A Product of Shape and Sequence Abstractions

Josselin Giet[1], Félix Ridoux[2,3], and Xavier Rival[1]

[1] INRIA Paris/CNRS/École Normale Supérieure/PSL Research University
[2] IMDEA Software Institute, Madrid, Spain
[3] Univ Rennes, F-35000 Rennes, France
`firstname.lastname@{ens.fr|ens-rennes.fr}`

Abstract. Traditional separation logic-based shape analyses utilize inductive summarizing predicates so as to capture general properties of the layout of data-structures, to verify accurate manipulations of, e.g., various forms of lists or trees. However, they also usually abstract away contents properties, so that they may only verify memory safety and invariance of data-structure shapes. In this paper, we introduce a novel abstract domain to describe sequences of values of unbounded size, and track constraints on their length and on extremal values contained in them. We define a reduced product of such a sequence abstraction together with an existing shape abstraction so as to infer both shape and contents properties of data-structures. We report on the implementation of the sequence domain, its integration into a static analyzer for `C` code, and we evaluate its ability to verify partial functional correctness properties for list and tree algorithms.

1 Introduction

Dynamically allocated data-structures based on lists, trees or graphs are common due to their flexibility as containers. However, programs using them are notoriously difficult to get right, especially in presence of destructive updates. Indeed, the correctness of such programs relies on a wide spectrum of properties that comprise memory safety (the absence of illegal pointer operations such as the dereference of a null pointer), the preservation of structural invariants like acyclicity, and subtle functional properties and relationships between the structure layout and its contents such as sortedness. For instance, let us consider a program that inserts an element in a binary search tree. First, it should not cause any runtime error or memory leak. Second, it should not create a cycle or break the tree structure. Third, it should preserve the binary search tree property and be functionally correct, namely ensure that the elements in the tree after insertion are the same as before plus the new element, inserted at the correct position, with respect to the order.

```
struct tree { struct tree *l, *r; int d; };
```

$\mathbf{tree}(\alpha) :=$
$\quad | \ \mathbf{emp} \wedge \alpha = \mathbf{0x0}$
$\quad | \ \exists \alpha_l, \alpha_r, \delta, \alpha.\mathtt{l} \mapsto \alpha_l * \alpha.\mathtt{r} \mapsto \alpha_r * \alpha.\mathtt{d} \mapsto \delta * \mathbf{tree}(\alpha_l) * \mathbf{tree}(\alpha_r) \wedge \alpha \neq \mathbf{0x0}$
$\mathbf{tree}_s(\alpha, S) :=$
$\quad | \ \mathbf{emp} \wedge \alpha = \mathbf{0x0}$
$\quad | \ \exists \alpha_l, \alpha_r, \delta, S_l, S_r, \alpha.\mathtt{l} \mapsto \alpha_l * \alpha.\mathtt{r} \mapsto \alpha_r * \alpha.\mathtt{d} \mapsto \delta * \mathbf{tree}_s(\alpha_l, S_l) * \mathbf{tree}_s(\alpha_r, S_r)$
$\qquad \wedge \alpha \neq \mathbf{0x0} \wedge S = S_l.\mathtt{[}\delta\mathtt{]}.S_r$

Fig. 1. A `C` tree data-type and associated inductive summarizing predicates

Abstract interpretation [17] provides a general framework to build a sound static analysis from a basic semantics and an abstraction relation, and to verify semantic properties. Notably, it has been applied to verify numerical properties [20,35], the absence of runtime errors [7], string properties [25,4], array properties [28,29,30,19], liveness properties [55], and security properties [5,26,24]. Several families of shape analyses have also been designed to infer properties of programs manipulating dynamic data-structures, including TVLA [51] and shape analyses based on separation logic [50]. They can reason over structures like lists [12,13] or more general families of structures with an inductive layout [14,31] such as binary trees.

However, few shape analyses reason not only about the layout of data-structures but also about their contents, so as to verify, e.g., that a container consists of the expected collection of elements with the expected multiplicity. While [39,21] handle set predicates, they do not track properties related to element multiplicities or order. Similarly, [41] handles sorting properties of specific families of composite structures in arrays but does not consider general lists or trees. The analyses presented in [10,9,11] precisely abstract singly-linked lists storing numerical data. They compute numerical properties over these data such as "variable `x` is the sum of all elements in list `l`", or relation between element values and indexes to express sorting. However, it does not handle trees or doubly linked lists. Therefore, in this paper, we seek for an abstraction of data-structure contents that can verify complex invariants (e.g., involving elements orders or multiplicities) as well as some functional properties (like sorting). To illustrate our approach, we consider the classical tree type definition shown in Figure 1 and assume that we only consider acyclic instances. The inductive predicate **tree** summarizes valid memory regions storing exactly a complete and acyclic tree. More precisely, the predicate $\mathbf{tree}(\alpha)$ either describes an empty tree (then, α is the null pointer), or a memory region where α points to a valid `tree` block, the `l` and `r` fields of which point to the roots of disjoint (possibly empty) subtrees, as expressed by predicates $\mathbf{tree}(\alpha_l)$ and $\mathbf{tree}(\alpha_r)$. Note that separating conjunction $*$ [50] combines disjoint memory regions. A basic region is either an atomic cell described by a points-to predicate such as $\alpha.\mathtt{d} \mapsto \delta$ or an instance of some inductive predicate. While predicate **tree** describes the layout of memory cells and pointers, it does not convey any information about their contents. By contrast, $\mathbf{tree}_s$ extends **tree** with an additional symbolic parameter S to expose the sequence of values stored in the tree, read from left to right. When the tree is empty, so is its sequence of elements. The sequence stored in a non-empty tree is obtained by first considering the left subtree, then the contents of the root node and finally the right tree. If we additionally require the elements of S be sorted, then $\mathbf{tree}_s(\alpha, S)$ describes binary search trees with root α.

An advantage of this approach is that it allows to split the abstraction into two rather independent components, namely a separation logic based abstraction of the data-structures and another abstraction for properties of sequences of values stored in them. While [39] extends inductive predicates in a similar manner, it only supports set constraints. Therefore, we introduce a new abstract domain devoted to the representation of constraints over sequences. Existing sequence abstractions typically rely on regular expressions or finite automata [45,3,47]. More recently, [4] extends such an abstraction with sub-string, length, and element position constraints. However, these abstractions lack predicates such as constraints over extremal elements or sortedness. Our sequence abstract domain expresses not only relational constraints (it can express that a symbolic sequence is a fragment of another) but also constraints over length, extremal values, and specific predicates like sortedness. Although we use this sequence abstract domain for shape analysis, it could be used independently for other kinds of analyses.

To take advantage of this abstraction in shape analysis, we define a reduced product with a separation logic-based shape abstract domain. This product ties symbolic parameters of inductive predicates in separation logic together with sequence constraints. Sequence constraints that are inferred during the analysis (for instance when unfolding inductive predicates) are passed to the sequence domain. The reduced product also ensures communication between both domains for the computation of abstract operators such as union.

To summarize, we make the following contributions:

- After we overview our analysis in Section 2, we introduce a relational abstract domain dedicated to reasoning over sequences in Section 3;
- We define a reduced product between the new sequence domain and a separation logic-based abstract domain so as to extend a shape analysis with sequence reasoning capability. We first introduce the basic elements of the reduced product in Section 4 in the context of singly-linked lists. We discuss issues related to general inductive predicates in Section 5.
- We report on the implementation of our analysis in the MemCAD static analyzer [38] and on its evaluation in Section 6. We show that it can cope with the verification of sorting programs and operations over binary search trees.

2 Overview

In this section, we give an overview of the main principles of our static analysis by demonstrating it on the insertion program shown in Figure 2. When applied to a binary search tree, this function inserts an element at the expected position to preserve sortedness. We study the verification of functional correctness expressed as partial correctness with respect to a pre-condition and a post-condition (Figure 2). To formalize these, we let **sort** be a symbolic function over sequences of values that

```
1   void insert(struct tree* t, int i){
2     // assume $\mathbf{tree}_s(\mathtt{t}, S) \wedge S = \mathbf{sort}(S)$
3     if(t == null){
4       // ...
5     }else{
6       struct tree* c = t;
7       while(c->d <= i && c->l != null ||
8             c->d > i && c->r != null)
9         c = (c->d <= i) ? c->l : c->r;
10      // ...
11    }
12  } // assert $\mathbf{tree}_s(\mathtt{t}, \mathbf{sort}(S.[\mathtt{i}]))$
```

Fig. 2. Function for insertion in a binary search tree

$$\big(\&\mathtt{t} \mapsto \alpha_0 * \&\mathtt{c} \mapsto \alpha_0 * \mathbf{tree}_s(\alpha_0, S)\big) \quad \wedge \quad (S = \mathbf{sort}(S) \wedge \alpha_0 \neq \mathbf{0x0})$$

(a) Abstract state at the end of line 6

$$\begin{pmatrix} \&\mathtt{t} \mapsto \alpha_0 * \&\mathtt{c} \mapsto \alpha_1 \\ * \; \alpha_0.\mathtt{l} \mapsto \alpha_1 * \mathbf{tree}_s(\alpha_1, S_l) \\ * \; \alpha_0.\mathtt{d} \mapsto \delta \\ * \; \alpha_0.\mathtt{r} \mapsto \alpha_2 * \mathbf{tree}_s(\alpha_2, S_r) \end{pmatrix} \wedge \begin{pmatrix} S = S_l.[\delta].S_r \wedge S = \mathbf{sort}(S) \\ \wedge \; S_l = \mathbf{sort}(S_l) \wedge S_r = \mathbf{sort}(S_r) \\ \wedge \; \mathtt{max}_{S_l} \leq \delta \leq \mathtt{max}_{S_r} \\ \wedge \; \delta \leq \mathtt{i} \wedge \alpha_0, \alpha_1 \neq \mathbf{0x0} \end{pmatrix}$$

(b) Abstract state at the end of line 9, first case of the condition

$$\begin{pmatrix} \&\mathtt{t} \mapsto \alpha_0 * \&\mathtt{c} \mapsto \alpha' \\ * \; \mathbf{treeseg}_s(\alpha_0, \alpha', S_1 \boxdot S_2) \\ * \; \mathbf{tree}_s(\alpha', S_0) \end{pmatrix} \wedge \begin{pmatrix} S = S_1.S_0.S_2 \wedge S = \mathbf{sort}(S) \\ \wedge \; S_i = \mathbf{sort}(S_i) \;\; i \in \{0, 1, 2\} \\ \wedge \; \mathtt{max}_{S_1} \leq \mathtt{i} \leq \mathtt{max}_{S_2} \wedge \mathtt{max}_{S_1} \leq \mathtt{max}_{S_0} \\ \wedge \; \mathtt{min}_{S_0} \leq \mathtt{max}_{S_2} \wedge \alpha_0, \alpha' \neq \mathbf{0x0} \end{pmatrix}$$

(c) Abstract state after the first widening

Fig. 3. Selected abstract states

maps any sequence to its sorted permutation. Then, the pre-condition assumption assumes that `t` is a well-formed tree described with predicate $\mathbf{tree}_s(\mathtt{t}, S)$ and such that $S = \mathbf{sort}(S)$ (i.e., such that the elements S in `t` are sorted). Likewise, the post-condition asserts that `t` is still a well-formed tree, the contents of which is sorted and comprises exactly the elements in S plus the added value `i`.

We now discuss the abstraction used by our static analysis. We combine an existing memory abstraction, inspired by separation logic-based shape analyses such as [12,14], a relational numerical abstraction such as convex polyhedra [20], and a novel abstract domain for sequences. Intuitively, the latter describes conjunctions of constraints over both symbolic sequences of values (such as S) and values manipulated by the program. These constraints consist of equalities of pairs of symbolic sequence expressions such as $S' = \mathbf{sort}(S.[\mathtt{i}])$. Moreover, the inductive predicates used in the memory abstraction are instances of the $\mathbf{tree}_s$ predicate of Figure 1. For instance, the abstract pre-condition simply consists of the memory predicate $\mathbf{tree}_s(\mathtt{t}, S)$ and the sequence predicate $S = \mathbf{sort}(S)$, for some existentially quantified symbolic sequence variable S.

The analysis proceeds by forward abstract interpretation [17]: it computes over-approximate abstract post-conditions for basic statements, and uses widening to enforce the convergence of abstract iterations for loops. Since the analysis uses a reduced product [18], an abstract state consists of a pair of components, namely the shape abstraction that describes the layout of data-structures and the contents' abstraction made of constraints over values and sequences of values. For each analysis step, information stored in either component may be used in order to refine the other, which we discuss next.

We focus on the analysis of the loop that searches for the insertion point in the **else** branch. First, the analysis of the condition test enriches the pre-condition with the constraint $\texttt{t} \neq \texttt{null}$ as shown in the abstract state in Figure 3(a). Then, the analysis continues with the loop. The condition is a disjunction thus the analysis considers each case separately. For the first case, it refines the abstract state to reflect that the condition `c—>d <= i && c—>l != null` evaluates to **true**. Since both memory cells `c—>d` and `c—>l` are abstracted by the predicate $\mathbf{tree}_s(\alpha_0, S)$, this predicate needs to be unfolded to enable the analysis of the condition. The first disjunct of inductive predicate $\mathbf{tree}_s$ (Figure 1), which corresponds to the null pointer, is ruled out by constraint $\texttt{t} \neq \texttt{null}$. Therefore, only the second disjunct (non-empty tree) needs be considered. This shows how one component of the abstract state can refine the other. Thus, the analysis generates a new abstract state that exposes the root of the tree and lets α_1, α_2, and δ denote the contents of its `l`, `r`, and `d` fields. We remark that the inductive predicate unfolding also splits the symbolic sequence into $S = S_l.\texttt{[}\delta\texttt{]}.S_r$. Then the sequence domain derives that S_l and S_r are sorted since they are subsequences of a sorted sequence. It also infers that all values in S_l are less than δ that is itself less than all values in S_r by definition of **sort**, which writes down $\texttt{max}_{S_l} \leq \delta \leq \texttt{min}_{S_r}$. Last, it also retains the numerical constraint $\texttt{i} \leq \delta$. Figure 3(b) shows the resulting abstract state after the assignment line 9. In the case of the other disjunct, the tree is also unfolded but `c` points to α_2 instead of α_1 and the constraint over `i` and δ is $\delta < \texttt{i}$.

The widening operator over-approximates abstract union of successive abstract iterates at loop head. In this case, it generalizes abstract states such as the ones shown in Figure 3(a) and Figure 3(b) by weakening them locally. Indeed, in all three states, `c` points to a well-formed tree containing a sequence S_0. Moreover, the remaining of the memory region corresponds to a (possibly empty) partial tree: if it was completed by a tree with root pointed by `c`, the whole region would form a complete tree with root pointed by `t`. We call such a partial tree a *tree segment predicate* (the name *segments* comes from the analogy with list segments) and observe that it is can be automatically derived from $\mathbf{tree}_s$ and defined by induction in Figure 4. When widening synthesizes an instance of $\mathbf{treeseg}_s$ in Figure 3(c), it needs to infer its sequence argument. The sequence S of elements stored into the whole tree can be split into three parts, S_0, S_1, and S_2 where S_0 is the sequence of elements stored in the subtree pointed to by `c` and S_1 (resp., S_2) denote the sequence of elements stored in the "left" (resp., "right") part of the tree segment. This implies that the sequence argument of $\mathbf{treeseg}_s$ is not a contiguous sequence. Therefore, it is represented as $S_1 \boxdot S_2$ in the loop invariant Figure 3(c) where the

$$\begin{array}{l}
\mathbf{treeseg}_s(\alpha, \alpha', S \boxdot S') := \\
\quad \mid \mathbf{emp} \land \alpha = \alpha' \land S = S' = \texttt{[]} \\
\quad \mid \exists \alpha_l, \alpha_r, \texttt{v}, S_l, S'_l, S_r, \alpha.\texttt{l} \mapsto \alpha_l * \alpha.\texttt{r} \mapsto \alpha_r * \alpha.\texttt{d} \mapsto \texttt{v} * \mathbf{treeseg}_s(\alpha_l, \alpha', S_l \boxdot S'_l) \\
\qquad * \ \mathbf{tree}_s(\alpha_r, S_r) \land \alpha \neq \mathbf{0x0} \land S = S_l \land S' = S'_l.\texttt{[v]}.S_r \\
\quad \mid \exists \alpha_l, \alpha_r, \texttt{v}, S_l, S_r, S'_r, \alpha.\texttt{l} \mapsto \alpha_l * \alpha.\texttt{r} \mapsto \alpha_r * \alpha.\texttt{d} \mapsto \texttt{v} * \mathbf{tree}_s(\alpha_l, S_l) \\
\qquad * \ \mathbf{treeseg}_s(\alpha_r, \alpha', S_r \boxdot S'_r) \land \alpha \neq \mathbf{0x0} \land S = S_l.\texttt{[}v\texttt{]}.S_r \land S = S'_r
\end{array}$$

Fig. 4. Inductive summarizing predicate describing tree segment

placeholder notation $\boxdot$ stands for the sequence of elements in the "missing subtree" of the segment. When composing $\mathbf{treeseg}_s$ and $\mathbf{tree}_s$ the analysis operations resolve sequences using such $\boxdot$ symbol. Based on this loop invariant, the analysis of the final few assignments of the insertion function produces an abstract state that implies the desired post-condition.

3 Abstract Domain for Sequences

In this section, we define the sequence abstract domain, including its elements and the constraints they denote, its concretization, and its main abstract operators.

3.1 Sequences abstraction

An element of the abstract domain of sequences is a conjunction of constraints over a finite set of symbolic variables that stand either for sequences of base values, for base values, or for sets of values. Beside sequence equalities and predicates like sortedness, we also consider numerical upper/lower bounds over the values in sequences and multi-set constraints over the collections of values in sequences.

Concrete states. Let $\mathbb{V}$ denote a set of values. Although $\mathbb{V}$ usually denotes a set of scalar values (including addresses), our only assumptions on $\mathbb{V}$ is that it has a total ordering $\preceq$ with extremal values $+\infty, -\infty$. Since our domain constrains both variables that range over $\mathbb{V}$ and variables that range over sequences of values in $\mathbb{V}$, we need several kinds of *symbolic variables.* In the following, we let symbols $\alpha, \alpha_0, \alpha'_0, \beta, \ldots \in \mathbb{X}_n$ denote *value symbolic variables*, namely, variables that stand for a value in $\mathbb{V}$. To express constraints on the set $\mathbb{V}^*$ of all the finite words on alphabet $\mathbb{V}$, we let a separate set $\mathbb{X}_s$, represent *sequence symbolic variables.* We note $S, S_1, S', P, \ldots \in \mathbb{X}_s$ such sequence variables. Finally, we write $\mathcal{M}(\mathbb{V})$ for the set of multisets of values in $\mathbb{V}$ and let $\mathbb{X}_m$ be the set of *multi-set valued symbolic variables.* Moreover, if $S \in \mathbb{X}_s$ is a sequence variable, we attach to it three numerical variables $\texttt{len}_S, \texttt{min}_S, \texttt{max}_S$ in $\mathbb{X}_n$ that respectively denote the length, minimum and maximum value of S, and that there exists a multi-set variable $\texttt{multi}_S \in \mathbb{X}_m$ that denotes the multi-set of its elements.

A *concrete state* comprises three functions that map each kind of symbolic variables to elements of the corresponding type. Due to the relationship between a

$$E \ ::= \ \texttt{[]} \mid \texttt{[}\alpha\texttt{]} \mid S \mid E.E \mid \mathbf{sort}(E) \qquad\qquad C\,(\in \mathbb{C}) \ \ ::= \ S = E$$

(a) Syntax of expressions (E) and constraints (C)

$$\begin{array}{rr} [\![\texttt{[]}]\!]_s(\sigma) = \varepsilon & [\![\texttt{[}\alpha\texttt{]}]\!]_s(\sigma) = \sigma_n(\alpha) \\ [\![S]\!]_s(\sigma) = \sigma_s(S) & [\![E_1.E_2]\!]_s(\sigma) = [\![E_1]\!]_s(\sigma).[\![E_2]\!]_s(\sigma) \end{array}$$

$$[\![\mathbf{sort}(E)]\!]_s(\sigma) = a_{\pi(1)} \ldots a_{\pi(n)} \text{ where } \begin{cases} [\![E]\!](\sigma) = a_1 \ldots a_n \\ \forall i \in [1, n-1], a_{\pi(i)} \preceq a_{\pi(i+1)} \\ \pi \text{ is a permutation of } [1, n] \end{cases}$$

$$(\sigma_n, \sigma_s) \models_s S = E \text{ iff } \sigma_s(S) = [\![E]\!]_s(\sigma_n, \sigma_s)$$

(b) Semantics

Fig. 5. Sequence expressions and constraints: syntax and semantics

sequence symbolic variable S and $\texttt{len}_S, \texttt{min}_S, \texttt{max}_S$, and $\texttt{multi}_S$, a concrete state is valid if and only if it maps these five variables into consistent objects. Formally:

Definition 1. *A* concrete state *is a tuple $\sigma = (\sigma_n, \sigma_m, \sigma_s)$ where the functions $\sigma_n : \mathbb{X}_n \to \mathbb{V}$, $\sigma_m : \mathbb{X}_m \to \mathcal{M}(\mathbb{V})$, $\sigma_s : \mathbb{X}_s \to \mathbb{V}^*$ are such that, for all S in $\mathbb{X}_s$,*

$$\begin{array}{r} \sigma_s(S) = a_1 \ldots a_n \Rightarrow \begin{cases} \ \ \sigma_n(\texttt{min}_S) = \min_i a_i \wedge \sigma_n(\texttt{max}_S) = \max_i a_i \\ \wedge\ \sigma_n(\texttt{len}_S) = n \wedge \sigma_m(\texttt{multi}_S) = \{a_1, \ldots, a_n\} \end{cases} \\ \sigma_s(S) = \varepsilon \Rightarrow \begin{cases} \ \ \sigma_n(\texttt{min}_S) = +\infty \wedge \sigma_n(\texttt{max}_S) = -\infty \\ \wedge\ \sigma_n(\texttt{len}_S) = 0 \wedge \sigma_m(\texttt{multi}_S) = \emptyset \end{cases} \end{array}$$

For short, given a state σ, we note its components σ_n, σ_m, and σ_s. We write Σ for the set of all such concrete states.

Abstract sequence constraints. The sequence abstract domain relies on expressions and constraints over symbolic variables. Their syntax is shown in Figure 5(a). An expression is either the empty sequence, or a sequence of length one that consists of a value symbolic variable, or a sequence symbolic variable, or a concatenation of expressions, or the sorting of a sequence expression returned by the function $\mathbf{sort} : E \to E$, (introduced in Section 2). Given a state σ, a sequence expression E evaluates into a sequence of values $[\![E]\!]_s(\sigma)$, as shown in Figure 5(b). Sequence constraints are *definition constraints* of the form $S = E$, as shown in Figure 5(a). Allowing only symbolic sequence variables in the left-hand side of equalities somewhat limits expressiveness but simplifies the machine representation of abstract elements. The semantics of constraints is defined based on a satisfaction relation $\models_s$ that is spelled out in Figure 5(b): we write $(\sigma_n, \sigma_s) \models_s C$ when constraint C holds in concrete state (σ_n, σ_s). We note $\mathbb{C}$ for the set of sequence constraints.

Parameter abstract domains. In the following, we assume two abstract domains are fixed, taken as parameters by the sequence abstraction. First, $\mathbb{D}^\sharp_n$ represents numerical constraints and provides a concretization function $\gamma_n : \mathbb{D}^\sharp_n \to \mathcal{P}(\mathbb{X}_n \to \mathbb{V})$. Possible choices for $\mathbb{D}^\sharp_n$ include intervals [17], octagons [46], or convex polyhedra [20] abstract domains. Second, $\mathbb{D}^\sharp_m$ represents multi-set constraints and provides a concretization function $\gamma_m : \mathbb{D}^\sharp_m \to \mathcal{P}(\mathbb{X}_m \to \mathcal{M}(\mathbb{V}))$. Our implementation uses a variation of the set domain of [39] that describes multi-set constraints.

Sequence abstraction. An abstract state consists either of a special element $\bot$ that denotes the empty set of concrete states or of a finite conjunction of sequence constraints together with numerical and multi-set constraints:

Definition 2 (Sequence abstraction). *The* abstract sequence domain $\mathbb{\Sigma}^\sharp$ *is defined as* $\{\bot\} \uplus \{(\sigma_n^\sharp, \sigma_m^\sharp, C_0 \wedge \ldots \wedge C_n) \mid \sigma_n^\sharp \in \mathbb{D}_n^\sharp, \sigma_m^\sharp \in \mathbb{D}_m^\sharp, C_0, \ldots, C_n \in \mathbb{C}\}$. *Furthermore, its concretization* $\gamma_{\mathbb{\Sigma}} : \mathbb{\Sigma}^\sharp \longrightarrow \mathcal{P}(\mathbb{\Sigma})$ *is defined by* $\gamma_{\mathbb{\Sigma}}(\bot) = \emptyset$ *and:*

$$\begin{aligned}&\gamma_{\mathbb{\Sigma}}(\sigma_n^\sharp, \sigma_m^\sharp, C_0 \wedge \ldots \wedge C_n) = \\ &\quad \big\{(\sigma_n, \sigma_m, \sigma_s) \,\big|\, \sigma_n \in \gamma_n(\sigma_n^\sharp) \wedge \sigma_m \in \gamma_m(\sigma_m^\sharp) \wedge \forall i,\ \sigma_n, \sigma_s \models_s C_i\ \big\}\end{aligned}$$

For consistency, we use $\sigma_s^\sharp$ as a generic notation for a finite conjunction of constraints $C_0 \wedge \ldots \wedge C_n$ and $\sigma^\sharp$ for a generic triple $(\sigma_n^\sharp, \sigma_m^\sharp, \sigma_s^\sharp)$. We remark that the empty conjunction of constraints concretizes into $\mathbb{\Sigma}$ thus we note it $\top$.

Machine representation. For the sake of algorithmic efficiency, we rely on an optimized machine representation for sequence constraints in non-bottom abstract states. First, we let equality constraints between variables be described by union-find data-structures, which enables the incremental computation of equality classes representatives. Emptiness constraints ($S = \texttt{[]}$) and sortedness constraints ($S = \mathbf{sort}(S)$) are marked by tags over sequence variables. Finally, other equality constraints are represented with a map data type, the keys of which are the left hand side variables. For instance, $S = \texttt{[}\alpha\texttt{]}$ boils down to a map entry $S \mapsto \texttt{[}\alpha\texttt{]}$.

3.2 Abstract operations

We now discuss abstract operations on sequence abstract states. In this subsection, we discuss two operations: $\mathfrak{guard}_{\mathbb{\Sigma}}$ refines an abstract sequence element into its conjunction with an additional constraint and $\mathfrak{verify}_{\mathbb{\Sigma}}$ attempts to discharge a sequence constraint (so as to, e.g., verify an assertion). We assume that the underlying domains also implement similar operators. For instance, we require the numerical domain to provide an operator $\mathfrak{guard}_n$ that inputs a numerical constraint and a $\sigma_n^\sharp \in \mathbb{D}_n^\sharp$ and refines the latter with that constraint.

Abstract sequence condition. First, we consider the *abstract sequence condition* operator $\mathfrak{guard}_{\mathbb{\Sigma}} : \mathbb{C} \times \mathbb{\Sigma}^\sharp \to \mathbb{\Sigma}^\sharp$ which refines an abstract state with an additional sequence constraint. While a naive implementation of $\mathfrak{guard}_{\mathbb{\Sigma}}(C, \sigma^\sharp)$ would simply add the constraint C to the conjunction $\sigma_s^\sharp$ component, this would be imprecise in general. Indeed, the conjunction $C \wedge \sigma_s^\sharp$ may be equivalent to $\bot$. Moreover, $C \wedge \sigma_s^\sharp$ may entail constraints that are strictly more precise than those in $\sigma_s^\sharp$.

At a high level, $\mathfrak{guard}_{\mathbb{\Sigma}}$ performs three kinds of operations:

1. *Compaction* simplifies constraints by rewriting the right hand side of definition constraints into the left hand side, wherever possible. For example, $S = S'.\texttt{[}\alpha\texttt{]}.S'' \wedge S_1 = S'.\texttt{[}\alpha\texttt{]}$ simplifies into $S = S_1.S'' \wedge S_1 = S'.\texttt{[}\alpha\texttt{]}$.
2. *Saturation* synthesizes additional numerical and multi-set constraints that can be derived from a newly added constraint. For instance, $S = S'.\texttt{[}\alpha\texttt{]}$ entails

that $\texttt{len}_S = 1 + \texttt{len}_{S'}$. Likewise, some constraints may entail that a sequence is empty. Another special kind of saturation occurs when the whole state can be reduced to $\perp$ as incompatible constraints are detected. As saturation is the most complex part of $\mathfrak{guard}_\Sigma$, we detail it below.

3. *Detection of cyclic constraints* prevents compaction and saturation from adding too many, redundant constraints, and it ensures the termination of algorithms iterating on definitions. We discuss this in Example 2.

We now discuss constraint saturation more in detail:

– The *length constraints saturation* derives numerical constraints from the equality of the length of both sides of a new definition constraint $S = E$. Indeed, such a constraint implies $\texttt{len}_S = \tau_{\texttt{len}}(E)$, which can be added to the $\sigma_n^\sharp$ component using $\mathfrak{guard}_n$, where $\tau_{\texttt{len}}$ is defined by:

$$\begin{array}{ccc} \tau_{\texttt{len}}(\texttt{[]}) = 0 & \tau_{\texttt{len}}(E.E') = \tau_{\texttt{len}}(E) + \tau_{\texttt{len}}(E') & \tau_{\texttt{len}}(S) = \texttt{len}_S \\ \tau_{\texttt{len}}(\texttt{[}\alpha\texttt{]}) = 1 & \tau_{\texttt{len}}(\textbf{sort}(E)) = \tau_{\texttt{len}}(E) & \end{array}$$

– The *multi-set contents constraints saturation* operates similarly, and derives multi-set equalities from definition constraints. Surely, $S = E$ entails $\texttt{multi}_S = \tau_{\texttt{mul}}(E)$, which can refine the $\sigma_m^\sharp$ part using $\mathfrak{guard}_m$ where $\tau_{\texttt{mul}}$ is defined by:

$$\begin{array}{ccc} \tau_{\texttt{mul}}(\texttt{[]}) = \emptyset & \tau_{\texttt{mul}}(E.E') = \tau_{\texttt{mul}}(E) \uplus \tau_{\texttt{mul}}(E') & \tau_{\texttt{mul}}(S) = \texttt{multi}_S \\ \tau_{\texttt{mul}}(\texttt{[}\alpha\texttt{]}) = \{\alpha\} & \tau_{\texttt{mul}}(\textbf{sort}(E)) = \tau_{\texttt{mul}}(E) & \end{array}$$

– The *detection of empty sequence variables* derives new definition constraints of the form $S = \texttt{[]}$ when either sequence constraints or numerical constraints entail the emptiness of S. For instance:
  - when $\sigma_s^\sharp$ contains constraints $S = \texttt{[]}$ and $S' = \texttt{[]}$, the constraint $S = S'.S''$ simplifies into $S = \texttt{[]}$;
  - when $\sigma_n^\sharp$ contains the constraint $\texttt{len}_S = 0$, then it follows that $S = \texttt{[]}$.

– The *detection of sorted sequence variables* do the same for constraints of the form $S = \textbf{sort}(S)$ thanks to definitions of S and to numerical inequalities:

$$\frac{S = S_1.\ldots.S_n \qquad \forall i, S_i = \textbf{sort}(S_i) \quad \forall i < j, \texttt{max}_{S_i} \le \texttt{max}_{S_j}}{S = \textbf{sort}(S)}$$

Such rule is very costly as it checks a quadratic amount of numerical inequalities. Nevertheless, relaxing the rule by only considering the case $j = i + 1$ is not sound, since sequence variables may be empty. Therefore, two consecutive elements in $\sigma_s(S)$ can come from non-consecutive sequence variables.

– The *extremal values inequalities saturation* derives numerical inequalities from a definition constraint $S = E$ by case analysis over the right hand side E, and can be summarized by a set of derivation rules. The rules below describe such reasoning steps:

$$\frac{S = E \qquad \alpha \in \textbf{fv}(E)}{\texttt{min}_S \le \alpha \le \texttt{max}_S} \qquad \frac{S = E \qquad S' \in \textbf{fv}(E)}{\texttt{min}_S \le \texttt{min}_{S'} \qquad \texttt{max}_{S'} \le \texttt{max}_S}$$

$$\frac{S = \texttt{[]} \qquad \alpha \in \mathbb{X}_n}{\texttt{max}_S < \alpha < \texttt{min}_S} \qquad \frac{S' = \textbf{sort}(S') \qquad S' = \ldots.\texttt{[}\alpha\texttt{]}.S.\ldots}{\alpha \le \texttt{min}_S}$$

As an example, the first rule states that numerical constraints can be derived from the knowledge that S is a concatenation of several components including a numerical variable α; in this case novel numeric constraints expressing that α is bounded by the extremal values of S can be added to $\sigma^\sharp_n$ using operator $\mathfrak{guard}_n$. Similarly, the second rule states that the extremal values of a sequence are bounded by the extremal values of any sequence containing it. The third rule states that an empty sequence supports arbitrary bounds. Finally, the fourth rule allows to reason over bounds when a sequence is known to be sorted.

– The *decomposition of equality constraints* synthesizes additional equality constraints that can be derived when two definition constraints $S = E_0$ and $S = E_1$ over the same name can be found in $\sigma^\sharp_s$. Indeed, when both E_0 and E_1 can be decomposed simultaneously, new equalities can be immediately derived:

$$\frac{\texttt{[}\alpha_0\texttt{]}.E_0 = \texttt{[}\alpha_1\texttt{]}.E_1}{\alpha_0 = \alpha_1 \qquad E_0 = E_1} \qquad\qquad \frac{S.E_0 = E_1 \qquad S = \texttt{[]}}{E_0 = E_1}$$

In less obvious decomposition cases, further constraints can still be derived with the help of the numerical constraints. Indeed:

$$\frac{S_0.E_0 = S_1.E_1 \qquad \texttt{len}_{S_0} = \texttt{len}_{S_1}}{S_0 = S_1 \qquad E_0 = E_1}$$

Obviously, this inference may take place only when $\texttt{len}_{S_0} = \texttt{len}_{S_1}$ can be proved in the numerical domain.

A special case of saturation occurs when incompatible constraints are detected. Then, the whole abstract state is reduced to $\perp$, following the principles of reduced product [18]. As an example, when the abstract state contains the constraints $S = \texttt{[}\alpha\texttt{]}$ and $\texttt{len}_S = 0$, such a reduction is performed.

To summarize, the computation of $\mathfrak{guard}_\Sigma(C, \sigma^\sharp)$ involves the addition to $\sigma^\sharp$ of a set of constraints that are derived from C. It is conservative in general. The termination of this computation follows from the fact that the added constraints only involve syntactic subcomponents of the elements of C and $\sigma^\sharp_s$.

Example 1. We consider the abstract state of Figure 3(b) and the constraint $S_1 = S_l.\texttt{[}\delta\texttt{]}$ where S_1 is a new symbolic sequence variable. First, the constraint is added to the abstract state. Second, compaction replaces the pattern $S_l.\texttt{[}\delta\texttt{]}$ with S_1 in all other constraints. Third, the numerical inequality $\alpha \geq \texttt{max}_{S_1}$ and the sortedness of S_1 entails that S_1 is sorted. Then, $S_1 = \mathbf{sort}(S_1)$ implies that δ is the maximum value of S_1. Moreover, the fact that S_l is a subsequence of S_1 entails that $\texttt{min}_{S_1} \leq \texttt{min}_{S_l}$ and $\texttt{max}_{S_l} \leq \texttt{max}_{S_1}$. Finally, since $\delta \leq \texttt{i}$, $\mathfrak{guard}_\Sigma$ also derives $\texttt{max}_{S_1} \leq \texttt{i}$. Finally, $\mathfrak{guard}_\Sigma$ produces:

$$\begin{array}{c} S = S_1.S_r \wedge S_1 = S_l.\texttt{[}\delta\texttt{]} \wedge S = \mathbf{sort}(S) \wedge S_i = \mathbf{sort}(S_i),\ i \in \{l, r, 1\} \\ \wedge\ \texttt{max}_{S_l} \leq \texttt{max}_{S_1} = \delta \leq \texttt{max}_{S_r} \wedge \texttt{min}_{S_1} \leq \texttt{max}_{S_l} \wedge \delta \leq \texttt{i} \wedge \alpha_0, \alpha_1 \neq \mathbf{0x0} \end{array}$$

Example 2. In this example, we show the detection of mutually cyclic constraints. We consider the abstract state $S_1 = S_2.S' \wedge S_2 = S''.S_3$, and the addition of $S_3 = S_1.S'''$. Inlining definition constraints for S_2 and S_3 would produce the cyclic constraint $S_1 = S''.S_1.S'''.S'$. Thus, this also implies that S', S'', S''' are empty and that S_1, S_2 and S_3 are equal. After removal of the cycle, $\mathfrak{guard}_{\mathbb{\Sigma}}$ produces:

$$S' = S'' = S''' = \texttt{[]} \wedge S_1 = S_2 = S_3 \wedge S_1 = S_2.S' \wedge S_2 = S''.S_3$$

Theorem 1 (Soundness of $\mathfrak{guard}_{\mathbb{\Sigma}}$). *For all abstract state $\sigma^\sharp$ and constraint C, we have $\{\sigma \in \gamma_{\mathbb{\Sigma}}(\sigma^\sharp) \mid \sigma \models_s C\} \subseteq \gamma_{\mathbb{\Sigma}}(\mathfrak{guard}_{\mathbb{\Sigma}}(C, \sigma^\sharp))$.*

Verification of a sequence constraint. Second, we define the *constraint verification operator* $\mathfrak{verify}_{\mathbb{\Sigma}} : \mathbb{\Sigma}^\sharp \times \mathbb{C} \rightarrow \{\mathbf{false}, \mathbf{true}\}$ which inputs a constraint C and an abstract state $\sigma^\sharp$ and returns **true** when it can prove that $\sigma^\sharp$ entails C. It is conservative in the sense that it may return **false** even when the constraint is satisfied. The computation of $\mathfrak{verify}_{\mathbb{\Sigma}}(C, (\sigma^\sharp_n, \sigma^\sharp_m, \sigma^\sharp_s))$ proceeds as follows:

1. If $\sigma^\sharp_s$ is $\perp$, it returns **true**.
2. For definition constraints $S = E$, $\mathfrak{verify}_{\mathbb{\Sigma}}$ inlines the definitions of variables, and returns **true** when both sides rewrite into syntactically equal expressions. The absence of cyclic constraints ensures this exploration terminates.
3. Otherwise, it returns **false**.

For constraints of the form $S = \mathbf{sort}(E)$, the operator uses a specific rule (shown below) since variables inside the **sort** function may be arbitrarily reordered. Instead, we take advantage of the multi-set abstract domain to establish that S and E have the same contents.

$$\frac{S = \mathbf{sort}(S) \qquad \texttt{multi}_S = \tau_{\texttt{mul}}(E)}{S = \mathbf{sort}(E)}$$

Theorem 2 (Soundness of $\mathfrak{verify}_{\mathbb{\Sigma}}$). *For all abstract state $\sigma^\sharp$ and constraint C, if $\mathfrak{verify}_{\mathbb{\Sigma}}(\sigma^\sharp, C) = \mathbf{true}$ then, we have $\gamma_{\mathbb{\Sigma}}(\sigma^\sharp) \subseteq \{\sigma \in \mathbb{\Sigma} \mid \sigma \models_s C\}$.*

3.3 Lattice operations

We now discuss join, widening and inclusion checking operations for loop analysis. We assume that $\mathbb{D}^\sharp_n$ provides a conservative inclusion test operator $\mathfrak{is\_le}_n$ (it inputs two elements of $\sigma^\sharp_n$ and returns **true** only when it succeeds proving the first is included in the second), an over-approximate join operator $\mathfrak{join}_n$ and a widening $\mathfrak{widen}_n$, and that $\mathbb{D}^\sharp_m$ provides similar operators $\mathfrak{is\_le}_m$, $\mathfrak{join}_m$, and $\mathfrak{widen}_m$, and we build similar operators for $\mathbb{\Sigma}^\sharp$.

Inclusion checking. The inclusion test operator inputs two abstract states and returns a boolean. When it returns **true**, the concretization of the first abstract state is included into that of the second one. The inclusion checking algorithm is based on the constraint representation of abstract states and boils down to a repeated application of $\mathfrak{verify}_{\mathbb{\Sigma}}$.

Definition 3 (Inclusion checking operator). *The operator $\mathfrak{is\_le}_{\mathbb{\Sigma}} : \mathbb{\Sigma}^\sharp \times \mathbb{\Sigma}^\sharp \rightarrow \{\mathbf{true}, \mathbf{false}\}$ is defined by:*

$$\begin{aligned}&\mathfrak{is\_le}_{\mathbb{\Sigma}}((\sigma^\sharp_{n,0}, \sigma^\sharp_{m,0}, \sigma^\sharp_{s,0}), (\sigma^\sharp_{n,1}, \sigma^\sharp_{m,1}, \wedge_i C_i)) \\ &\quad := \mathfrak{is\_le}_n(\sigma^\sharp_{n,0}, \sigma^\sharp_{n,1}) \wedge \mathfrak{is\_le}_m(\sigma^\sharp_{m,0}, \sigma^\sharp_{m,1}) \wedge (\wedge_i \mathfrak{verify}_{\mathbb{\Sigma}}(C_i, \sigma^\sharp_{s,0}))\end{aligned}$$

Theorem 3. *The operator* $\mathfrak{is\_le}_\Sigma$ *is sound in the sense that, for all* $\sigma_0^\sharp, \sigma_1^\sharp \in \Sigma^\sharp$*, if* $\mathfrak{is\_le}_\Sigma(\sigma_0^\sharp, \sigma_1^\sharp) = \mathbf{true}$*, then* $\gamma_\Sigma(\sigma_0^\sharp) \subseteq \gamma_\Sigma(\sigma_1^\sharp)$*.*

Upper bounds. As usual, we define two over-approximate upper-bound operators, namely, a classical join operator $\mathfrak{join}_\Sigma : \Sigma^\sharp \times \Sigma^\sharp \to \Sigma^\sharp$ and a widening $\mathfrak{widen}_\Sigma : \Sigma^\sharp \times \Sigma^\sharp \to \Sigma^\sharp$ that ensures termination.

Essentially, the $\mathfrak{join}_\Sigma$ operator proceeds component-wise (like $\mathfrak{is\_le}_\Sigma$ as defined in Definition 3) and essentially preserves sequence constraints that appear in both arguments. In the case of definition constraint, it first saturates the conjunctions of constraints, so as to maximize the possible sets of common constraints. The algorithm of $\mathfrak{widen}_\Sigma$ is similar, except that it does not saturate its left argument for the sake of termination. This implies that $\mathfrak{widen}_\Sigma$ always returns a conjunction of constraints that forms a subset of the constraints of its left argument.

Both operators are sound and furthermore, $\mathfrak{widen}_\Sigma$ guarantees termination.

Theorem 4 (Soundness of $\mathfrak{join}_\Sigma$ and $\mathfrak{widen}_\Sigma$, termination of $\mathfrak{widen}_\Sigma$). *For all abstract states* $\sigma_0^\sharp$*,* $\sigma_1^\sharp$*, we have:* $\gamma_\Sigma(\sigma_0^\sharp) \cup \gamma_\Sigma(\sigma_1^\sharp) \subseteq \gamma_\Sigma(\mathfrak{join}_\Sigma(\sigma_0^\sharp, \sigma_1^\sharp))$ *and* $\gamma_\Sigma(\sigma_0^\sharp) \cup \gamma_\Sigma(\sigma_1^\sharp) \subseteq \gamma_\Sigma(\mathfrak{widen}_\Sigma(\sigma_0^\sharp, \sigma_1^\sharp))$*. Moreover, the operator* $\mathfrak{widen}_\Sigma$ *ensures termination: for all sequence* $(\sigma_n^\sharp)_{n\in\mathbb{N}}$ *of abstract states the sequence* $((\sigma^\sharp)'_n)_{n\in\mathbb{N}}$ *defined by* $(\sigma^\sharp)'_0 = \sigma_0^\sharp$ *and* $(\sigma^\sharp)'_{n+1} = \mathfrak{widen}_\Sigma((\sigma^\sharp)'_n, \sigma_{n+1}^\sharp)$ *is ultimately stationary.*

Example 3 (Join). In this example, we consider the computation of the join of two abstract states taken from the analysis of the program of Figure 2. The analysis of the loop at line 7 involves the computation of the join of the three abstract states below. For concision, we omit inequality constraints involving extremal values of empty sequences.

$$\sigma_0^\sharp ::= \begin{cases} \quad S = S_0 \wedge S_1 = S_2 = \texttt{[]} \\ \wedge\ S = \mathbf{sort}(S) \wedge S_i = \mathbf{sort}(S_i), i \in \{0,1,2\} \end{cases}$$
$$\sigma_1^\sharp ::= \begin{cases} \quad S = S_0.S_2 \wedge S_1 = \texttt{[]} \wedge \texttt{max}_{S_0} \leq \texttt{min}_{S_2} \wedge \texttt{i} \leq \texttt{min}_{S_2} \\ \wedge\ S = \mathbf{sort}(S) \wedge S_i = \mathbf{sort}(S_i), i \in \{0,1,2\} \end{cases}$$
$$\sigma_2^\sharp ::= \begin{cases} \quad S = S_1.S_0 \wedge S_2 = \texttt{[]} \wedge \texttt{max}_{S_1} \leq \texttt{min}_{S_0} \wedge \texttt{max}_{S_1} \leq \texttt{i} \\ \wedge\ S = \mathbf{sort}(S) \wedge S_i = \mathbf{sort}(S_i), i \in \{0,1,2\} \end{cases}$$

The most notable step is the saturation of the first argument, that injects constraint $S = S_1.S_0.S_2$, as a consequence of $S = S_0$ and $S_1 = S_2 = \texttt{[]}$ in $\sigma_0^\sharp$, $S = S_0.S_2$ and $S_1 = \texttt{[]}$ in $\sigma_1^\sharp$ and $S = S_1.S_0$ and $S_2 = \texttt{[]}$ in $\sigma_2^\sharp$. After this, constraints that hold in only either argument are dropped, as, e.g., constraint $S_1 = \texttt{[]}$ in $\sigma_1^\sharp$. The result of the union corresponds to the abstract state in Figure 3(c).

4 Combination of sequence abstraction and shape analysis

In this section, we define a shape analysis with inductive predicates that infers invariants about both the layout of data-structures and the sequences of values they store. For the sake of simplicity, we consider only a singly-linked list predicate

```
struct list { struct list* n; int d; };
```

$$
\begin{array}{l}
\mathbf{lseg}_s(\alpha_0, \alpha_1, S \boxdot\,) := \\
\;\mid \mathbf{emp} \wedge \alpha_0 = \alpha_1 \wedge S = \texttt{[]} \\
\;\mid \exists\, \alpha', \delta, S',\ \alpha_0.\texttt{n} \mapsto \alpha' * \alpha_0.\texttt{d} \mapsto \delta * \mathbf{lseg}_s(\alpha', \alpha_1, S' \boxdot\,) \wedge \alpha_0 \neq \mathbf{0x0} \wedge S = \texttt{[}\delta\texttt{]}.S'
\end{array}
$$

Fig. 6. A `C` list data-type and the inductive summarizing predicate describing list segments

(Figure 6) throughout this section, although our analysis and its implementation are parameterized by user-defined inductive predicates [14,15]. The generalization to other structures will be discussed in Section 5.

4.1 Language and semantics

Although our implementation is based on the MemCAD analyzer [38] and targets the C language, our formalization only considers a restricted fragment. We let $\mathbb{X}$ denote a finite set of program variables. We consider a basic imperative language, where commands are assignments, conditional statements, loops, and sequences of commands. Expressions are either l-values that evaluate to addresses, or r-values, that evaluate to scalars. An l-value l is either a program variable $\texttt{v} \in \mathbb{X}$, the access to an l-value field $l.\texttt{f}$ (for concision, we let $\texttt{f}$ denote both the field name and the corresponding memory offset), or the dereference $*e$ of an expression e. An r-value e is either a constant $n \in \mathbb{V}$, or the reading of the memory cell defined by an l-value l, or the address $\&l$ or an l-value l, or the application $e_0 \oplus e_1$ of a binary operator to two sub-expressions. For simplicity, we assume here that operators are deterministic and cause no errors. The grammar is shown below:

$$l ::= \texttt{v} \mid l.\texttt{f} \mid *e \quad e ::= n \mid l \mid \&l \mid e \oplus e \quad c ::= l = e \mid \mathbf{if}(e)\{c\} \mid \mathbf{while}(e)\{c\} \mid c; c$$

We note $\mathbb{A}$ for the set of addresses, which is a subset of the set of values $\mathbb{V}$. A memory state m is a partial function from addresses to values. We note $\mathbb{M}$ for the set of memory states and let $\emptyset$ denote the empty memory. Furthermore, we assume that each program variable $\texttt{x}$ has a fixed address denoted by $\underline{\texttt{x}} \in \mathbb{A}$. Based on these definitions, we set up the program semantics as follows. First, we define the semantics of expressions by induction over their syntax. The semantics of an l-value l is a function $[\![l]\!]_l : \mathbb{M} \to \mathbb{A}$ that maps a memory state m to the address l evaluates to in m. Similarly, the semantics $[\![e]\!]_e : \mathbb{M} \to \mathbb{V}$ of an expression e maps a memory state to a value. Finally, the semantics $[\![c]\!] : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M})$ of a command c maps any set of input memory states M to the set of all possible output memory states when starting from any $m \in M$. The definition of all three semantics is classical and shown in Figure 7, where $f_\oplus : \mathbb{V}^2 \to \mathbb{V}$ denotes the semantics of operator $\oplus$.

4.2 Combined memory and sequence abstraction

Sequence aware shape abstraction. We start with the definition of abstract memory predicates, following an approach similar to that of separation logic based shape analyses with inductive definitions [12,14], extended with sequence information.

$$\begin{array}{rclrcl}
[\![\mathtt{x}]\!]_l(m) & := & \underline{\mathtt{x}} & [\![n]\!]_e(m) & := & n \\
[\![l.\mathtt{f}]\!]_l(m) & := & [\![l]\!]_l(m) + \mathtt{f} & [\![l]\!]_e(m) & := & m([\![l]\!]_l(m)) \\
[\![* e]\!]_l(m) & := & [\![e]\!]_e(m) & [\![\&l]\!]_e(m) & := & [\![l]\!]_l(m) \\
 & & & [\![e_0 \oplus e_1]\!]_e(m) & := & f_\oplus([\![e_0]\!]_e(m), [\![e_1]\!]_e(m))
\end{array}$$

$$\begin{array}{rcl}
[\![l = e]\!](M) & := & \{m[[\![l]\!]_l(m) \mapsto [\![e]\!]_e(m)] \mid m \in M\} \\
[\![\mathbf{if}(e)\{c_0\}]\!](M) & := & [\![c_0]\!]\left(\{m \in M \mid [\![e]\!]_e(m) \neq 0\}\right) \cup \ \{m \in M \mid [\![e]\!]_e(m) = 0\} \\
[\![\mathbf{while}(e)\{c\}]\!](M) & := & \{m \in \mathbf{lfp}F \mid [\![e]\!]_e(m) = 0\} \\
 & & \text{where } F(M') = M \cup [\![c]\!](m \in M' \mid [\![e]\!]_e(m) \neq 0\}) \\
[\![c_0; c_1]\!](M) & := & [\![c_1]\!] \circ [\![c_0]\!](M)
\end{array}$$

Fig. 7. Semantics of programs

As explained early in the section, our formalization considers a single inductive predicate describing list segments, and parameterized with a symbolic sequence variable that stands for the sequence of the values contained in them (Figure 6). Considering only list segments has two advantages. First, complete lists can be expressed as list segments the last element of which has a "next" field equal to **0x0**. Second, it simplifies reasoning over sequences as it avoids branching structures (considered in Section 5). Abstract states rely on scalar symbolic variables in $\mathbb{X}_n$ to denote values and addresses and consist of separating conjunctions [50] of points-to predicates and of list segment predicates:

Definition 4 (Abstract memory states). *The set of* abstract memory states $\mathbb{M}^\sharp$ *is described by the grammar below, where* $\alpha_0, \alpha_1 \in \mathbb{X}_n$ *and* $S \in \mathbb{X}_s$*:*

$$m^\sharp ::= \mathbf{emp} \mid m^\sharp * m^\sharp \mid \alpha_0.\mathtt{f} \mapsto \alpha_1 \mid \mathbf{lseg}_s(\alpha_0, \alpha_1, S \boxdot)$$

We note $\mathbb{M}^\sharp$ *for the set of abstract memory states.*

As usual, **emp** denotes the empty memory region and $m_0^\sharp * m_1^\sharp$ denotes the disjoint union of memory regions described by $m_0^\sharp$ (resp., $m_1^\sharp$). The abstract predicate $\alpha_0.\mathtt{f} \mapsto \alpha_1$ denotes a single memory cell, the address of which is described by α_0 plus the offset of $\mathtt{f}$ and the contents of which is described by α_1. Finally, $\mathbf{lseg}_s(\alpha_0, \alpha_1, S)$ stands for a (possibly empty) list segment that starts at an address described by α_0, ending with a pointer to address α_1, where each list element consists of two fields, namely, a pointer to the next element and a data field, and such that the sequence of the values of the data fields is described by sequence variable S. In logical terms, the predicate $\mathbf{lseg}_s(\alpha_0, \alpha_1, S)$ is defined inductively as shown in Figure 6.

As the definition of $\mathbf{lseg}_s$ in Figure 6 shows, the concretization of abstract memory states indirectly involves sequence variables (and also multi-set variables). Indeed, given an abstract memory state $m^\sharp$ and a sequence variable S that appears in $m^\sharp$, the concretization of $m^\sharp$ also constrains S, $\mathtt{len}_S$, and $\mathtt{multi}_S$. To reflect this, we let the concretization of an abstract memory $m^\sharp$ return a set of tuples that comprise not only a memory state m, but also a valuation that maps each symbolic variable in $m^\sharp$ to a value of the corresponding type (scalar, multi-set, or sequence). Such a valuation boils down to a triple $(\sigma_n, \sigma_m, \sigma_s)$ (Definition 2). The definition of the concretization is based on a set of inductive derivation rules that follow the syntax of abstract memories and unfold the list segment predicates (Figure 8).

$$\frac{}{\emptyset, \sigma \models_{\mathbb{M}} \mathbf{emp}} \qquad \frac{m = [\sigma_n(\alpha_0) + \mathtt{f} \mapsto \sigma_n(\alpha_1)]}{m, (\sigma_n, \sigma_m, \sigma_s) \models_{\mathbb{M}} \alpha_0.\mathtt{f} \mapsto \alpha_1} \qquad \frac{\forall i,\ m_i, \sigma \models_{\mathbb{M}} m_i^\sharp}{m_0 \uplus m_1, \sigma \models_{\mathbb{M}} m_0^\sharp * m_1^\sharp}$$

$$\frac{\sigma_n(\alpha_0) = \sigma_n(\alpha_1) \qquad \sigma_n, \sigma_s \models_s S = \mathtt{[]}}{\emptyset, (\sigma_n, \sigma_m, \sigma_s) \models_{\mathbb{M}} \mathbf{lseg}_s(\alpha_0, \alpha_1, S)}$$

$$\frac{\begin{array}{c} m, (\sigma_n, \sigma_m, \sigma_s) \models_{\mathbb{M}} \alpha_0.\mathtt{n} \mapsto \alpha_2 * \alpha_0.\mathtt{d} \mapsto \alpha_3 * \mathbf{lseg}_s(\alpha_2, \alpha_1, S_1 \boxdot\ ) \\ \sigma_n(\alpha_0) \neq 0 \qquad \sigma_n, \sigma_s \models_s S = \mathtt{[}\alpha_3\mathtt{]}.S_1 \qquad S_1 \text{ fresh} \end{array}}{m, (\sigma_n, \sigma_m, \sigma_s) \models_{\mathbb{M}} \mathbf{lseg}_s(\alpha_0, \alpha_1, S \boxdot\ )}$$

Fig. 8. Concretization of abstract memory states

Definition 5 (Concretization of abstract memory states). *The* concretization of abstract memory states $\gamma_{\mathbb{M}}$ *maps an abstract memory* $m^\sharp$ *to a set of pairs* $(m, \sigma) \in \mathbb{M} \times \Sigma$ *and is defined by:*

$$\gamma_{\mathbb{M}}(m^\sharp) = \{(m, \sigma) \mid (m, \sigma) \models_{\mathbb{M}} m^\sharp\}$$

As examples of abstract memory states, we refer the reader to the left conjuncts of the three abstract states shown in Figure 3.

Combined abstract domain. The analysis needs to reason accurately over sequence variables not only when they are bound in an inductive predicate, but also when these predicates are unfolded. Thus, it requires a product abstract domain based on the memory abstract domain fixed in Definition 4 and Definition 5 and on the sequence abstract domain introduced in Section 3. Moreover, like most shape analyses, it sometimes needs to make case splits due to the disjunctive nature of the inductive predicate $\mathbf{lseg}_s$. Thus, the combined abstraction is defined as follows:

Definition 6 (Combined abstraction). *The elements of the* combined state abstract domain $\mathbb{S}^\sharp$ *are finite disjunctions of pairs of the form* $(m^\sharp, \sigma^\sharp) \in \mathbb{M}^\sharp \times \Sigma^\sharp$. *Furthermore, the concretization* $\gamma_{\mathbb{S}}$ *maps an element* $s^\sharp$ *of* $\mathbb{S}^\sharp$ *into a set of memories* m *and is defined by:*

$$\gamma_{\mathbb{S}}((m^\sharp, \sigma^\sharp)) := \{m \mid \exists \sigma \in \gamma_{\Sigma}(\sigma^\sharp), (m, \sigma) \in \gamma_{\mathbb{M}}(m^\sharp)\} \quad \gamma_{\mathbb{S}}(\textstyle\bigvee_i s_i^\sharp) := \textstyle\bigcup_i \gamma_{\mathbb{S}}(s_i^\sharp)$$

Concatenating segments. Before we move to the analysis algorithms, we discuss a principle for logical reasoning over segments that many analysis operations rely on. Intuitively, a pair of consecutive segments may be merged into a single segment, that stores a sequence of elements that is the concatenation of the elements in the two initial segments. Reciprocally, it is possible to split a segment based on a partition of the sequence of its elements. The lemma below formalizes this.

Lemma 1 (Concatenation (list predicates)). *We assume* $\alpha_0, \alpha_1, \alpha_2$ *distinct symbolic variables and let* $m_0^\sharp := \mathbf{lseg}_s(\alpha_0, \alpha_1, S_1 \boxdot\ ) * \mathbf{lseg}_s(\alpha_1, \alpha_2, S_2 \boxdot\ )$, $m_1^\sharp := \mathbf{lseg}_s(\alpha_0, \alpha_2, S \boxdot\ )$, *and* $\sigma^\sharp := S = S_1.S_2$. *Then, we have (i)* $\gamma_{\mathbb{S}}(m_0^\sharp, \sigma^\sharp) \subseteq \gamma_{\mathbb{S}}(m_1^\sharp, \sigma^\sharp)$, *and (ii) if* $(m, \sigma_1) \in \gamma_{\mathbb{S}}(m_1^\sharp, \sigma^\sharp)$, *then there exists* σ_0 *such that* $(m, \sigma_0) \in \gamma_{\mathbb{S}}(m_0^\sharp, \sigma^\sharp)$ *and, for all* $\beta \in \mathbb{V}$ *such that* $\beta \neq \alpha_1$, $\sigma_0(\beta) = \sigma_1(\beta)$.

4.3 Computation of abstract post-conditions

Abstract post-conditions are computed by a pair of families of functions:

– given l-value l and expression e, $\mathfrak{assign}_{\mathbb{S},l=e} : \mathbb{S}^\sharp \to \mathbb{S}^\sharp$ computes an over-approximation for the assignment command $l = e$;

– given expression e, $\mathfrak{guard}_{\mathbb{S},e} : \mathbb{S}^\sharp \to \mathbb{S}^\sharp$ computes an over-approximation for the effect of the condition expression e.

In the following paragraphs, we give the main steps of the algorithms to compute them. They both ensure the soundness conditions that state that, for all l-value l, expression e, and abstract state $s^\sharp \in \mathbb{S}^\sharp$, we have $[\![l = e]\!](\gamma_{\mathbb{S}}(s^\sharp)) \subseteq \gamma_{\mathbb{S}}(\mathfrak{assign}_{\mathbb{S},l=e}(s^\sharp))$ and $\{m \in \gamma_{\mathbb{S}}(s^\sharp) \mid [\![e]\!]_e(m) \neq 0\} \subseteq \gamma_{\mathbb{S}}(\mathfrak{guard}_{\mathbb{S},e}(s^\sharp))$.

Simple cases. The computation of post-conditions for assignments and tests that involve only fully exposed cells is straightforward and follows classical shape analysis techniques [15]. For instance:

$$\begin{aligned}
&\mathfrak{assign}_{\mathbb{S},\mathtt{x.f=y}}(\underline{\mathtt{x}}.\mathtt{f} \mapsto \alpha_0 * \underline{\mathtt{y}} \mapsto \alpha_1 * m^\sharp, (\sigma_n, \sigma_m, \sigma_s)) \\
&\quad = (\underline{\mathtt{x}}.\mathtt{f} \mapsto \alpha_1 * \underline{\mathtt{y}} \mapsto \alpha_1 * m^\sharp, (\sigma_n, \sigma_m, \sigma_s)) \\
&\mathfrak{guard}_{\mathbb{S},\mathtt{x.f}\neq\mathtt{0x0}}(\underline{\mathtt{x}}.\mathtt{f} \mapsto \alpha_0 * m^\sharp, (\sigma_n, \sigma_m, \sigma_s)) \\
&\quad = (\underline{\mathtt{x}}.\mathtt{f} \mapsto \alpha_0 * m^\sharp, (\mathfrak{guard}_n(\alpha_0 \neq 0, \sigma_n), \sigma_m, \sigma_s))
\end{aligned}$$

where $\mathfrak{guard}_n$ denotes a sound condition test for the numerical domain [20].

Unfolding inductive predicates. The more difficult cases in post-conditions arise when some of the memory cells that are affected by the statement are summarized as part of an inductive predicate as, e.g., in $\mathfrak{assign}_{\mathbb{S},\mathtt{x=x.n}}(\underline{\mathtt{x}} \mapsto \alpha_0 * \mathbf{lseg}_s(\alpha_0, \alpha_1, S\boxdot))$. In such cases, some inductive predicates need to be *unfolded*, before falling back to the simpler situation shown in the two aforementioned cases.

The unfolding operation is based on rewriting rules that follow directly from the inductive nature of $\mathbf{lseg}_s$. We note $\rightsquigarrow$ the unfolding relation that rewrites an abstract state into another. Basic cases of $\rightsquigarrow$ proceed as follows:

$$\begin{aligned}
&(\mathbf{lseg}_s(\alpha_0, \alpha_1, S\boxdot) * m^\sharp, (\sigma_n^\sharp, \sigma_m^\sharp, \sigma_s^\sharp)) \\
&\rightsquigarrow \left\{ \begin{array}{l}
\quad (m^\sharp, \mathfrak{guard}_{\Sigma}(S = \mathtt{[]}, \mathfrak{guard}_n(\alpha_0 = \alpha_1, \sigma_n^\sharp), \sigma_m^\sharp, \sigma_s^\sharp)) \\
\vee \left(\begin{array}{l} \alpha_0.\mathtt{n} \mapsto \alpha_2 * \alpha_0.\mathtt{d} \mapsto \alpha_3 * \mathbf{lseg}_s(\alpha_2, \alpha_1, S_1\boxdot) * m^\sharp, \\ \qquad \mathfrak{guard}_{\Sigma}(S = \mathtt{[}\alpha_3\mathtt{]}.S_1, \mathfrak{guard}_n(\alpha_0 \neq 0, \sigma_n^\sharp), \sigma_m^\sharp, \sigma_s^\sharp) \end{array} \right)
\end{array} \right.
\end{aligned}$$

where α_2, α_3, S_1 are fresh. Unfolding is proved sound by the rules of Figure 8 in the sense that, for all $s_0^\sharp, s_1^\sharp \in \mathbb{S}^\sharp$, if $s_0^\sharp \rightsquigarrow s_1^\sharp$, then $\gamma_{\mathbb{S}}(s_0^\sharp) \subseteq \gamma_{\mathbb{S}}(s_1^\sharp)$.

The soundness of $\mathfrak{assign}_{\mathbb{S},\cdot}$ and $\mathfrak{guard}_{\mathbb{S},\cdot}$ follows from that of the unfolding relation, from that of the assignment and condition test of the underlying abstract domains, and from the (straightforward) handling of the unfolded cases.

We remark that the main difference compared to baseline shape analyses is that unfolding produces additional predicates about the sequence variables, which are added into the sequence domain. In turn, the addition of these constraints may yield increased precision due to internal reduction.

4.4 Computation of lattice operations

The lattice operations required for the analysis of loops comprise the conservative inclusion test and the over-approximation of concrete upper bounds. Moreover, the former is used in the definition of the latter. Again, the algorithms to compute them are based on those of classical shape analyses. Thus, we emphasize the extensions that are required to infer sequence information and refer the reader to [15] for a full description of shape abstraction inclusion and widening algorithms.

Inclusion checking. The inclusion test function performs a proof search to try to establish inclusion. Although the rule system actually used is more complex, the inclusion proof system can be summarized down to three basic principles. First, when two abstract states have the same abstract memory component, proving inclusion boils down to checking the inclusion in $\mathbb{\Sigma}^\sharp$. Second, when the left-hand side contains several inductive predicate instances that can be summarized into one in the right-hand side, the analysis tries to concatenate them using Lemma 1. Third, when the right-hand side can be unfolded and the left-hand side is included into one of the unfolded disjuncts, then the inclusion holds for the initial pair. The rules below formalize these three principles.

$$\frac{\mathfrak{is\_le}_{\mathbb{\Sigma}}(\sigma_l^\sharp, \sigma_r^\sharp) = \mathbf{true}}{(m^\sharp, \sigma_l^\sharp) \sqsubseteq (m^\sharp, \sigma_r^\sharp)}$$

$$\frac{\mathfrak{verify}_{\mathbb{\Sigma}}(\sigma_l^\sharp, S = S_1.S_2) = \mathbf{true}}{(\mathbf{lseg}_s(\alpha, \beta, S_1 \boxdot\ ) * \mathbf{lseg}_s(\beta, \delta, S_2 \boxdot\ ) * m_l^\sharp, \sigma_l^\sharp) \sqsubseteq (\mathbf{lseg}_s(\alpha, \delta, S \boxdot\ ), \sigma_r^\sharp)}$$

$$\frac{s_r^\sharp \rightsquigarrow \vee_i \overbrace{(m_i^\sharp, \mathfrak{guard}_{\mathbb{\Sigma}}(\sigma_i^\sharp, C_i))}^{s_i^\sharp} \qquad \exists j, \mathfrak{verify}_{\mathbb{\Sigma}}(C_j, \sigma_l^\sharp) = \mathbf{true} \wedge (m_l^\sharp, \sigma_l^\sharp) \sqsubseteq s_j^\sharp}{(m_l^\sharp, \sigma_l^\sharp) \sqsubseteq s_r^\sharp}$$

The $\mathfrak{is\_le}_{\mathbb{S}}$ function takes two abstract states and attempts to construct a proof tree that establishes inclusion based on these principles. The main specificities of the product with a sequence abstract domain are the requirement for $\mathfrak{is\_le}_{\mathbb{S}}$ to track sequence concatenation constraints and the use of the inclusion checking function of the sequence abstract domain. The soundness of $\mathfrak{is\_le}_{\mathbb{S}}$ follows from the soundness of the shape inclusion algorithm and of the underlying domains operations:

Theorem 5 (Soundness of $\mathfrak{is\_le}_{\mathbb{S}}$). *For all $s_0^\sharp, s_1^\sharp \in \mathbb{S}^\sharp$, if $\mathfrak{is\_le}_{\mathbb{S}}(s_0^\sharp, s_1^\sharp) = \mathbf{true}$ then $\gamma_{\mathbb{S}}(s_0^\sharp) \subseteq \gamma_{\mathbb{S}}(s_1^\sharp)$.*

Join and Widening. The cases of join and widening are more subtle, since these operators may need to introduce $\mathbf{lseg}_s$ predicates together with fresh symbolic sequence variables, and to infer accurate relations over these new variables. Indeed, these algorithms are based on the following two principles:
- when the memory components of the two arguments are equal, we use it for the shape specific part of the result;

$$
\begin{array}{rl}
[\![l = e]\!]^\sharp(s^\sharp) := \mathfrak{assign}_{\mathbb{S},l=e}(s^\sharp) & \quad [\![c_0; c_1]\!]^\sharp(s^\sharp) := [\![c_1]\!]^\sharp \circ [\![c_0]\!]^\sharp(s^\sharp) \\
[\![\mathbf{if}(e)\{c_0\}]\!]^\sharp(s^\sharp) := \mathfrak{join}_{\mathbb{S}}\left([\![c_0]\!]^\sharp(\mathfrak{guard}_{\mathbb{S},e\neq 0}(s^\sharp)), \mathfrak{guard}_{\mathbb{S},e=0}(s^\sharp)\right) & \\
[\![\mathbf{while}(e)\{c\}]\!]^\sharp(s^\sharp) := \mathfrak{guard}_{\mathbb{S},e=0}(\lim_n s^\sharp_n) & \\
\text{where } s^\sharp_0 := s^\sharp \text{ and } s^\sharp_{n+1} := \mathfrak{widen}_{\mathbb{S}}(s^\sharp_n, [\![c]\!]^\sharp(\mathfrak{guard}_{\mathbb{S},e\neq 0}(s^\sharp_n))) &
\end{array}
$$

Fig. 9. Abstract interpretation of a command

– when the memory components of the two arguments differ, they need to be *weakened* by replacing memory fragments with novel instances of $\mathbf{lseg}_s$, with fresh symbolic sequence variables, and by checking inclusion holds using $\mathfrak{is\_le}_{\mathbb{S}}$.

To illustrate the second case, we consider the over-approximation of the two abstract states defined by $s^\sharp_0 := (\alpha_0.\mathtt{n} \mapsto \alpha_1 * \alpha_0.\mathtt{d} \mapsto \alpha_3 * \mathbf{lseg}_s(\alpha_1, \alpha_2, S \boxdot), \sigma^\sharp_0)$ and $s^\sharp_1 := (\mathbf{lseg}_s(\alpha_0, \alpha_1, S \boxdot) * \alpha_1.\mathtt{n} \mapsto \alpha_2 * \alpha_1.\mathtt{d} \mapsto \alpha_3, \sigma^\sharp_1)$. Clearly, the memory part of both states may be weakened to the same abstract memory $\mathbf{lseg}_s(\alpha_0, \alpha_2, S'' \boxdot)$ where S'' is fresh. This gives the shape specific part of the result. However, in the case of $s^\sharp_0$, this weakening holds under the constraint $S'' = [\alpha_3].S$, whereas it holds under the constraint $S'' = S'.[\alpha_3]$ in the case of $s^\sharp_1$. Therefore, the sequence abstract states should be updated according to these two constraints before calling the corresponding operator in the sequence domain, which produces $\mathfrak{join}_{\Sigma}(\mathfrak{guard}_{\Sigma}(S'' = [\alpha_3].S, \sigma^\sharp_0), \mathfrak{guard}_{\Sigma}(S'' = S'.[\alpha_3], \sigma^\sharp_1))$. Note that this weakening also generates numerical and multi-set constraints. This constraint synthesis issue is carried out by an extension of the inclusion checking algorithm that keeps track of the fresh variables introduced by the widening and accumulates constraints over these.

Theorem 6 (Soundness of $\mathfrak{join}_{\mathbb{S}}, \mathfrak{widen}_{\mathbb{S}}$ and its termination). *The upper bound operator $\mathfrak{join}_{\mathbb{S}}, \mathfrak{widen}_{\mathbb{S}} : \mathbb{S}^\sharp \times \mathbb{S}^\sharp \to \mathbb{S}^\sharp$ are sound in the sense that, for all $s^\sharp_0, s^\sharp_1 \in \mathbb{S}^\sharp$, then $\gamma_{\mathbb{S}}(s^\sharp_0) \cup \gamma_{\mathbb{S}}(s^\sharp_1) \subseteq \gamma_{\mathbb{S}}(\mathfrak{join}_{\mathbb{S}}(s^\sharp_0, s^\sharp_1))$ and $\gamma_{\mathbb{S}}(s^\sharp_0) \cup \gamma_{\mathbb{S}}(s^\sharp_1) \subseteq \gamma_{\mathbb{S}}(\mathfrak{widen}_{\mathbb{S}}(s^\sharp_0, s^\sharp_1))$. Moreover, $\mathfrak{widen}_{\mathbb{S}}$ also ensures the termination property [17].*

4.5 Static analysis of a simple language

The analysis of a command c is a function $[\![c]\!]^\sharp : \mathbb{S}^\sharp \to \mathbb{S}^\sharp$ that over-approximates $[\![c]\!]$. It is defined by induction over the syntax in Figure 9. Note that the convergence of the sequence of abstract iterates follows from the termination property of $\mathfrak{widen}_{\mathbb{S}}$, and the analysis uses $\mathfrak{is\_le}_{\mathbb{S}}$ to detect stabilization. For conditional statements, we analyze the two branches separately after assuming the corresponding constraint, and we merge the two resulting states using $\mathfrak{join}_{\mathbb{S}}$. It is sound (the proof of soundness is classical [15] and proceeds by induction over the syntax):

Theorem 7 (Soundness). *For all command c, $[\![c]\!] \circ \gamma_{\mathbb{S}} \,\dot{\subseteq}\, \gamma_{\mathbb{S}} \circ [\![c]\!]^\sharp$.*

5 Shape and sequence predicates for non-linear structures

This section discusses the general inductive predicates used by our analysis. While Section 4 only considered basic list predicates so as to introduce the analysis in

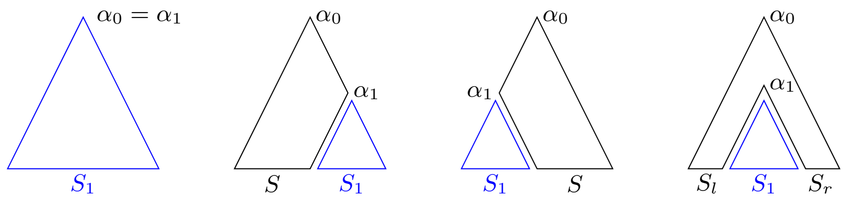

Fig. 10. Concatenation cases for tree segments and full tree predicates

a simpler setup, we now show our analysis handles m-ary trees (thus including lists when $m = 1$), possibly with parent pointers. We require that the sequence arguments of inductive definitions denote (sub-)sets of elements stored in structure (we comment on this restriction in Remark 1). The following paragraphs show the specificities of sequence predicates for such data-structures, the derivation of segment predicates and how it affects analysis operations.

Segment predicates and sequence information. Segment predicates such as **lseg** play a very important role in the analysis, e.g., to analyze data-structure traversals, as in Section 2. Basic analysis operations split or merge inductive predicates that describe full structures and segments. As we remarked in Section 4, sequence information needs to be maintained when such steps are performed and Lemma 1 provides the method to do so for **lseg**. As observed in Section 2, the method derived from Lemma 1 will not work for non linear structures.

Indeed, let us consider the tree segment predicate $\mathbf{treeseg}_s$ shown in Figure 4, which describes all the possible ways to decompose memory states that store a full tree at node α_0 and where α_1 is the address of one of its subtrees. Equivalently, the memory can be decomposed into a tree segment between α_0 and α_1 and a full tree at root α_1. We note S_0 (resp., S_1) the sequence of elements in the whole structure (resp., the subtree). Figure 10 depicts all possible configurations. In the first case, the subtree and the tree are equal, so the segment is empty and $S_0 = S_1$. In the second case, the subtree at α_1 is a leftmost subtree and $S_0 = S_1.S_r$ for some S_r. The third case is symmetric. The fourth case is the most general and $S_0 = S_l.S_1.S_r$ for some S_l, S_r. Therefore, the most general definition of the sequence(s) of elements in the segment (when the subtree in shown blue is excluded) is $S_0 = S_l \boxdot S_r$, where $\boxdot$ is a placeholder that abstracts the sequence of the elements in the "missing" subtree, and where S_l, S_r may denote the empty sequence.

Following this discussion, we now study concatenation of tree segment predicates. Let us assume two disjoint regions respectively abstracted by $\mathbf{treeseg}_s(\alpha, \alpha', S'_l \boxdot S'_r)$ and by $\mathbf{treeseg}_s(\alpha', \alpha'', S''_l \boxdot S''_r)$. Then, the union of these two regions may be abstracted by $\mathbf{treeseg}_s(\alpha, \alpha'', S_l \boxdot S_r)$ where $S_l = S'_l.S''_l$ and $S_r = S''_r.S'_r$. Note that the sequence expression attached to the latter segment is calculated as $S_l.S'_l \boxdot S'_r.S_r = (S_l \boxdot S_r)[\boxdot \leftarrow S'_l \boxdot S'_r]$. Similar reasoning may be carried out to concatenate a segment and a full tree predicate. Based on these observations, we propose a concatenation lemma for $\mathbf{treeseg}_s$:

Lemma 2 (Concatenation (tree case)). *We assume symbolic variables $\alpha, \alpha', \alpha''$ and sequence variables $S, S', S_l, S_r, S'_l, S'_r, S''_l, S''_r$.*

- *Let $m_0^\sharp := \mathbf{treeseg}_s(\alpha, \alpha', S'_l \boxdot S'_r) * \mathbf{tree}_s(\alpha', S')$, $m_1^\sharp := \mathbf{tree}_s(\alpha, S)$ $\sigma^\sharp := S = S'_l.S'.S'_r$. Then, $\gamma_{\mathbb{S}}(m_0^\sharp, \sigma^\sharp) \subseteq \gamma_{\mathbb{S}}(m_1^\sharp, \sigma^\sharp)$.*
- *Let $m_0^\sharp := \mathbf{treeseg}_s(\alpha, \alpha', S'_l \boxdot S'_r) * \mathbf{treeseg}_s(\alpha', \alpha'', S''_l \boxdot S''_r)$, $m_1^\sharp := \mathbf{treeseg}_s(\alpha, \alpha'', S_l \boxdot S_r)$ $\sigma^\sharp := S_l = S'_l.S''_l \wedge S_r = S''_r.S'_r$. Then, $\gamma_{\mathbb{S}}(m_0^\sharp, \sigma^\sharp) \subseteq \gamma_{\mathbb{S}}(m_1^\sharp, \sigma^\sharp)$.*

Derivation of segment predicates from full predicates. While inductive predicates (e.g., the definition of lists or trees) are user-supplied, our analysis automatically derives the corresponding segment predicates. Indeed, given a full predicate (like $\mathbf{tree}_s$) for an m-ary form of tree (including lists), the segment predicate is obtained by the sequence of steps below:

- each sequence argument S_i is replaced by a marked sequence $S_i \boxdot S'_i$,
- a rule describes empty segments; it abstracts an empty memory region, constrains its extremal points to be equal and its sequence contents to be empty;
- for each inductive rule that contains recursive calls to the inductive predicate, and for each such call c, the segment predicate should include a rule replacing c with a segment instance; moreover, in each such segment rule, the linearity of the sequence concatenations should be reflected by sequence constraints.

As an example, we illustrate this in the case of $\mathbf{tree}_s$:

Example 4 (Tree segments). The definition of $\mathbf{tree}_s$ is shown in Figure 1. As it has one sequence parameter, the corresponding segment predicate has two, that we note S_0 and S_1 and writes down $\mathbf{treeseg}_s(\alpha, \alpha', S_0 \boxdot S_1)$. We now detail the derivation of the $\mathbf{treeseg}_s$ predicate shown in Figure 4. As stated above, $\mathbf{treeseg}_s$ includes a rule for empty segments (the first one in Figure 4), which corresponds to an empty region, two equal pointers and two empty sequences. The first rule of $\mathbf{tree}_s$ corresponds to the empty tree; it has no recursive call and cannot appear in segments. The second rule of $\mathbf{tree}_s$ has two recursive calls (for the left and right subtrees), thus, it gives rise to two rules in $\mathbf{treeseg}_s$, that stand for cases where the segment is in the left (resp., right) subtree. Finally, we consider the constraints over sequences in the last rule (right subtree). Given the notation in Figure 4, the sequence of values in the whole tree is the argument $S \boxdot S'$ of $\mathbf{treeseg}_s$ which is equal to $S_l.\mathtt{[}v\mathtt{]}.S_r \boxdot S'_r$ in the last rule. This equality entails the constraints $S = S_l.\mathtt{[}v\mathtt{]}.S_r$ and $S' = S'_r$ which thus appear in the last rule of $\mathbf{treeseg}_s$.

Remark 1 (Limitation of sequence arguments). We observe that the inference of the sequence constraints by linearity as shown in Example 4 can only be achieved since the sequence constraints in $\mathbf{tree}_s$ specify that its segment argument collects a set of elements found at some fields in the structure. As an example, the analysis would not support an alternative definition of $\mathbf{tree}_s$ where the inductive rule would have the sequence constraint $S = \mathbf{sort}(S_l.\mathtt{[v]}.S_r)$, as it does not allow the derivation of precise constraints over sub-sequences for segments. We note that this limitation does not prevent capturing precisely binary search trees in the product abstract domain of Definition 5 with element $\mathbf{tree}_s(\alpha, S) \wedge S = \mathbf{sort}(S)$; instead, it only requires the shape predicate be written in a certain way.

Analysis. The analysis requires users to supply inductive predicates for full structures as well as target pre- and post-conditions. Segment predicates are inferred automatically as shown in the previous paragraph, as well as the appropriate concatenation lemma. Finally, the analysis operators are similar to those shown in Section 4, except that they use the concatenation property inferred from the definition of the full structure inductive predicate. For instance, when using the tree inductive predicate of Figure 1, the analysis infers the segment of Figure 4 and the concatenation lemma 2. The analysis satisfies the soundness property of Theorem 7. To conclude the section, we discuss a couple of steps of the computation of widening in the analysis of the program in Figure 2.

Example 5 (Inclusion checking). We consider the following abstract states:
– $s_0^\sharp = (\alpha.\mathtt{l} \mapsto \alpha_0 * \alpha.\mathtt{d} \mapsto \alpha_1 * \alpha.\mathtt{r} \mapsto \alpha_2 * \mathbf{tree}_s(\alpha_2, S_r), \sigma_0^\sharp)$;
– $s_1^\sharp = (\mathbf{treeseg}_s(\alpha, \alpha_0, S \boxdot S'), \sigma_1^\sharp)$.

Both $s_0^\sharp$ and $s_1^\sharp$ appear during the widening at the first iteration. We study the evaluation of the inclusion test $\mathfrak{is\_le}_{\mathbb{S}}(s_0^\sharp, s_1^\sharp)$. We first remark the following unfoldings (where $\alpha_3, \alpha_4, \alpha_5, S_l$, and S_l' are fresh) yield a similar abstract memory, up to existentially quantified symbolic variable names:

$$\begin{aligned} s_1^\sharp \rightsquigarrow\ & (\alpha.\mathtt{l} \mapsto \alpha_3 * \alpha.\mathtt{d} \mapsto \alpha_4 * \alpha.\mathtt{r} \mapsto \alpha_5 * \mathbf{treeseg}_s(\alpha_3, \alpha_0, S_l \boxdot S_l') \\ & * \mathbf{tree}_s(\alpha_5, S_r), \sigma_0^\sharp) \wedge S = S_l \wedge S' = S_l'.[\alpha_4].S_r \\ \rightsquigarrow\ & (\alpha.\mathtt{l} \mapsto \alpha_3 * \alpha.\mathtt{d} \mapsto \alpha_4 * \alpha.\mathtt{r} \mapsto \alpha_5 * \mathbf{emp} * \mathbf{tree}_s(\alpha_5, S_r), \\ & \sigma_0^\sharp) \wedge S = S_l \wedge S' = S_l'.[\alpha_4].S_r \wedge S_l = [] \wedge S_l' = [] \wedge \alpha_0 = \alpha_3 \end{aligned}$$

By the definition of $\mathfrak{is\_le}_{\mathbb{S}}$ in Section 4.4, $\mathfrak{is\_le}_{\mathbb{S}}(s_0^\sharp, s_1^\sharp)$ returns true if and only if $\mathfrak{is\_le}_{\Sigma}(\sigma_0^\sharp, \sigma_1^\sharp)$, $\mathfrak{verify}_{\Sigma}(\sigma_0^\sharp, S = [])$ and $\mathfrak{verify}_{\Sigma}(\sigma_0^\sharp, S' = [\alpha_1].S_r)$ all return true.

Example 6 (Widening). We now study the computation of the first widening in the analysis of the program shown in Section 2. For brevity, we only consider the second disjunct after the condition. The arguments of widening of abstract memory states and the result are shown in Figure 11. As mentioned in Section 4.4, the widening operator seeks for regions that can be described in a similar manner in the both of its arguments, possibly after weakening them. Matching colors in Figure 11 highlight pairings of similar regions. Recall that all symbolic variables $(\alpha, \alpha_0, \ldots)$ are existentially quantified within a same state. We observe the terms in blue, green and purple are pairwise equal and require no weakening. The areas in red though are not equal. For clarity, we add an **emp** term in $m_0^\sharp$. As observed in Example 5, the matching terms in $m_1^\sharp$ can be weakened into $\mathbf{treeseg}_s(\alpha_0, \alpha_1, S_1 \boxdot S_2)$, provided $S_1 = \mathtt{[]}$ and $S_2 = \mathtt{[}\delta\mathtt{]}.S_r$. The same holds for **emp** in $m_0^\sharp$. The table in the bottom of Figure 11 summarizes the correspondence between existentially quantified symbolic variables that realizes the association of regions.

The above paragraph describes the computation of the abstract memory state shown in Figure 3(c). The computation of the sequence abstract state of Figure 3(c) proceeds by application of $\mathfrak{widen}_{\Sigma}$.

$$\underbrace{\begin{pmatrix} \&\mathtt{t} \mapsto \alpha_0 \\ * \&\mathtt{c} \mapsto \alpha_0 \\ * \mathbf{emp} \\ * \mathbf{tree}_s(\alpha_0, S) \end{pmatrix}}_{m_0^\sharp} \quad \underbrace{\begin{pmatrix} \&\mathtt{t} \mapsto \alpha_0 * \&\mathtt{c} \mapsto \alpha_1 \\ * \alpha_0.\mathtt{l} \mapsto \alpha_1 * \mathbf{tree}_s(\alpha_1, S_l) \\ * \alpha_0.\mathtt{d} \mapsto \alpha_2 \\ * \alpha_0.\mathtt{r} \mapsto \alpha_3 * \mathbf{tree}_s(\alpha_3, S_r) \end{pmatrix}}_{m_1^\sharp} \quad \underbrace{\begin{pmatrix} \&\mathtt{t} \mapsto \alpha \\ * \&\mathtt{c} \mapsto \alpha' \\ * \mathbf{treeseg}_s(\alpha, \alpha', S_1 \boxdot S_2) \\ * \mathbf{tree}_s(\alpha', S_0) \end{pmatrix}}_{m_f^\sharp}$$

$m_f^\sharp$	α	α'	S_0	S_1	S_2
$m_0^\sharp$	α_0	α_0	S	`[]`	`[]`
$m_1^\sharp$	α_0	α_1	S_l	`[]`	$[\alpha_2].S_r$

Fig. 11. Shape union between states from Figures 3(a) and 3(b) (Greek letters denote existentially quantified symbolic variables; identical colors denote similar regions).

6 Implementation and evaluation

In this section, we report on the implementation and evaluation of the product shape and sequence analysis. We consider the following research questions:

- **(RQ1)** Is the combined analysis of Section 4 and Section 5 precise enough to prove functional properties on programs implementing classical algorithms over dynamic data-structures (like lists, sorted lists, and binary search trees), and does it help a baseline analysis verify structural invariants are preserved?
- **(RQ2)** Can this analysis successfully verify real-world C libraries?
- **(RQ3)** How significant is the overhead of the combined analysis compared to the baseline?

Implementation. We have implemented the sequence abstract domain and the product with the shape abstraction of the MemCAD static analyzer [38,1]. The analysis inputs C programs and user-supplied inductive predicates describing data-structures together with pre- and post-conditions and attempts to verify them, as well as absence of runtime errors. We set convex polyhedra [20] implemented in the Apron library [35] as numerical abstraction and an extension of [39] as multi-set abstraction.

Experiments. We consider two sets of experiments. The first one (Table 1) consists of custom implementations of classical algorithms over lists, sorted lists, and binary search trees and includes sorting, insertion and deletion algorithms. The second (Table 2) collects list data-structure implementations taken from the Linux [54] and FreeRTOS [33] operating systems as well as the Generic data-structure library (GDSL) [23], which all involve specificities like back pointers or sentinel nodes. For each data-structure, we provide an inductive definition written in the DSL of MemCAD. This amounts to a single definition a few lines long for each series of tests. For each test, we also specify the pre- and post-condition of procedures. When a procedure may behave differently depending on the shape of their input, we provide two pre-/post-condition pairs. This occurs for the "Pop" function, which does nothing when applied to the empty list. Two target properties are studied:

- **PrSafe**: absence of memory errors and structural preservation (with respect to list or tree invariants but without checking anything about their contents);

Example	without seq			with seq parameters					
	time	#iter	**PrSafe**	time				#iter	**PrSafe**
	all		verified	all	num	seq	shape		+ **Fc** verified
Singly linked list									
Push	4.0		✓	4.8	0.5	0.5	0.9		✓
Pop	5.1		✓	5.4	0.9	1.4	0.8		✓
Pop (empty)	4.9		✓	4.7	0.8	0.5	1.4		✓
concat	6.5	2	✓	15.7	3.4	3.3	2.7	2	✓
deep copy	12.1	2	✓	20.4	3.7	2.9	5.5	2	✓
length	9.5	3	✓	45.0	22.5	5.0	8.1	3	✓
insert at position	19.0	3	✓	101.9	61.3	7.9	12.2	3	✓
remove at position	17.2	3	✓	92.5	55.5	6.5	12.5	3	✓
inserting in a sorted list	13.5	3	✓	82.5	39.0	10.0	9.2	3	✓
minimum	11.8	3	✓	92.3	42.4	11.1	16.8	3	✓
maximum	11.8	3	✓	93.2	42.9	11.2	17.0	3	✓
insertion sort	24.6	2, 2	✓	714.6	328.6	90.0	126.3	4, 3	✓
bubble sort	40.6	2;2,3	✓(†)	776.3	399.5	89.2	141.5	3;3,3	✓(†)
merge	36.8	4	✓	352.2	180.9	41.0	54.9	4	✓
Binary trees									
Delete leftmost	11.2	3	✓	80.5	38.2	9.4	12.0	3	✓
Delete rightmost	11.5	2	✓	58.1	27.5	6.8	7.6	2	✓
Binary search trees									
Insertion	25.2	2	✓	150.4	58.0	17.2	15.5	2	✓
Delete max	22.9	2	✗	141.2	68.6	15.2	17.2	2	✓
Delete min	22.0	3	✗	177.9	87.9	19.2	22.8	3	✓
Search (present)	26.6	2	✓	107.2	48.6	15.7	14.4	2	✓
Search (absent)	24.0	3	✓	76.7	29.4	11.4	11.7	3	✓
BST to list (heap sort)	23.8	3	✓	76.5	29.2	11.4	11.7	3	✓
list to BST (heap sort)	34.2	2,2	✓	408.0	188.0	56.5	68.4	3,2	✓

Table 1. Experimental results on custom examples (Time in milliseconds averaged over 100 runs. For loop iterations, disjoint loops are separated by a semicolon, nested loops by a comma, and the first number corresponds to the outer loop. For inner loops, we take the maximum number of iterations needed to stabilize it.)

– **Fc**: partial functional correctness (including sortedness and the preservation of the elements stored in data-structures).

We ran the experiments on a machine with an i7-8700 processor with 32 Gb of RAM running Ubuntu 18.04. For each test case, we run the analysis *without* and then *with* sequence abstraction to compare runtimes and check if the analyses prove the expected property. When using the analysis without sequence abstraction, only **PrSafe** is considered (this abstraction cannot express **Fc**), whereas the analysis of sequences attempts to discharge both **PrSafe** *and* **Fc**. Table 1 displays raw results for the first series of tests. Table 2 shows the results of the main tests in the second series of tests.

Verification of complex properties. As shown in Table 1, the analysis with sequences fully verifies both memory safety and functional correctness (**PrSafe** and **Fc**) for all target codes including three different list sorting programs, operations on binary search trees as well as heap sort (elements of a list are all inserted in an empty binary search tree and collected in a left to right order back into a list). These examples all require the inference of fairly involved invariants. The analysis without sequences can only verify **PrSafe**, yet it fails to do so in several examples, where the use of sequences actually also lets the analysis verify **PrSafe** (in addition to **Fc**). This result is somewhat surprising, as we would not expect sequence information

Example	without seq			with seq parameters					
	time	#iter	**PrSafe**	time				#iter	**PrSafe**
	all		verified	all	num	seq	shape		+ **Fc** verified
Linux lists									
Init	1.1		✓	2.6	0.2	0.3	1.1		✓
Input	13.6		✓	21.4	2.7	2.4	8.2		✓
Output	22.7		✓	31.5	4.8	4.8	10.5		✓
Output (empty)	33.8		✓	9.3	1.4	1.0	2.5		✓
FreeRTOS lists									
vListInit	4.3		✓	6.1	1.3	0.4	0.6		✓
vListInsertEnd	23.8		✓	40.3	10.8	1.8	5.3		✓
vListInsert	87.4	4	✓	370.5	202.4	27.2	37.9	4	✓
vListRemove	47.5		✓	163.4	82.6	9.2	20.0		✓
GDSL (lists)									
Flush	24.3	2	✓	59.4	18.4	5.4	16.1	2	✓
Free	35.3	2	✓(†)	79.9	25.1	7.4	24.0	2	✓(†)
Remove head (empty)	34.1		✓	111.9	50.9	6.5	25.4		✓
Remove head (non-empty)	34.0		✓	16.3	5.7	1.1	3.7		✓
Remove tail (empty)	49.5		✓	284.8	165.0	13.6	39.3		✓
Remove tail (non-empty)	49.5		✓	16.2	5.7	1.1	3.6		✓
Search max	69.7	5	✓	708.4	429.7	43.1	145.7	5	**PrSafe** **Fc**
Search min	69.4	5	✓	634.0	380.3	35.4	131.2	5	**PrSafe** **Fc**
Search by position	104.5	3;2	✗(†)	1182.8	796.3	40.7	108.2	3;3	✓(†)

Table 2. Experimental results on libraries programs (Time in milliseconds averaged over 100 runs. For loop iterations, disjoint loops are separated by a semicolon, nested loops by a comma, and the first number corresponds to the outer loop. For inner loops, we take the maximum number of iterations needed to stabilize it.)

be required to establish basic safety. One caveat is that one example (bubble sort) required the manual insertion of a directive to MemCAD to delay folding. We conjecture the shape folding operator could be improved to avoid this. All other analyses are fully automatic. We conclude the product with sequences not only allows to prove **Fc** even in challenging cases, but may also help with **PrSafe**.

Verification of real-world libraries. We now consider Table 2. These examples involve lists with invariants that are considerably more sophisticated than $\mathbf{lseg}_s$, as they are all doubly-linked lists with headers. While GDSL lists contain a pointer to stored value blocks, both Linux and FreeRTOS lists are intrusive lists in the sense of the Linux kernel terminology: the `C` struct containing the `n` and `prev` fields is a substructure of the list node, which implies structure accesses require more complex pointer operations. FreeRTOS lists explicitly store a pointer from substructures to owners, whereas Linux lists rely on pointer arithmetic to access containing blocks. Finally, both FreeRTOS and GDSL lists have a header that stores the number of elements in the lists. FreeRTOS list nodes store a pointer to this header. The analysis with sequences proves both **PrSafe** and **Fc** for all Linux and FreeRTOS primitives. It was also able to fully verify almost all the GDSL list library, although two cases required a manual directive to prevent aggressive folding both with and without sequences (as for bubblesort in Table 1) (they are marked with (†) in the tables). Only two functions for the extraction of minimal/maximal values could not be fully verified with respect to **Fc** (note that **PrSafe** still gets proved): in these codes, the memory widening is too aggressive and folds the node storing the function results, which prevents proving that the returned value is indeed the extremal value in the sequence. All other examples not included in Table 2 are verified. We conclude the analysis handles real-world programs.

Overhead. We now compare performance between the analyses with/without sequences in Tables 1 and 2. While the overhead is modest for the smaller programs, it becomes higher for the more challenging cases, up to roughly 10x-20x. While significant, this cost should be considered in comparison to the much stronger properties proved (i.e., not only **PrSafe** but also partial correctness **Fc** in addition to **PrSafe**). We found two reasons for this increase. First, as shown in the tables, most of the increase is accounted for by the numerical abstract domain partly due to the larger number of symbolic variables that stand for sequence bounds. We believe this overhead could be much reduced with a finer-grained numerical domain packing [7,52]. By contrast, the time spent in memory and sequence domains remains reasonable. Second, the analysis with sequences requires greater numbers of abstract iterates to stabilize loop iterates, as shown in the tables, which explains an important slowdown. This is to be expected due to the more complex value constraints (including polyhedra) used in the analysis with sequences.

7 Related works

In this section, we discuss previous works on the abstractions of sequences stored in data-structures.

Linear and contiguous structures (arrays and strings). Several previous works have tried to tie properties of container data-structures with properties of their contents. In particular, [28,29] have extended array abstractions with basic contents properties. More recently [30] introduced array segmentations and [19] made the computation of the array segmentations dynamic during the analysis. The latter two can express that an array is sorted and verify that a function produces sorted arrays. However, they do so with specific predicates rather than an abstraction for sequences. Thus, they cannot express that the set of elements in an array is preserved, which is required to prove a sorting function correct. By contrast, our sequence abstraction handles both sortedness and contents preservation.

Strings and buffers also motivated many research works, as operations on them may incur a security risk. In particular, improper handling of zero terminated strings make opens the door to buffer overrun attacks. Therefore, works such as CSSV [25] abstract the presence or absence of zeroes in strings and their positions in order to verify buffer operations. Besides zeroes, these works do not keep any contents' information.

As noted earlier, several recent works applied concepts such as regular expressions and automata in order to build string abstract domains, that convey precise contents information [45,3,47]. These works are typically aimed at inferring precise information on strings that denote pieces of programs meant to be computed and evaluated at runtime as in the case of JavaScript's `eval` construction. Automata and regular expressions are most adequate for such target properties. More recently,

[4] extended these works with length and element position constraints. These abstractions are not aimed at numerical sequences, and fail to express sortedness. By contrast, our sequence abstraction relies on length, extremal elements and sortedness constraints and fails to express regular expressions-based properties as these would not be useful for our intended application. An interesting area of future work would be to build a reduced product of sequence abstractions to combine the expressiveness of these works and of ours.

Shape analyses for dynamic data-structures. Many abstractions for dynamic data-structures have been proposed. Sagiv *et al.* introduced a shape analysis based on three value logic in [51], that was later extended to handle more complex data-structures such as tree [42]. The seminal work by Reynolds [50], introduced separation logic, that many analyses including ours rely upon. Separation logic has been used in order to reason over not only sequential programs [12] but also concurrent programs [48,57] and to prove properties like linearizability of concurrent data structures [56]. It serves as a basis for structure abstraction in several static analyzers like Smallfoot [12], Facebook Infer [13] (which also performs bi-abduction to synthesize pre- and post-condition pairs), Forester [31] (which uses automata to represent abstract states), and MemCAD [38] (which features a modularized abstract domain). Bi-abduction methods have also been extended to infer inductive predicates on a per-function basis [37] or to infer pre- and post-conditions for programs manipulating lists and using bit-level memory accesses and pointer arithmetic [32]. All the shape abstractions mentioned so far can only keep track of very limited contents properties.

Indeed, inferring precise information about the contents of dynamic data-structures is notoriously difficult, since the memory abstraction layout changes depending on the program point which makes abstraction complex. A first approach to this issue consists in splitting the analysis in two phases, where the first analysis infers only structural invariants and translates the initial program into a purely numerical program, that is taken as input by the second analysis, that discovers numerical invariants. This technique has been applied by [43,27] in order to infer complexity bounds and verify termination of programs based on information on the size of the data-structures. A second approach [14] consists of a reduced product between a memory abstract domain and a numerical abstract domain. While harder to implement, it ensures information can be communicated in both directions between the memory and the value abstract domains, whereas the staged analysis approach only lets the value abstract domain benefit from memory layout information. More recently, [39] combines shape and set abstractions with a reduced product which allows verifying programs on graphs. As it only considers set constraints, it does not capture any order information.

The tools CINV [8] and CELIA [9] (extended with interprocedural analysis support in [11]) are the most closely related to our approach. These static analyzers handle list manipulating programs and are parameterized by an abstract domain called a *data-word domain* to reason on the structure and contents of lists by attaching size or set constraints to them, or constraints quantified over the position of elements, which allows expressing sortedness. Although the heap abstraction does not make explicit use of separation logic the list abstraction follows a similar structure. A first important difference with our work is that CINV and CELIA only handle singly-linked lists, whereas our analysis supports a large range of inductive definitions included doubly linked-lists, trees, binary search trees with and without parent pointers. Indeed, our approach integrates sequence reasoning into a shape analysis that can be parameterized by a wide variety of inductive predicates. This more general scope requires extensions to the analysis algorithms, such as the automatical inference of concatenation lemmas (Lemma 1 and 2) and the use of abstract operators based on them. A second difference comes from the sequence domain and the interaction with it. The data-word domain to handle sortedness relies on a decidable fragment of first order array theory based on constraints of the form $\forall \mathbf{y}, P(\mathbf{y}) \Rightarrow U(\mathbf{y}, Q_1, \dots)$, where the guard constraint $P(\mathbf{y})$ belongs to a predefined, user-provided set of *guard-patterns* constraining the index variables y_j, and U is a conjunction of linear constraints on y_j and $Q_i[y_j]$. This domain does not manipulate symbolic sequence expressions but rather follows a structural approach. For example, the concatenation constraint $S = S_1.S_2$ is expressed as $\forall y_1, y_2, y_1 < \mathtt{len}_{S_1} \land y_2 < \mathtt{len}_{S_2} \Rightarrow S_1[y_1] = S[y_1] \land S_2[y_2] = S[y_2 + \mathtt{len}_{S_2}]$. Therefore, it requires the user to specify prior to the analysis the appropriate guard pattern. Our sequence abstraction requires no such parameterization.

Provers for memory and contents properties. Separation logic has also been used as foundation for verification tools based on entailment checking procedures, some of which also consider contents properties. Songbird [53] uses a sequent-based approach to attempt deciding implication in a fragment of separation logic enriched with pure predicates. The procedure presented in [34] relies on tree automata to decide implications that involve inductive predicates. SLAD [10] decides entailment on a logic for singly-linked lists and the data stored in them. It handles order constraints on linear structures like lists and arrays. More recently, [22] used bi-abduction to reason about ordered data by explicitly storing bounds on elements in the inductive predicate. This work only considers full structure predicates and does not handle segment predicates. All these tools can be used to discharge implication proof obligations and can be used in verification tools where invariants are either manually written or inferred by some other means.

Additionally, separation logic is also heavily used in approaches based on proof assistants [16]. In that case, contents properties are naturally expressed in the proof assistant language.

Solvers for sequence properties. Finally, we remark that our language of sequence constraints based on concatenation of atoms has some similarity with the string logic that can be found in some decision procedures. Though the logic of word equations with at least two atoms is known to be undecidable [49], its quantifier free fragment has a PSPACE complete decision procedure [44]. Following the work of [2] that classifies the field of string constraints solving in three main branches, the automata based approach, using finite state automata to represent the set of constraints [40], the word based approach, that decomposes constraints using algebraic results such as Levi's lemma [6], and the unfolding based approach, which expresses each string variable as a bounded sequence of variables such as bit vectors [36], our abstraction can be categorized as mostly word-based. To the best of our knowledge, no SMT solver is able to reason on the sortedness of word expressions. We refer the reader to [2] for a comprehensive survey on string constraint solving. By comparison with these works, we provide an abstract domain interface on top of the sequence operation, which allows its use in a static analysis tool, following an instance of reduced product [18].

8 Conclusion and future works

In this paper, we presented a novel sequence abstract domain that relies on existing numerical and set abstractions, and extended a shape analysis with sequence reasoning. We demonstrated that the resulting analysis can be used in order to verify not only memory safety or structural preservation but also far more advanced correctness properties on a wide variety of inductive structures including various kinds of lists and trees. In particular, it could prove the functional correctness of several list sorting programs and of operations over binary search trees.

A combination of our analysis with a termination analysis [55,27] could verify not only partial correctness but also full correctness, which would be a first interesting direction for future works. Defining a reduced product over abstract domains for sequences would be also a useful research direction, as it would allow to strengthen the expressiveness of the analysis. Last, the evaluation also shows that performance of the combined analysis could be improved with the use of a more efficient dynamic packing [52] for relational constraints.

Acknowledgment The authors want to thank Thierry Martinez for his work on libraries used by the MemCAD analyzer. This work was supported by the VeriAMOS ANR-18-CE25-0010 French ANR project.

References

1. Artifact for "A Product of Shape and Sequence Abstractions". Zenodo (Jul 2023), https://doi.org/10.5281/zenodo.8186871
2. Amadini, R.: A survey on string constraint solving. ACM Computing Survey (2021)
3. Arceri, V., Mastroeni, I.: An automata-based abstract semantics for string manipulation languages. In: VPT@Programming (2019)
4. Arceri, V., Olliaro, M., Cortesi, A., Ferrara, P.: Relational string abstract domains. In: VMCAI (2022)
5. Assaf, M., Naumann, D.A., Signoles, J., Totel, E., Tronel, F.: Hypercollecting semantics and its application to static analysis of information flow. In: POPL (2017)
6. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: A string solver with theory-aware heuristics. In: FMCAD (2017)
7. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI (2003)
8. Bouajjani, A., Drăgoi, C., Enea, C., Rezine, A., Sighireanu, M.: Invariant synthesis for programs manipulating lists with unbounded data. In: CAV (2010)

9. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Abstract domains for automated reasoning about list-manipulating programs with infinite data. In: VMCAI (2012)
10. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Accurate invariant checking for programs manipulating lists and arrays with infinite data. In: ATVA (2012)
11. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: On inter-procedural analysis of programs with lists and data. In: PLDI (2011)
12. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Footprint analysis : A shape analysis that discovers preconditions. In: SAS (2007)
13. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL (2009)
14. Chang, B.Y.E., Rival, X.: Relational inductive shape analysis. In: POPL. ACM (2008)
15. Chang, B.E., Dragoi, C., Manevich, R., Rinetzky, N., Rival, X.: Shape analysis. FNT (1-2) (2020)
16. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: ICFP (2011)
17. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. ACM (1977)
18. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL (1979)
19. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL (2011)
20. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL (1978)
21. Cox, A., Chang, B.Y.E., Rival, X.: Automatic analysis of open objects in dynamic language programs. In: SAS (2014)
22. Curry, C., Le, Q.L.: Bi-abduction for shapes with ordered data (2020), arXiv, https://arxiv.org/abs/2006.10439
23. Darnis, N.: The generic data-structure library (2004), https://directory.fsf.org/wiki/GDSL
24. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.: Scaling static analyses at facebook. CACM (2019)
25. Dor, N., Rodeh, M., Sagiv, S.: Cssv: towards a realistic tool for statically detecting all buffer overflows in c. In: PLDI (2003)
26. Ferrara, P., Burato, E., Spoto, F.: Security analysis of the OWASP benchmark with julia. In: ITASEC (2017)
27. Fiedor, T., Holík, L., Rogalewicz, A., Sinn, M., Vojnar, T., Zuleger, F.: From shapes to amortized complexity. In: VMCAI (2018)
28. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In: POPL (2005)
29. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL (2008)
30. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI (2008)
31. Holík, L., Lengál, O., Rogalewicz, A., Simácek, J., Vojnar, T.: Fully automated shape analysis based on forest automata. In: CAV (2013)
32. Holík, L., Peringer, P., Rogalewicz, A., Šoková, V., Vojnar, T., Zuleger, F.: Low-level bi-abduction. In: ECOOP (2022)
33. Inc., A.: The freertos kernel (2022), https://github.com/FreeRTOS

34. Iosif, R., Rogalewicz, A., Vojnar, T.: Deciding entailments in inductive separation logic with tree automata. In: ATVA (2014)
35. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV (2009)
36. Kiezun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: A solver for word equations over strings, regular expressions, and context-free grammars. ACM Transaction of Software Engineering Methodology (2013)
37. Le, Q.L., Gherghina, C., Qin, S., Chin, W.N.: Shape analysis via second-order bi-abduction. In: CAV (2014)
38. Li, H., Berenger, F., Chang, B.Y.E., Rival, X.: Semantic-directed clumping of disjunctive abstract states. In: POPL (2017)
39. Li, H., Rival, X., Chang, B.E.: Shape analysis for unstructured sharing. In: SAS (2015)
40. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A dpll(t) theory solver for a theory of strings and regular expressions. In: CAV (2014)
41. Liu, J., Chen, L., Rival, X.: Automatic verification of embedded system code manipulating dynamic structures stored in contiguous regions. IEEE Transactions on Computer Aided Design and Integration of Circuits Systems (2018)
42. Loginov, A., Reps, T., Sagiv, M.: Automated verification of the deutsch-schorr-waite tree-traversal algorithm. In: SAS (2006)
43. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: POPL (2010)
44. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Mathematics of the USSR—Sbornik **32**(4) (1977)
45. Midtgaard, J., Nielson, F., Nielson, H.R.: A parametric abstract domain for lattice-valued regular expressions. In: SAS (2016)
46. Miné, A.: The octagon abstract domain. HOSC (2006)
47. Negrini, L., Arceri, V., Ferrara, P., Cortesi, A.: Twinning automata and regular expressions for string static analysis. In: VMCAI (2021)
48. O'Hearn, P.: Resources, concurrency and local reasoning. In: CONCUR (2004)
49. Quine, W.V.: Concatenation as a basis for arithmetic. Journal of Symbolic Logic **11**(4) (1946). https://doi.org/10.2307/2268308
50. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)
51. Sagiv, M., Reps, T., Whilhelm, R.: Solving shape-analysis problems in languages with destructive updating. TOPLAS (1998)
52. Singh, G., Püschel, M., Vechev, M.T.: Fast polyhedra abstract domain. In: POPL (2017)
53. Ta, Q.T., Le, T.C., Khoo, S.C., Chin, W.N.: Automated mutual explicit induction proof in separation logic. In: FAC (2016)
54. Torvalds, L.: The linux kernel (2022), https://git.kernel.org
55. Urban, C.: The abstract domain of segmented ranking functions. In: SAS (2013)
56. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: VMCAI (2009)
57. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: CONCUR (2007)