. Having a

[4] A difference is that retained point-to predicate only retains information about a given cell, instead of modifying the heap summary to be precise on this cell.

variable δ materialized to represent the contents of β allows inferring $\delta \neq 0$ from the test `parent->parent != 0`.

Staged points-to predicates. A *staged* points-to predicate $\alpha \mapsto_\ell \beta$ represents a store operation performed by the program, but that is not yet propagated to the main domain $\mathbb{S}^\sharp$. The idea is that if that if an invariant is temporarily violated, subsequent stores may restore it; by grouping and delaying the stores to the type-based abstract domain $\mathbb{S}^\sharp$, we prevent $\mathbb{S}^\sharp$ from needing to take ill-typed states into account. In the concretization, the heap represented by the staged points-to predicates take precedence (operator $\triangleright$) over the heap represented by the type-based domain. Note this concretization allows describing states that are not well-typed, hence the codomain of γ_U is $\mathcal{P}\left(\mathbb{S} \times (\mathbb{V}^\sharp \to \mathbb{V})\right)$ instead of $\mathcal{P}\left(\mathbb{S}_t \times (\mathbb{V}^\sharp \to \mathbb{V})\right)$.

Example 8 (Staged points-to predicate). The contents of the memory allocated at line at line 40 of Figure 1 are unconstrained, and may not correspond to the type `node*` of the address returned by `malloc`: the reachable states at this line include ill-typed states that are not representable by $\mathbb{S}^\sharp$. This is fixed by introducing staged points-to predicates from the address returned by `malloc`, which allows the abstract value of the typed domain to represent only well-typed states, but still take into account the call to `malloc`. These staged points-to predicates are modified by the subsequent statements, and from line 44, the staged points-to predicates can be dropped by performing the corresponding stores to the memory, because the reachable states are now well-typed.

Static analysis operations. The addition of points-to predicates only changes the behaviour of memory operations (load,store, and `malloc`). The definition of these operations rely on determination of must and may-alias information between pairs (α_t, β_u) of abstract values. This is done using both the types (Theorem 1) and numerical information about addresses (e.g. addresses of an array at two indices i and j with $i < j$ will not alias), but this can be enhanced with information coming from other domains (like allocation sites [1]).

A $\texttt{malloc}_t$ of type t is interpreted simply by adding a staged points-to predicate $\alpha_{t*} \mapsto_\ell \beta_t$ where both α and β are fresh symbolic variables.

Loading a value of size ℓ at address α_t returns the value β_u if a points-to predicate $\epsilon_v \mapsto_\ell \beta_u$ exists in the domain and we can prove $\alpha = \epsilon$. Otherwise we performs a "weak read" by evaluating the load on the type-based domain, and joining the result with the values of all the staged points-to predicates whose addresses may alias with α_t. Finally, if β_u is the result of this operation, the analysis adds a new retained points-to predicate $\alpha_t \mapsto_\ell \beta_u$.

Storing a value δ_u of size ℓ at address α_t first needs to remove all points-to predicates that may alias with α_t. Retained points-to predicates are simply dropped, but staged points-to predicate must be propagated by performing the corresponding stores to the type-based shape domain. Then, a new staged points-to predicate $\alpha_t \mapsto_\ell \delta_u$ is added.

Example 9. Consider again the abstract state (Figure 3) at line 19 of Figure 1. The statement `x->parent = parent->parent` first reads `parent->parent` from mem-

ory, and retrieves δ, from the points-to predicate $\beta \mapsto \delta$. The store to `x->parent` (corresponding to address α) first needs to drop points-to predicates that may alias with α; on this abstract state, only $\beta \mapsto \delta$ is concerned. Finally, a new points-to predicate $\alpha \mapsto \delta$ is added. Note that $\alpha \neq \beta$ is an invariant of the program; if the type based shape domain were complemented with a more precise abstraction, then the points-to predicate $\beta \mapsto \delta$ would not need to be dropped.

8 Experimental evaluation

Research questions. The goal of our experimental evaluation is to evaluate the performance and precision of our analysis, the effort required for its parametrization, its ability to handle low-level (binary and system) code and complex sharing patterns.

Methodology. We have implemented two analyses (available at `https://zenodo.org/record/5512941`) using the Codex library for abstract interpretation: one for C code using the Frama-C platform (Frama-C/Codex); one for binary code using the Binsec platform (Binsec/Codex). All analyses have been conducted on a standard laptop (Intel Xeon E3-1505M 3Ghz, 32GB RAM). We took the mean values between 10 runs, and report the mean (all standard deviations were below 4%).

We ran our analysis on all the C benchmarks from two shape analysis publications; moreover we analyze their compiled version using gcc 10.3.0 with different levels of optimizations. These benchmarks are challenging: the `graph-*` benchmarks from Li et al. [22] were used to verify unstructured sharing patterns; to complete this evaluation we extend this with our running example. The other benchmarks from Li et al. [21] were used to demonstrate scalability issues faced by shape analyzers. Both benchmarks were created to demonstrate shape analysis, which is a more precise abstraction than the one we propose. Thus, they are suitable to evaluate performance, ability to handle complex sharing patterns, and precision.

This evaluation completes that in Nicole et al. [29], where we ran our analyzer on the kernel executable of an industrial embedded kernel (Asterios, developed by Krono-safe) to verify security properties (including full memory safety), with only 58 lines of manual annotations, which demonstrated the ability to handle low-level code, precision, performance and low amount of parametrization on a larger use case.

Results Table 1 provides the results of the evaluation. The benchmarks are grouped by the data structure they operate on; we report the number of lines describing physical types (**gen**erated from existing types information, or manually **ed**ited) shared by a group. The annotations mostly consist in constraining some pointers types to be non-null. The **pre** column describes necessary **pre**-conditions of the verified function (e.g. that a pointer argument must not be null). The **LOC** column is the number of lines of code of each function, excluding comments, blank lines and subroutines. The ratio of lines of manual annotations per line of code for a group, goes from 0% to 7.8%, with a mean of 3.2% and median of 2.7%.

The next columns in the table provide the **Time** taken by the full analysis (in s), the number of alarms of the full analysis ($\mapsto$ column) and the analysis without the retained and staged points-to predicates ($\not\mapsto$ column), for the C version of the code and the various binaries produced by GCC. For brevity we have omitted the time taken by the $\not\mapsto$ analysis

Benchmark	Annotations gen	ed	pre	LOC	C Time	C ↦	C ↛	O0 Time	O0 ↦	O0 ↛	O1 Time	O1 ↦	O1 ↛	O2 Time	O2 ↦	O2 ↛	O3 Time	O3 ↦	O3 ↛
sll-delmin	11	0	1	25	0.27	0	0	0.13	0	0	0.15	0	0	0.15	0	0	0.13	0	0
sll-delminmax			1	49	0.30	0	0	0.19	0	0	0.17	0	0	0.17	0	0	0.16	0	0
psll-bsort			0	25	0.30	0	22	0.41	0	3	0.25	0	3	0.26	0	3	0.29	0	3
psll-reverse	10	0	0	11	0.28	0	2	0.10	0	1	0.13	0	1	0.10	0	1	0.10	0	1
psll-isort			0	20	0.29	0	2	0.34	0	1	0.34	0	1	0.32	0	1	0.33	0	1
bstree-find	12	0	1	26	0.27	0	0	0.14	0	0	0.13	0	0	0.15	0	0	0.16	0	0
gdll-findmin			1	49	0.50	0	0	0.41	0	0	0.39	0	0	0.41	0	0	0.42	0	0
gdll-findmax			1	58	0.55	0	0	0.33	0	0	0.22	0	0	0.21	0	0	0.20	0	0
gdll-find	25	5	1	78	0.56	0	0	0.15	0	0	0.15	0	0	0.14	0	0	0.14	0	0
gdll-index			1	55	0.53	0	0	0.32	0	0	0.33	0	0	0.30	0	0	0.29	0	0
gdll-delete			1	107	0.57	0	2	0.16	0	0	0.14	0	0	0.13	0	0	0.13	0	0
javl-find			2	25	0.35	0	0	0.23	0	0	0.28	0	0	0.18	0	0	0.19	0	0
javl-free	45	12	1	27	0.35	0	4	0.11	0	3	0.12	0	0	0.10	0	0	0.11	0	0
javl-insert			2	95	0.53	6	56	0.52	12	20	0.39	30	34	0.43	29	34	0.43	29	34
javl-insert-32×			2	95	16.68	192	1792	28.28	14	20	33.14	34	34	32.00	32	34	40.01	32	34
gbstree-find			1	53	0.58	0	0	0.38	0	0	0.40	0	0	0.56	0	0	0.59	0	0
gbstree-delete	23	5	1	165	0.81	0	0	0.90	0	0	0.72	0	0	0.67	0	0	0.66	0	0
gbstree-insert			1	133	0.55	0	7	0.26	0	0	0.21	0	0	0.23	0	0	0.24	0	0
brbtree-find			2	29	0.32	0	0	0.17	0	0	0.19	0	0	0.23	0	0	0.23	0	0
brbtree-delete	24	3	2	329	0.79	103	127	1.15	44	73	1.23	46	53	0.85	58	63	0.84	58	63
brbtree-insert			2	177	0.61	24	47	0.90	11	23	0.47	16	17	1.22	21	17	0.97	21	17
bsplay-find			1	81	0.53	0	18	0.25	0	7	0.23	0	7	0.23	0	7	0.23	0	7
bsplay-delete	22	1	1	95	0.72	0	38	0.45	0	11	0.44	0	10	0.44	0	10	0.44	0	10
bsplay-insert			1	101	0.57	0	18	0.25	0	7	0.25	0	7	0.25	0	7	0.25	0	7
graph-nodelisttrav			1	12	0.20	0	0	0.10	0	0	0.10	0	0	0.10	0	0	0.11	0	0
graph-path			1	19	0.21	0	14	0.15	0	5	0.16	0	0	0.14	0	0	0.16	0	0
graph-pathrand			1	25	0.22	0	10	0.13	0	0	0.21	0	0	0.12	0	0	0.11	0	0
graph-edgeadd	23	0	1	15	0.27	0	2	0.12	0	1	0.11	0	1	0.10	0	1	0.10	0	1
graph-nodeadd			1	15	0.26	0	2	0.10	0	1	0.08	0	1	0.09	0	1	0.10	0	1
graph-edgedelete			1	11	0.20	0	2	0.10	0	1	0.10	0	0	0.10	0	0	0.11	0	0
graph-edgeiter			1	22	0.23	0	0	0.13	0	0	0.11	0	0	0.12	0	0	0.12	0	0
uf-find			1	11	0.31	0	24	0.07	0	6	0.09	0	0	0.08	0	0	0.07	0	0
uf-merge	33	3	1	17	0.34	0	50	0.13	0	7	0.18	0	0	0.18	0	0	0.15	0	0
uf-make			0	9	0.31	0	4	0.05	0	3	0.06	0	3	0.07	0	3	0.06	0	3
Total verified						30	13		30	16		30	21		30	21		30	21

Table 1. Results of the evaluation

in the benchmarks; on average this analysis takes 1.5% less time for the C, and 20% less for binary code (maximum: 45%). The number of alarms is counted differently in C (one possible alarm each time the analyzer evaluates a statement) and in binary (where alarms are uniquified per instruction), but in both 0 alarms means that the analyzer verified type-safety. We observe that the full analyzer succeeds in verifying 30 benchmarks (out of 34), both in C and binary code. Removing the points-to predicates makes the analysis significantly less precise, as only 13 benchmarks are verified in C, and between 16 (for `-O0`) and 21 (for `-O1`,`-O2`,`-O3`) in binary code.

Discussion and conclusions Our combination of domains is effective at verifying type safety (which entails spatial memory safety) on C and binary code, even for benchmarks that have complex sharing patterns, with a low amount of annotations. The analysis performs evenly well on all benchmarks, and scales well on $\texttt{javl} - \texttt{insert} - 32\times$, which is challenging even for shape analysis with disjunctive clumping [21]. We interpret the fact that binary analysis is faster than the C analysis by implementation issues in the C analyzer.

The points-to predicates are very important for precision, as otherwise the number of false alarms raises significantly. The analysis succeeds equally on binary programs and on C programs, despite the complex code patterns that the C compiler may produce. Note that without points-to predicates, more binary codes are verified than in C: indeed in some cases the compiler performs a register promotion of a heap value, which removes the need for a points-to predicate.

9 Related works and conclusion

Memory analyses based on type inference. Several analyses that partially verify spatial memory safety using static type inference have been proposed. As unification-based type inference is less expressive than abstract interpretation, it is insufficiently precise to verify spatial memory safety, which is generally addressed by also using dynamic verification (e.g. Cyclone [16], CCured [28], CheckedC [14]). Still, the structural subtyping notion that we use is similar to the physical subtyping by Chandra and Reps [4], even if the physical type safety property that they verify does not include spatial memory safety (e.g. it does not check pointer arithmetics or null pointer dereferences). Liquid types [32] provide refinement types similar to ours, that are type checked by enhancing type inference with abstract interpretation and SMT solving. They discuss several limitations that our work solves: lack of structural subtyping (that we solve using our ordering on concrete and abstract types), and conservative decisions of when to fold and unfold variables (that we solve by using abstract interpretation instead of type inference [8], which allows our focusing decisions to be based on the current results of the analysis).

Other type-based memory analyses. Type-based alias analyses [12] propose a system to determine aliasing based on subtyping relations, which is present in our work (Theorem 1). These analyses assume that type safety is verified by other means (e.g. type checking), while our abstract interpretation also verifies type safety, on unsafe languages like C and binary code. Data structure analysis [19] produces a flow-insensitive description of data structure layout similar to our description of types (excluding numerical predicates), which could be used to split our types into distinct subtypes, making our analysis more precise. The structural analysis by Marron et al. [26] is also an intermediate between pointer and shape analyses, which is more precise than our type-based shape domain as it builds a flow-sensitive abstract heap information (the storage shape graph), while our description of types is flow-invariant. But their analysis proceeds on a type-safe language with no type cast, pointer arithmetic, interior pointers, or uninitialized data. Contrary to their results, our experience indicates that strong updates are important to verify the preservation of structural invariants, which we believe comes from the lower-level nature of our source languages. In a previous work [29] we used our type-based domain to verify security properties of an industrial embedded kernel; this work formally presents the analysis, extended with retained and staged points-to predicates and support for dynamic memory allocation.

Shape analyses. Many challenges arise in programs manipulating memory. These have been individually adressed by existing work on shape analyses, for instance to limit disjunctions [25,21], to adapt to custom data structures [34,7], to interpret low-level memory operations [18,20,13], to allow composite data structures [2,37], interaction with arrays [24], data structure invariants [7], or unstructured sharing [22]. Our type-based analysis is less precise than a full shape analysis, as e.g. it cannot verify temporal memory safety (i.e. use-after-free errors), but it simultaneously handles all the above aspects in a simpler analysis, which is sufficiently precise to verify preservation of structural invariants and spatial memory safety.

References

1. Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, 1994.

2. Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2007.
3. Neil Brown. Linux kernel design patterns - part 2. *Linux Weekly News*, June 2009.
4. Satish Chandra and Thomas Reps. Physical type checking for C. In *ACM SIGSOFT Software Engineering Notes*, volume 24, pages 66–75. ACM, 1999.
5. Bor-Yuh Evan Chang and Xavier Rival. Modular construction of shape-numeric analyzers. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday*, volume 129 of *EPTCS*, pages 161–185, 2013.
6. Bor-Yuh Evan Chang, Xavier Rival, and George Necula. Shape analysis with structural invariant checkers. In *Static Analysis Symposium (SAS)*, pages 384–401. Springer, 2007.
7. Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 384–401. Springer, 2007.
8. Patrick Cousot. Types as abstract interpretations. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 1997.
9. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 1977.
10. Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Conference on Programming Languages Design and Implementation (PLDI)*, pages 230–241. ACM, 1994.
11. Dino Distefano, Peter O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 287–302. Springer, 2006.
12. Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Conference on Programming Languages Design and Implementation (PLDI)*, pages 106–117, 1998.
13. Kamil Dudka, Petr Peringer, and Tomás Vojnar. Byte-precise verification of low-level list manipulation. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 215–237. Springer, 2013.
14. Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development (SecDev '18)*, pages 53–60. IEEE, September 2018.
15. Timothy S. Freeman and Frank Pfenning. Refinement types for ML. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 268–277. ACM, 1991.
16. Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.
17. Andrew Kennedy. Compiling with Continuations, Continued. In *International Colloquium on Functional Programming (ICFP)*, page 14, 2007.

18. Jörg Kreiker, Helmut Seidl, and Vesal Vojdani. Shape analysis of low-level C with overlapping structures. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 214–230, 2010.
19. Chris Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005. *See* `http://llvm.cs.uiuc.edu`.
20. Vincent Laviron, Bor-Yuh Evan Chang, and Xavier Rival. Separating shape graphs. In *European Symposium on Programming (ESOP)*, pages 387–406, 2010.
21. Huisong Li, Francois Berenger, and Bor-Yuh Evan Chang. Semantic-directed clumping of disjunctive abstract states. In *Symposium on Principles of Programming Languages (POPL)*, pages 32–45, 2017.
22. Huisong Li, Xavier Rival, and Bor-Yuh Evan Chang. Shape analysis for unstructured sharing. In *Static Analysis Symposium (SAS)*, pages 90–108, 2015.
23. Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.
24. Jiangchao Liu and Xavier Rival. An array content static analysis based on non-contiguous partitions. *Comput. Lang. Syst. Struct.*, 47:104–129, 2017.
25. Roman Manevich, Shmuel Sagiv, Ganesan Ramalingam, and John Field. Partially disjunctive heap abstraction. In Roberto Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2004.
26. Mark Marron. Structural analysis: Shape information via points-to computation. *arXiv e-prints*, page arXiv:1201.1277, Jan 2012.
27. Mark Marron, Manuel V. Hermenegildo, Deepak Kapur, and Darko Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In *Conference on Compiler Construction (CC)*, pages 245–259, 2008.
28. George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.
29. Olivier Nicole, Matthieu Lemerre, Sébastien Bardin, and Xavier Rival. No crash, no exploit: Automated verification of embedded kernels. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 27–39, 2021.
30. Olivier Nicole, Matthieu Lemerre, and Xavier Rival. Lightweight shape analysis based on physical types (full version). Technical report, CEA List, ENS, 2021. https://binsec.github.io/assets/publications/papers/2021-vmcai-full-with-appendices.pdf.
31. John Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logics In Computer Science (LICS)*, pages 55–74. IEEE, 2002.
32. Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*, pages 131–144, Madrid, Spain, 2010. Association for Computing Machinery.
33. Mooly Sagiv, Thomas Reps, and Reinhard Whilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):50, 1998.
34. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.
35. Yannis Smaragdakis and George Balatsouras. Pointer Analysis. *FNT in Programming Languages*, 2(1):1–69, 2015.

36. Robert E. Tarjan and Jan van Leeuwen. Worst-Case Analysis of Set Union Algorithms. *JACM*, 31:245–281, 1984.
37. Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. Reduced product combination of abstract domains for shapes. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *Lecture Notes in Computer Science*, pages 375–395. Springer, 2013.