Semantic-Directed Clumping of Disjunctive Abstract States [*]

Huisong Li
INRIA Paris / CNRS / École Normale Supérieure / PSL* Research University
huisong@di.ens.fr

Francois Berenger
INRIA Paris / CNRS / École Normale Supérieure / PSL* Research University
francois.berenger@inria.fr

Bor-Yuh Evan Chang
University of Colorado Boulder
evan.chang@colorado.edu

Xavier Rival
INRIA Paris / CNRS / École Normale Supérieure / PSL* Research University
rival@di.ens.fr

Abstract

To infer complex structural invariants, shape analyses rely on expressive families of logical properties. Many such analyses manipulate abstract memory states that consist of *separating conjunctions* of basic predicates describing atomic blocks or summaries. Moreover, they use *finite disjunctions* of abstract memory states in order to account for dissimilar shapes. Disjunctions should be kept small for scalability, though precision often requires keeping additional case splits. In this context, deciding when and how to merge case splits and to replace them with summaries is critical both for precision and efficiency. Existing techniques use sets of syntactic rules, which are tedious to design and prone to failure. In this paper, we design a *semantic criterion* to clump abstract states based on their silhouette, which applies not only to the conservative union of disjuncts but also to the weakening of separating conjunctions of memory predicates into inductive summaries. Our approach allows us to define union and widening operators that aim at preserving the case splits that are required for the analysis to succeed. We implement this approach in the MemCAD analyzer and evaluate it on real-world C codes from existing libraries dealing with doubly-linked lists, red-black trees, AVL-trees and splay-trees.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages-Program analysis

Keywords Static analysis, abstract interpretation, heap abstraction, separation logics, disjunctions, silhouette, clumping of disjuncts

1. Introduction

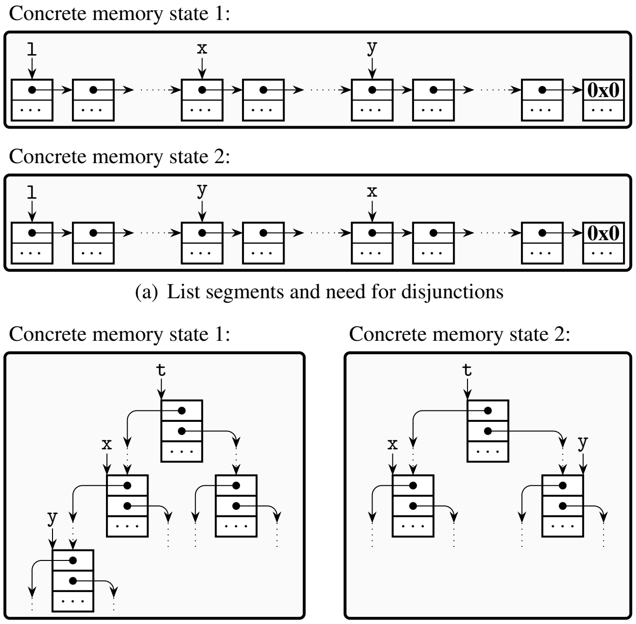
Over the last two decades, a wide spectrum of analyses have been developed so as to compute invariants for programs manipulating complex, unbounded data structures, and in order to prove memory safety, structural invariance or functional properties. Shape analyses [17, 35] focus on dynamic structures such as linked lists, trees and graphs. Array analyses like [13] deal with contiguous structures indexed by ranges of integers. Dictionary analyses [14, 16] handle structures indexed using sets of keys. All these analyses need to use expressive sets of predicates in order to summarize structures of unbounded size, while keeping information about their shape.

Many of these works [9, 13, 14, 17] implicitly or explicitly use *separation* [32] to describe sets of concrete memory states, which means they divide memory states into regions and describe each region individually using specific predicates. Typically region predicates either describe a finite region very precisely (namely, cell-by-cell), or summarize an unbounded region using an inductive predicate [4, 9, 17, 38], a property that holds over a possibly empty range of cells [13] or for a set of keys [14]. An abstract state combines region predicates. For instance, [17] partitions memories into list segments. Similarly, [13] partitions arrays into segments. Thus, an abstract state can be viewed as a logical formula described by a grammar of the form:

$$\begin{array}{llcll} \text{abstract states:} & \mathrm{m}(\in \mathbb{M}) & ::= & \mathrm{p} * \ldots * \mathrm{p} & \\ \text{region predicates:} & \mathrm{p}(\in \mathbb{P}) & ::= & \mathrm{p_e} & \text{(exact description)} \\ & & \mid & \mathrm{p_s} & \text{(summary)} \end{array}$$

However, such a set of logical predicates is often not sufficient to describe precisely a set of concrete memory states. Indeed, when the program being analyzed may produce states with too different structures, it becomes impossible to abstract them all into a single separating conjunction of base predicates. For instance, Fig. 1(a) shows a pair of concrete memory states that the above predicates cannot describe precisely, yet that may arise in real programs. Both memory states contain a list pointed to by variable `l`, and two "cursors" $\mathtt{x}, \mathtt{y}$ pointing somewhere in that list. The only difference between these two memories is the order of the cursors $\mathtt{x}, \mathtt{y}$. A code fragment localizing independently two elements in the list will produce such states. Analyses like [9, 17] instantiate a generic list segment predicate $\mathbf{ls}(a, b)$ to abstract a segment of list starting at a and finishing with a pointer to b. Using this abstraction, concrete memory state 1 can be abstracted by $\mathbf{ls}(\mathtt{l}, \mathtt{x}) * \mathbf{ls}(\mathtt{x}, \mathtt{y}) * \mathbf{ls}(\mathtt{y}, \mathbf{0x0})$, yet that predicate cannot abstract the second one in the same time as $\mathtt{x}, \mathtt{y}$ appear in the reverse order. Instead, static analyses

[*] The research leading to these results has received funding from the European Research Council under the European Union's seventh framework programme (FP7/2007-2013), grant agreement 278673, Project MemCAD, from the ARTEMIS Joint Undertaking no 269335 (see Article II.9 of the JU Grant Agreement), and the U.S. National Science Foundation under grant number CCF-1055066.



(a) List segments and need for disjunctions

(b) Tree fragments and need for disjunctions

Figure 1. Structures and disjunctions.

need to handle *disjunctions* of abstract states:

$$\begin{array}{lrcl} \text{disjunctive abstract states:} & \mathrm{d}(\in \mathbb{D}) & ::= & \mathrm{m} \vee \ldots \vee \mathrm{m} \\ \text{abstract states:} & \mathrm{m}(\in \mathbb{M}) & ::= & \mathrm{p} * \ldots * \mathrm{p} \end{array}$$

Then, $\mathbf{ls}(\mathtt{l}, \mathtt{x}) * \mathbf{ls}(\mathtt{x}, \mathtt{y}) * \mathbf{ls}(\mathtt{y}, \mathbf{0x0}) \vee \mathbf{ls}(\mathtt{l}, \mathtt{y}) * \mathbf{ls}(\mathtt{y}, \mathtt{x}) * \mathbf{ls}(\mathtt{x}, \mathbf{0x0})$ abstracts the memories of Fig. 1(a). Without a disjunction, the information about either of the cursors would have to be dropped.

The same situation arises for other kinds of structures. For instance, Fig. 1(b) displays two concrete memory states containing a tree, with root `t` and two inside pointers. These shapes cannot be described only using separation logic and tree inductive definitions: indeed, in the first case, `x` points to a subtree of `t` and `y` points to a subtree of `x` whereas in the second case the subtrees pointed to by `x` and `y` appear in separate children of `t`.

In practice, disjunctions are a huge challenge to static analysis tools. While the creation of new disjunctions occurs naturally when the analysis needs to reason over operations that read or write into summaries, letting the number of disjuncts grow makes the analysis slower and consume more memory. Yet, getting rid of unnecessary disjuncts turns out to be much harder a task than introducing them. To *clump* a disjunctive abstract state d, an analysis needs:

1. to *sort* the disjuncts of d into sets of abstract states $\mathrm{M}_0, \ldots, \mathrm{M}_n$, such that all abstract states in M_i are similar enough;
2. to *compute* for each set M_i an abstract state m_i that conservatively over-approximates all the elements of M_i; this weakening should infer how sets of region predicates can be folded into summary region predicates.

Both steps are critical. The first step should determine the right set of disjuncts: leaving too many disjuncts would make it impossible to scale whereas excessively reducing the size of the disjunction will prevent the weakening step from producing precise summaries.

Existing approaches all come with limitations and are often challenged by the first step (disjunct sorting). Canonicalization operators [35] solve this problem by using a finite set of "canonical" abstract states, replacing each abstract state with a canonicalized version of it. Although the analysis may use an infinite domain, the precision of canonicalization outputs is limited by that of the finite set of canonical abstract states. The canonicalization operator of [17] and the join operators of [9, 38] utilize local rewriting rules based on the syntax of abstract states. They cannot reason on global shape properties, thus may miss chances to clump some disjuncts. State partitioning [12] and trace partitioning [22, 33] provide frameworks for sensitivity in static analysis, but do not provide a general strategy to choose which disjunctions to preserve. We observe that static strategies based on the control flow structure of programs (conditions, loops, etc) are likely to produce inadequate disjunct clumps as they ignore the shapes. On the other hand, disjunctive completion [11] authorizes any disjunction of abstract states (so that simplification is never required); however, it cannot deal with infinite sets of abstract predicates and is prohibitively costly when the set of abstract states is finite. Pruning disjunction is thus a major challenge in many memory reasoning tools.

In this paper, we observe that semantic properties of abstract states can help characterize the clumping of disjuncts. For instance, the reason why the memory states of Fig. 1(a) cannot be abstracted together lies in the order of the pointers. This information is described by a very concise "*silhouette*" abstraction of the abstract states themselves that focuses on the backbone of inductive structures. Indeed, if we let $\rightsquigarrow$ be the relation that states that there is a link path from one pointer to another, then $\mathtt{l} \rightsquigarrow \mathtt{x} \rightsquigarrow \mathtt{y}$ in the first state and $\mathtt{l} \rightsquigarrow \mathtt{y} \rightsquigarrow \mathtt{x}$ in the second one; moreover, these path-based silhouettes abstract the **ls** predicates (as well as other points-to predicates). In essence, this technique uses a form of lightweight canonicalization, but only as a guide to decide which disjuncts to clump, whereas the analysis computations (including the abstract join for clumping) still all take place in the initial, infinite lattice. Similarly, the states of Figure 1(b) can be discriminated by a similar abstraction of paths.

Therefore, we propose to let silhouette abstractions of the abstract states guide the algorithms for *clumping* and *weakening* disjuncts. We make the following contributions:

- we set up a path-based silhouette abstraction of abstract states to capture shape similarities and guide weakening in Sect. 3;
- we design algorithms for the clumping (Sect. 4) and the widening (Sect. 5) of disjunctive abstract states; and
- we implement the silhouette-guided algorithms in the MemCAD analyzer [36] and assess their efficiency with the verification of several real-world C libraries manipulating structures such as doubly-linked lists, red-black trees, AVL trees, and splay trees.

2. Overview

In this section, we formalize the disjunct clumping problem and describe the core contribution of the paper at a high level.

Clumping predicates based on their abstraction. As in the introduction, we assume that an abstract state $\mathrm{m} \in \mathbb{M}$ is a separating conjunction $\mathrm{p}_0 * \ldots * \mathrm{p}_n$ of basic region predicates, and we consider an analysis that computes an over-approximation for reachable concrete memories represented as disjunctive abstract states of the form $\mathrm{d} = (\mathrm{m}_0 \vee \ldots \vee \mathrm{m}_k)$. We let $\mathbf{join}_{\mathbb{M}}$ denote a computable join operation over the set of abstract states that over-approximates unions of sets of memory states. Thus, it is always sound for the analysis to replace a disjunctive abstract state $(\mathrm{m}_0 \vee \mathrm{m}_1)$ with a non-disjunctive abstract state $\mathbf{join}_{\mathbb{M}}(\mathrm{m}_0, \mathrm{m}_1)$, yet this operation may in general lose some information. Disjunct clumping aims at identifying a partition of a finite set of disjuncts so that each component of the partition can be joined using $\mathbf{join}_{\mathbb{M}}$ without a significant loss of precision. For instance, as observed in the introduction, joining into a single abstract state abstraction the concrete states of Fig. 1(a) would not allow to represent all segments separately, therefore such disjuncts should not be clumped together.

```
typedef struct node_t {
  struct node_t *l, *r; // left, right child
  int bal, d; // balancing and content
} node;

int insert_non_empty( node * t, int i ){
  assume( t != null );
  node *h = (node*) malloc(sizeof(node));
  node *x, *y, *p, *q;
  h->l = y = p = t;
  x = h;
  // phase 1: insertion point localization
  while( 1 ){
    if( i < p->d )
      q = p->l;
    else
      q = p->r;
    if( q == null )
      break;
    if( q->bal != 0 ){
      x = p;
      y = q;
    }
    p = q;
  }
  // phase 2: insertion at position p...
  // phase 3: rebalancing at position x, y...
}
```

Figure 2. Excerpt of C source code for the AVL tree insert function.

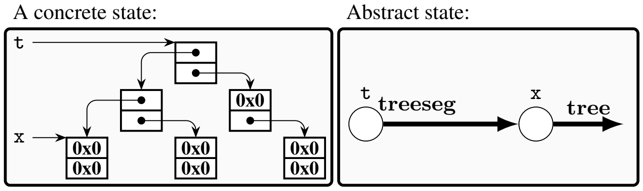


Figure 3. Abstract states.

In this paper, we propose an approach to disjunct clumping that is based on the use of an *abstraction of abstract states*:

- we shall design a set of logical predicates $\mathbb{S}$ that describe the *silhouettes* of abstract states in a compact and simple representation, and a computable silhouette equivalence relation $\sim$ that defines which silhouettes are similar;
- we shall define a computable abstraction function $\theta : \mathbb{M} \longrightarrow \mathbb{S}$ that maps an abstract state into its silhouette.

As an example, we defined $\rightsquigarrow$ in the introduction, as a compact abstraction of pointer chains, and used it to build silhouettes.

Using this notion of silhouette, we define the clumping algorithm $\mathbf{clump}$ that inputs a disjunctive abstract state d and returns another disjunctive abstract state $\mathbf{clump}(\mathrm{d})$ that over-approximates d with at most as many (though usually fewer) disjuncts:

1. it inputs a disjunctive abstract state $\mathrm{m}_0 \vee \ldots \vee \mathrm{m}_n$;
2. it computes $\theta(\mathrm{m}_0), \ldots, \theta(\mathrm{m}_n)$, and packs together abstract states with the same image by θ up to equivalence relation $\sim$;
3. it reduces each group into a single abstract state, by repeatedly applying $\mathbf{join}_{\mathbb{M}}$, and returns a more compact disjunction.

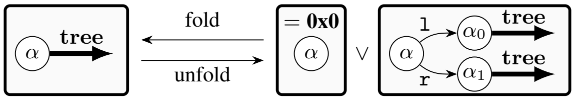
In the rest of this section, we define an instance of clumping of disjunctive predicates, by defining $\mathbb{S}$, θ and $\sim$ and we demonstrate how this approach enables the analysis of a complex code fragment, while limiting the size of disjunctions.

Example: insertion into an AVL tree. In the following, we study the verification of a function that inserts an element into an AVL tree. AVL trees achieve balancing by enforcing that the heights of two subtrees of a single node differ by at most one, and by storing that difference on each node, so that insertion and removal algorithms can rebalance subtrees incrementally, using "rotation" operations. This property makes insertion and removal involved, thus the preservation of structural invariants and absence of memory errors (such as illegal pointer operations or leakage of subtrees) are difficult to verify. In particular, the insertion and removal functions need to distinguish many cases, which their verification should also discriminate. Thus, verification algorithms will produce large numbers of disjuncts, and could greatly benefit from clumping. Fig. 2 shows an excerpt of an AVL insertion function. While the verification of the full code (from [37]) is presented in Sect. 6, we study here a simplified version for the sake of clarity. We only consider the handling of disjunctions of memory shapes; in particular, we ignore the AVL trees numeric balancing constraints (for our purpose, this restriction has no impact on the analysis).

The fragment in Fig. 2 handles the insertion into a non-empty tree, and carries it out in three phases. First, the node p at which the new element should be inserted is localized, as well as the deepest edge (with source x and target y) in the tree where the balancing property is locally broken by this insertion (indeed, this is the only point where a rebalancing will actually be required, since a rotation at this point will prevent any other balancing constraint from being broken). Second, a new node is allocated and inserted at position p. Finally, the rebalancing itself is performed. For concision, we consider in detail the first phase only. This phase should compute pointers p, x, y. To verify this function, the analysis should compute invariants that characterize precisely the shape of trees, subtrees and the relative positions of pointers p, x, y, t in all cases, including the corner case where tree t consists of a single node, and the insertion is actually done above it.

Abstract states and analysis. To express invariants over the code of Fig. 2, the analysis needs summary predicates describing trees of unbounded size as well as cursors inside trees. Thus, we assume a pair of summary predicates $\mathbf{tree}$ and $\mathbf{treeseg}$. More precisely, if α denotes a symbolic address, we let $\mathbf{tree}(\alpha)$ describe a well-formed complete tree stored at address α. Similarly, we let $\mathbf{treeseg}(\alpha, \beta)$ describe a tree stored at address α missing a subtree at address β. Therefore abstract values stand for separating conjunctions of points-to predicates and of summary predicates. We use shape graphs, as shown in Fig. 3 as a more intuitive representation, where nodes stand for symbolic addresses (and the values of variables), and bold edges stand for full summary predicates (edges with no destination node) and for segment predicates (edges with a destination node). The abstract state on the right side of the figure describes states where t points to a tree, and x points to a (possibly non-strict) subtree of the tree pointed to by t. It is made of the separating conjunction of a $\mathbf{treeseg}$ segment summary and a $\mathbf{tree}$ summary. An example concrete memory is shown in the left side of the figure.

Both summary predicates have a natural inductive structure. A region abstracted by $\mathbf{tree}$ is either an empty tree or a tree with a node and two disjoint subtrees. This remark boils down to a pair of fold / unfold rules, that fully characterize $\mathbf{tree}$ (likewise $\mathbf{treeseg}$ is defined by such a pair of rules):



The analysis performs a forward abstract interpretation [10], starting from an abstract precondition that accounts for all the possible valid call states, that is all the memory states where t points to a non-empty, well-formed tree (described by $\mathbf{tree}$). It computes abstract post-conditions for each statement, and unfolds summaries on-demand: for instance, in the first iteration $\mathtt{p} = \mathtt{t}$,

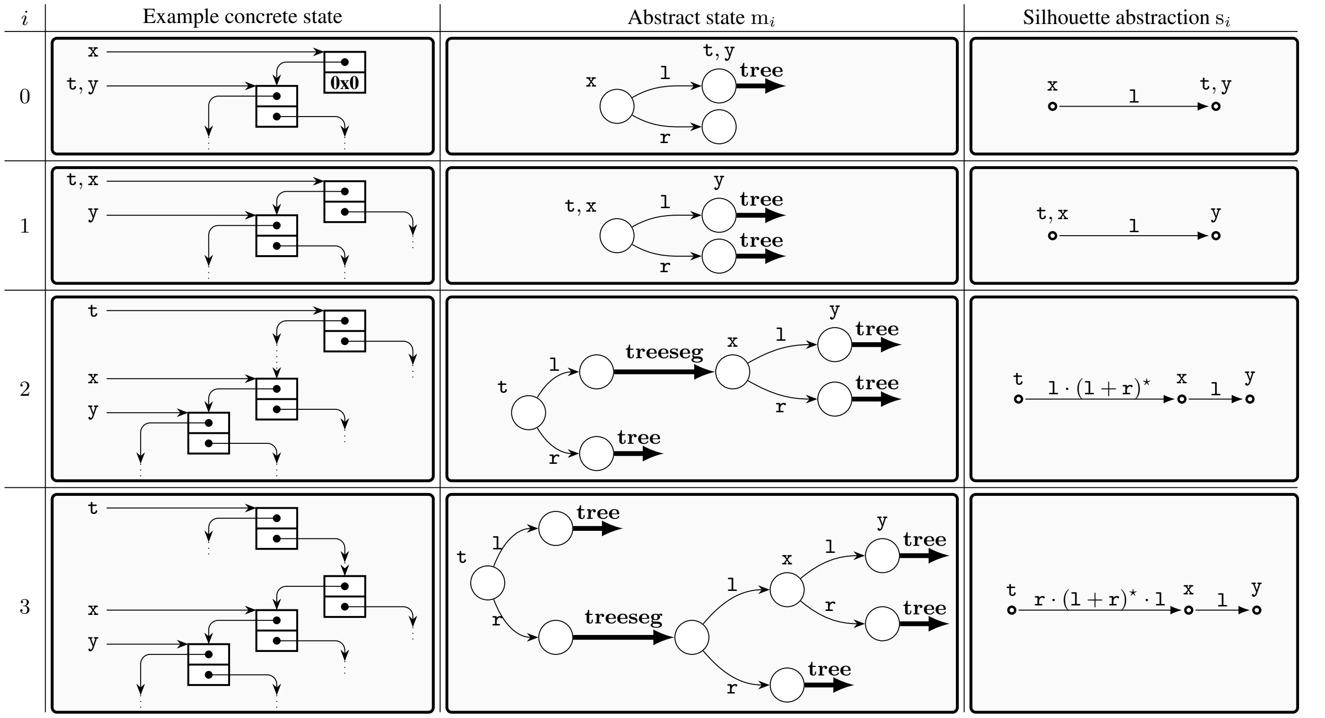


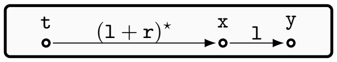
Figure 4. Selected abstract states from the analysis of an insertion into an AVL tree (4 disjuncts out of 32).

and at line 14, the reading of field `d` of the node pointed to by `p` requires unfolding the **`tree`** summary predicate attached to `t`. Such unfoldings generate disjunctions of case splits. Conversely, join and widening applied at loop head should fold back case splits, so as to compute a loop invariant.

Silhouette abstraction. Fig. 4 describes a few abstract states that are observed at line 25, at the exit of the first loop, and that illustrate the challenges of clumping disjuncts. The figure shows abstract states, the generic form of concrete memories they represent and their silhouettes to be explained below. For concision, and as only the relative positions of cursors matter, we focus on `t`, `x`, `y` and omit fields `d`, `bal`, and variables `h`, `p`, `q`. Labels `t`, `x`, `y` decorate the nodes that represent their value. Moreover, we show only a sample of 4 among 32 disjuncts. First, abstract state m_0 abstracts memories where `x`, `y` were not advanced. Second, abstract state m_1 abstracts memories where `x` was advanced to the root of the tree, and the search continued in the left subtree. Third, abstract state m_2 abstracts memories where the search visits the left subtree of `t`, `x` is advanced into that subtree, and `y` is the left child of `x`. Last, abstract state m_3 describes a similar condition as m_2 but when the search visits the right subtree of `t` and `x` is a left child. Some of these abstract states are very similar to each other and can be joined to reduce the cost without significantly affecting the precision of the analysis. Indeed, `x` and `y` occupy the same relative positions in m_2 and m_3; moreover, in both cases the two cursors point to subtrees of `t`. Thus, both m_2 and m_3 can be approximated by an abstract state with a segment from `t` to `x` and where `y` is the left child of `x`. Furthermore, m_1 can also be weakened similarly: the relative positions of `x`, `y` are the same, and the only minor difference is that `x` is not a strict subtree of `t` since $\mathtt{t} = \mathtt{x}$, but this equality can also be described by an (empty) segment. On the other hand, m_0 abstracts very different memories, where `x` is *not* a subtree of `t`, so it cannot be described with a segment from `t` to `x`. Any abstract state that over-approximates both m_0 and m_1 would discard all information about either `t` or `x`, which would make the proof of structural preservation impossible. Therefore, an ideal clumping would join $\mathrm{m}_1, \mathrm{m}_2, \mathrm{m}_3$ together but keep m_0 separate.

Intuitively, abstract states where the relative positions of cursors `t`, `x`, `y` into the tree are similar can be clumped together with no severe precision loss. To capture this intuition, the notion of *silhouette* shown in the last column of Fig. 4 retains only the relative positions of `t`, `x`, `y` and the access paths between them. As access paths may be of unbounded length, we abstract them with regular expressions describing sequences of fields dereferenced between nodes. For instance, the silhouette s_0 of m_0 boils down to a single edge, with a single path labeled by `l`. Even the more complex m_3 is characterized by only two edges respectively labeled by $\mathtt{r}\cdot(\mathtt{l}+\mathtt{r})^\star\cdot\mathtt{l}$ (between `t` and `x`) and by `l` (between `x` and `y`).

Clumping disjuncts. Silhouettes make the dissimilarity of m_0 with the other states in Fig. 4 obvious, due to the incompatible order of cursors. However, we can also see that the silhouettes of m_2 and m_3, while not syntactically identical are actually equal up-to a generalization of the path regular expression for the left edge into $(\mathtt{l}+\mathtt{r})^\star$. This generalization matches the fact that any structure segment can be weakened into a **`treeseg`** predicate, whatever the sequence of "left-right" branches it encompasses:

$\mathrm{s}_\sim =$ 

The silhouette of m_1 also corresponds to a special case of $\mathrm{s}_\sim$. Therefore, we let $\sim$ denote the similarity of silhouettes up-to generalization of access paths. With this notation, we have:

$$\mathrm{s}_0 \not\sim \mathrm{s}_1 \qquad \mathrm{s}_0 \not\sim \mathrm{s}_2 \qquad \mathrm{s}_0 \not\sim \mathrm{s}_3 \qquad \mathrm{s}_1 \sim \mathrm{s}_2 \sim \mathrm{s}_3$$

Therefore, the groups computed here are $\{\mathrm{m}_0\}, \{\mathrm{m}_1, \mathrm{m}_2, \mathrm{m}_3\}$, and the four disjuncts of Fig. 4 are clumped into a disjunctive abstract state composed of only two disjuncts m_0 and $\mathrm{m}_{1,2,3}$. The

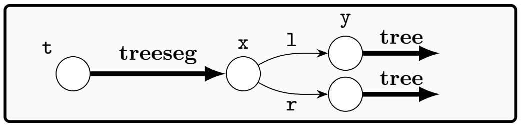


Figure 5. Weakened abstract state $\mathrm{m}_{1,2,3}$.

computation of the $\mathrm{m}_{1,2,3}$ (that we discuss in the next paragraph) may use any weakening algorithm for abstract states, such as canonical abstraction [35], canonicalization of symbolic heaps [17], or shape graph join [9].

In essence, clumping relies on a weak canonicalization [35] that returns silhouettes, and then selects groups based on an equivalence relation over the set of silhouettes. To ensure the termination of abstract iterates over loops, the finiteness of the clumping relation $\sim$ is required (i.e., it should have a finite set of equivalence classes), though the set of silhouettes may still be infinite. However, clumping may be performed at other points than loop heads (in order to shrink abstract states, without discarding any information), and then it does not require a finite $\sim$. In this point of view, clumping provides a more flexible approach to the handling of disjunctions than canonicalization, since it does not need to project abstract states into a finite set of predicates, and since it may still take advantage of precise binary operators for weakening.

Clumping region predicates. The information computed in silhouettes also provides a guideline for weakening abstract states. Indeed, let us consider silhouette $\mathrm{s}_\sim$, that is weaker than the silhouettes of $\mathrm{m}_1, \mathrm{m}_2, \mathrm{m}_3$. As it contains an edge labeled by $(\mathtt{l} + \mathtt{r})^\star$ between $\mathtt{t}$ and $\mathtt{x}$, it suggests weakening the fragment of these abstract states that are between $\mathtt{t}$ and $\mathtt{x}$ into a $\mathbf{treeseg}$ segment predicate. The soundness of such a weakening can be verified by checking entailment of a fragment of m_1 (resp., $\mathrm{m}_2, \mathrm{m}_3$) and a $\mathbf{treeseg}$ predicate. The resulting weaker abstract state $\mathrm{m}_{1,2,3}$ is shown in Fig. 5. Here, the role of the silhouette is to provide guidance on how abstract states may be weakened. The advantage of this view to the computation of weakening is to benefit from global semantic information about the structure of abstract states that would be neglected by weakening operators based on syntactic rules only (like the canonical heap abstraction of [17] and the join of [9]).

3. Silhouette Abstraction and Guided Weakening

In this section, we formalize silhouettes, and show that they provide a useful abstraction to reason about the weakening of abstract states.

3.1 Preliminaries: Memories and Abstract States

Before we formalize silhouettes, we define memories and abstract states. We let $\mathbb{X}$ denote a set of pointer variables and $\mathbb{V}$ denote a set of values (that includes numeric addresses). We let fields (noted as $\mathtt{l}, \mathtt{r}, \ldots$) denote both field names and offsets. A memory state $\sigma \in \Sigma$ is a partial function from addresses and variables to values. We let $\sigma_{\restriction \mathbb{X}}$ (resp., $\sigma_{\restriction \mathbb{H}}$) denote the restriction of σ to variables (resp., heap addresses). If $\mathtt{x}$ (resp., a) is a variable (resp., an address), we write $\sigma(\mathtt{x})$ (resp., $\sigma(a)$) for the content of variable $\mathtt{x}$ (resp., the cell of address a). This basic memory model does not allow computing the address of a variable, but a more general definition based on a classical heap and environment pair could be used without changing the principle of our approach.

An *abstract state* describes a set of concrete memory states. It is either $\perp$ (denoting $\emptyset$) or a separating conjunction of region predicates that abstract *separate* memory regions [32] in conjunction with numerical constraints such as equalities and disequalities. The syntax of abstract states is given in Fig. 6. Abstract states utilize a set $\mathbb{A} = \{\alpha, \beta, \ldots\}$ of *symbolic addresses* to abstract values and heap addresses. A node n is either a variable $\mathtt{x}$ or a symbolic address α. A region predicate is either $\mathbf{emp}$ describing an empty region, a points-to predicate $\mathrm{n} \cdot \mathtt{f} \mapsto \mathrm{n}'$ (that describes a heap memory cell at base address n with offset $\mathtt{f}$ and with content n'), or a summary predicate $\mathbf{ind}(\mathrm{n})$ describing a full inductive structure or $\mathbf{indseg}(\mathrm{n}, \mathrm{n}')$ for a (possibly empty) inductive segment from address n to n'. This set of abstract states is parameterized by a user-supplied inductive predicate $\mathbf{ind}$, that characterizes a family of data structures of interest and serves as a template that the analysis will instantiate. It is specified by a disjunction of cases, which boil down to separating conjunctions of points-to and inductive summary predicates (e.g., lists, doubly-linked lists, trees, and trees with parent pointers can be expressed this way). Segment predicate $\mathbf{indseg}$ stands for the segment version of $\mathbf{ind}$ and describes $\mathbf{ind}$-structures with one "hole" (i.e., one missing substructure), and can also be defined by induction. The $\mathbf{ls}$ list segment predicate of [17] is an inductive predicate. The $\mathbf{tree}$ predicate used in Sect. 2 can be defined by induction as well:

$$
\begin{aligned}
\mathbf{tree}(\mathrm{n}) ::= {} & \mathbf{emp} \wedge \mathrm{n} = \mathbf{0x0} \\
\vee {} & \mathrm{n} \cdot \mathtt{l} \mapsto \alpha_l * \mathrm{n} \cdot \mathtt{r} \mapsto \alpha_r * \mathrm{n} \cdot \mathtt{bal} \mapsto \alpha_b \\
& * \mathrm{n} \cdot \mathtt{d} \mapsto \alpha_d * \mathbf{tree}(\alpha_l) * \mathbf{tree}(\alpha_r) \wedge \mathrm{n} \neq \mathbf{0x0}
\end{aligned}
$$

The inductive definition of $\mathbf{treeseg}(\mathrm{n}, \mathrm{n}')$ has three cases: either the segment is empty and $\mathrm{n} = \mathrm{n}'$, or it is non-empty and n' is in the left subtree of n or it is non-empty and n' is in the right subtree of n.

Abstract states can also be viewed as shape graphs. For instance, Fig. 3 depicts as a shape graph the abstract state $\mathbf{treeseg}(\mathtt{t}, \mathtt{x}) * \mathbf{tree}(\mathtt{x})$.

The logical meaning of an abstract state m is defined by its concretization $\gamma_{\mathbb{M}}(\mathrm{m}) \subseteq \Sigma \times (\mathbb{A} \to \mathbb{V})$, as a set of pairs made of a memory state σ and a function ν that maps each symbolic address or variable into its concrete counterpart in σ. The bottom of Fig. 6 gives the definition of $\gamma_{\mathbb{M}}(\mathrm{m})$ and $\gamma_{\mathbb{D}}(\mathrm{d})$ in the main cases. Symbolic addresses are existentially quantified, and thus the concretization of abstract states is unchanged by renaming of symbolic addresses. Similarly, swapping or merging equal nodes preserves the meaning of abstract states.

Abstract transfer functions compute post-conditions for basic statements like assignments, allocation and deallocation operations in a local manner [17]. As an example, we consider abstract state $\mathrm{m} = \mathbf{tree}(\mathtt{t}) \wedge \mathtt{t} = \mathtt{x} \wedge \mathtt{t} \neq \mathbf{0x0}$ and assignment `x = x->l`. As summary $\mathbf{tree}(\mathtt{t})$ is read, it needs to be unfolded. Due to the side constraint $\mathtt{t} \neq \mathbf{0x0}$, the only disjunct created is

$$
\begin{array}{llllllll}
\mathrm{n} & ::= & \alpha & (\alpha \in \mathbb{A}) & \mathrm{c} & ::= & \mathrm{n} \odot \mathbf{0x0} & (\odot \in \{=, \neq\}) \\
 & | & \mathtt{x} & (\mathtt{x} \in \mathbb{X}) & & | & \mathrm{n} = \mathrm{n}' & \\
\mathrm{p} & ::= & \mathbf{emp} & \multicolumn{5}{l}{\text{(empty memory)}} \\
 & | & \mathrm{n} \cdot \mathtt{f} \mapsto \mathrm{n}' & \multicolumn{5}{l}{\text{(single memory cell)}} \\
 & | & \mathbf{ind}(\mathrm{n}) & \multicolumn{5}{l}{\text{(inductive summary predicate)}} \\
 & | & \mathbf{indseg}(\mathrm{n}, \mathrm{n}') & \multicolumn{5}{l}{\text{(segment summary predicate)}} \\
\mathrm{m} & ::= & \multicolumn{6}{l}{(\mathrm{p} * \ldots * \mathrm{p}) \wedge (\mathrm{c} \wedge \ldots \wedge \mathrm{c})} \\
\mathrm{d} & ::= & \multicolumn{6}{l}{\mathrm{m} \vee \ldots \vee \mathrm{m}}
\end{array}
$$

$$
\begin{array}{l}
[\![\mathtt{x}]\!](\sigma, \nu) = \sigma(\mathtt{x}) \qquad\qquad [\![\alpha]\!](\sigma, \nu) = \nu(\alpha) \\
\gamma_{\mathbb{C}}(\mathrm{n} \odot 0) = \{(\sigma, \nu) \mid [\![\mathrm{n}]\!](\sigma, \nu) \odot 0\} \\
\gamma_{\mathbb{M}}(\mathrm{n} \cdot \mathtt{f} \mapsto \mathrm{n}') = \{(\sigma, \nu) \mid \sigma_{\restriction \mathbb{H}} = [[\![\mathrm{n}]\!](\sigma, \nu) + \mathtt{f} \mapsto [\![\mathrm{n}']\!](\sigma, \nu)]\} \\
\gamma_{\mathbb{M}}(\mathbf{ind}(\mathrm{n})) = \bigcup\{\gamma_{\mathbb{M}}(\mathrm{m}) \mid \mathbf{ind}(\mathrm{n}) \xrightarrow{\text{unfold}} \mathrm{m}\} \\
\gamma_{\mathbb{M}}((\mathrm{p}_0 * \ldots * \mathrm{p}_k) \wedge (\mathrm{c}_0 \wedge \ldots \wedge \mathrm{c}_l)) = \\
\quad \{(\sigma, \nu) \mid \exists \sigma_0, \ldots, \sigma_k,\ \sigma_{\restriction \mathbb{H}} = \sigma_{0 \restriction \mathbb{H}} \uplus \ldots \uplus \sigma_{k \restriction \mathbb{H}} \wedge \\
\qquad \forall i,\ (\sigma_{i \restriction \mathbb{X}} = \sigma_{\restriction \mathbb{X}} \wedge (\sigma_i, \nu) \in \gamma_{\mathbb{M}}(\mathrm{p}_i)) \wedge \forall j,\ (\sigma, \nu) \in \gamma_{\mathbb{C}}(\mathrm{c}_j)\} \\
\gamma_{\mathbb{D}}(\mathrm{m}_0 \vee \ldots \vee \mathrm{m}_n) = \gamma_{\mathbb{D}}(\mathrm{m}_0) \cup \ldots \cup \gamma_{\mathbb{D}}(\mathrm{m}_n)
\end{array}
$$

Figure 6. Non-bottom abstract states and concretization main cases.

$\mathtt{t} \cdot \mathtt{l} \mapsto \alpha * \mathtt{t} \cdot \mathtt{r} \mapsto \beta * \mathbf{tree}(\alpha) * \mathbf{tree}(\beta) \wedge \mathtt{t} = \mathtt{x}$. Then, the analysis boils down to a local update, which produces the abstract post-condition $\mathtt{t} \cdot \mathtt{l} \mapsto \mathtt{x} * \mathtt{t} \cdot \mathtt{r} \mapsto \beta * \mathbf{tree}(\mathtt{x}) * \mathbf{tree}(\beta)$.

3.2 Silhouettes

Abstract states provide a precise description of sets of concrete states. Even summarized regions are characterized by inductive predicates that convey very detailed information about their structure. The purpose of silhouettes as presented in Sect. 2 is to identify similarities among abstract states without looking into the details of the shapes of the structures. Thus, we define silhouettes as graphs, yet with edges that retain less information than region predicates. As remarked in Sect. 2, information about reachability on pointer paths is relevant to the characterization of groups of abstract states that could be clumped together. Paths can be described using basic regular expressions over fields. We let $\mathbb{E}$ denote the set of the regular expressions that satisfy the grammar $\mathrm{e} ::= \epsilon \mid \mathtt{f} \mid (\mathtt{f}_0 + \ldots + \mathtt{f}_n)^\star \mid \mathrm{e} \cdot \mathrm{e}$ which are respectively adequate to describe equalities, points-to edges, segment edges, and sequences of edges. This leads us to the following definition:

Definition 1 (Silhouette)**.** *A* silhouette s *is a graph defined by a set of nodes $N \subseteq \mathbb{X} \uplus \mathbb{A}$, and a set of edges E that are labeled by regular expressions in $\mathbb{E}$ (we write $(\mathrm{n}, \mathrm{e}, \mathrm{n}')$ for such an edge).*

A silhouette collects a conjunction of reachability constraints over paths. Thus, the concretization $\gamma_{\mathbb{S}}(\mathrm{s})$ of a silhouette s is defined as the set of all pairs (σ, ν) that satisfy all the constraints defined by the edges of s. Deciding whether (σ, ν) satisfies the constraint defined by an edge $(\mathrm{n}, \mathrm{e}, \mathrm{n}')$ boils down to evaluating the values described by n, n' into values a, a' and checking whether dereferencing a sequence of fields described by e starting from a allows to reach a'. Using the semantics of nodes $[\![\mathrm{n}]\!](.)$ (Fig. 6), the following set of rules formalizes this:

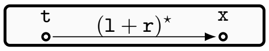
$$\frac{(\sigma,\nu) \vDash [\![\mathrm{n}]\!](\sigma,\nu), \mathrm{e}, [\![\mathrm{n}']\!](\sigma,\nu)}{(\sigma,\nu) \vDash (\mathrm{n}, \mathrm{e}, \mathrm{n}')} \qquad \frac{}{(\sigma,\nu) \vDash a, (\mathtt{f}_0 + \ldots + \mathtt{f}_n)^\star, a}$$

$$\frac{\sigma(a + \mathtt{f}) = a'}{(\sigma,\nu) \vDash a, \mathtt{f}, a'} \qquad \frac{\exists i,\ (\sigma,\nu) \vDash \sigma(a + \mathtt{f}_i), (\mathtt{f}_0 + \ldots + \mathtt{f}_n)^\star, a'}{(\sigma,\nu) \vDash a, (\mathtt{f}_0 + \ldots + \mathtt{f}_n)^\star, a'}$$

$$\frac{}{(\sigma,\nu) \vDash a, \epsilon, a} \qquad \frac{\exists a'',\ (\sigma,\nu) \vDash a, \mathrm{e}, a'' \wedge (\sigma,\nu) \vDash a'', \mathrm{e}', a'}{(\sigma,\nu) \vDash a, \mathrm{e} \cdot \mathrm{e}', a'}$$

For example, the concrete memory shown in Fig. 3 can be described by the silhouette shown below with its graphical representation:

$$\mathrm{s} = \{(\mathtt{t}, (\mathtt{l} + \mathtt{r})^\star, \mathtt{x})\}$$



3.3 Computation of Silhouettes

To allow silhouettes to assist in clumping abstract states, it is crucial to have an efficient way to compute them. First, empty regions, equalities / disequalities to **0x0**, and full inductive predicates (of the form $\mathbf{ind}(\alpha)$) do not contribute to the silhouette. Second, a points-to edge $\mathrm{n} \cdot \mathtt{f} \mapsto \mathrm{n}'$ simply contributes an edge $(\mathrm{n}, \mathtt{f}, \mathrm{n}')$. Last, a segment predicate $\mathbf{indseg}(\mathrm{n}, \mathrm{n}')$ contributes an edge $(\mathrm{n}, \mathcal{E}(\mathbf{ind}), \mathrm{n}')$, where $\mathcal{E}(\mathbf{ind})$ denotes the set of paths that can be induced by a segment of inductive predicate $\mathbf{ind}$: in the case of the tree predicate of Sect. 2, $\mathcal{E}(\mathbf{tree}) = (\mathtt{l} + \mathtt{r})^\star$. Moreover, the silhouette of an abstract state is obtained by collecting the contribution of each region predicate. Therefore the silhouette of an abstract state can be defined as a set of edges computed by the function Π defined by:

$$\begin{array}{ll} \Pi(\mathrm{n} \odot \mathbf{0x0}) = \emptyset & \Pi(\mathrm{n} = \mathrm{n}') = \{(\mathrm{n}, \epsilon, \mathrm{n}')\} \\ \Pi(\mathbf{emp}) = \emptyset & \Pi(\mathrm{n} \cdot \mathtt{f} \mapsto \mathrm{n}') = \{(\mathrm{n}, \mathtt{f}, \mathrm{n}')\} \\ \Pi(\mathbf{ind}(\mathrm{n})) = \emptyset & \Pi(\mathbf{indseg}(\mathrm{n}, \mathrm{n}')) = \{(\mathrm{n}, \mathcal{E}(\mathbf{ind}), \mathrm{n}')\} \\ \multicolumn{2}{l}{\Pi(\mathrm{p}_0 * \ldots * \mathrm{p}_k \wedge \mathrm{c}_0 \wedge \ldots \wedge \mathrm{c}_l) =} \\ \multicolumn{2}{r}{\Pi(\mathrm{p}_0) \cup \ldots \cup \Pi(\mathrm{p}_n) \cup \Pi(\mathrm{c}_0) \cup \ldots \cup \Pi(\mathrm{c}_n)} \end{array}$$

This translation function is sound as it returns a silhouette that describes more memories than the abstract state it is applied to:

Theorem 1 (Soundness)**.** *The silhouette translation function Π is* sound*: for all memory state m, $\gamma_{\mathbb{M}}(\mathrm{m}) \subseteq \gamma_{\mathbb{S}}(\Pi(\mathrm{m}))$.*

The proof proceeds by induction over abstract states.

Silhouettes describe conjunctions of constraints, thus can be weakened into coarser approximations of sets of memory states, either by dropping or by weakening some constraints. Given a silhouette $\mathrm{s} = (N, E)$ and a set of nodes $N' \subseteq N$, we let the *restriction* of s to N' be the silhouette $\mathrm{s}_{\lceil N'}$ defined by the set of nodes N' and the edges obtained as acyclic concatenations of edges of s forming paths from N' to N'. This weakening effectively allows us to ignore some nodes. For instance, if $X \subseteq \mathbb{X}$ is a set of variables that play a special role, then $\Pi(\mathrm{m})_{\lceil X}$ is a silhouette of m that will only retain information about the variables in X. We note this silhouette $\Pi(\mathrm{m}, X)$.

Example 1 (Silhouette computation)**.** *We consider the abstract state m of Fig. 3. Then, $\Pi(\mathrm{m})$ is the silhouette shown at the end of Sect. 3.2. Moreover, $\Pi(\mathrm{m}, \{\mathtt{t}\})$ is $\emptyset$.*

3.4 Silhouette-based Weak Entailment Check

The core purpose of the silhouette abstraction is to determine when abstract states are similar enough to be joined together without a significant loss in precision. Since join computes over-approximation of several abstract states, we first study the relationship between entailment check of abstract states and the silhouettes.

Entailment check for abstract states. Entailment check of abstract states is usually based on sets of proof rules that establish inclusion locally and express separation, reflexivity, and unfolding of inductive summary predicates [4, 9, 17]. The rules below describe such a typical system (for clarity, numerical constraints are elided):

$$\frac{\mathrm{m} \text{ is of the form } \mathrm{n} \cdot \mathtt{f} \mapsto \mathrm{n}' \text{ or } \mathbf{ind}(\mathrm{n}) \text{ or } \mathbf{indseg}(\mathrm{n}, \mathrm{n}')}{\mathrm{m} \sqsubseteq_{\mathbb{M}} \mathrm{m}}$$

$$\frac{\mathrm{m}' \text{ is of the form } \mathbf{ind}(\mathrm{n}) \text{ or } \mathbf{indseg}(\mathrm{n}, \mathrm{n}') \text{ and } \mathrm{m}' \xrightarrow{\text{unfold}} \mathrm{m}}{\mathrm{m} \sqsubseteq_{\mathbb{M}} \mathrm{m}'}$$

$$\frac{\mathrm{m} \sqsubseteq_{\mathbb{M}} \mathbf{ind}(\mathrm{n}')}{\mathbf{indseg}(\mathrm{n}, \mathrm{n}') * \mathrm{m} \sqsubseteq_{\mathbb{M}} \mathbf{ind}(\mathrm{n})} \qquad \frac{\mathrm{m}_0 \sqsubseteq_{\mathbb{M}} \mathrm{m}'_0 \quad \mathrm{m}_1 \sqsubseteq_{\mathbb{M}} \mathrm{m}'_1}{\mathrm{m}_0 * \mathrm{m}_1 \sqsubseteq_{\mathbb{M}} \mathrm{m}'_0 * \mathrm{m}'_1}$$

An entailment checking algorithm noted $\mathbf{leq}_{\mathbb{M}}$ implements a proof search based on these rules. It is in general *sound* and *incomplete*: when $\mathbf{leq}_{\mathbb{M}}(\mathrm{m}_0, \mathrm{m}_1)$ returns $\mathbf{true}$, then $\mathrm{m}_0 \sqsubseteq_{\mathbb{M}} \mathrm{m}_1$ thus $\gamma_{\mathbb{M}}(\mathrm{m}_0) \subseteq \gamma_{\mathbb{M}}(\mathrm{m}_1)$, yet the reverse implication does not hold in general. Indeed, complete proof search algorithms with backtracking are expensive. Moreover, completeness can in general not be ensured in the presence of more complex sets of numerical constraints [9] than in the restricted set of abstract states used here.

Silhouette entailment check. As an alternative to the abstract state entailment check, we propose to first use a weaker and cheaper entailment check on silhouettes, which is based on a classical inclusion of constraints. We let $\mathcal{L}(\mathrm{e})$ denote the language of e.

Definition 2 (Silhouette entailment check)**.** *Let $\mathrm{s}_0, \mathrm{s}_1 \in \mathbb{S}$. We let $\mathbf{leq}_{\mathbb{S}}(\mathrm{s}_0, \mathrm{s}_1)$ return $\mathbf{true}$ if and only if for all edge $(\mathrm{n}, \mathrm{e}, \mathrm{n}')$ of s_1 there exists a (possibly empty) sequence of edges $(\mathrm{n}_0, \mathrm{e}_1, \mathrm{n}_1), \ldots, (\mathrm{n}_{k-1}, \mathrm{e}_k, \mathrm{n}_k)$ in s_0 such that $\mathrm{n}_0 = \mathrm{n}$, $\mathrm{n}_k = \mathrm{n}'$ and $\mathcal{L}(\mathrm{e}_1 \cdot \ldots \cdot \mathrm{e}_k) \subseteq \mathcal{L}(\mathrm{e})$ (or, if the sequence is empty, $\epsilon \in \mathcal{L}(\mathrm{e})$).*

Theorem 2 (Soundness)**.** *The entailment check $\mathbf{leq}_{\mathbb{S}}$ is sound: given $\mathrm{s}_0, \mathrm{s}_1 \in \mathbb{S}$, if $\mathbf{leq}_{\mathbb{S}}(\mathrm{s}_0, \mathrm{s}_1) = \mathbf{true}$, then $\gamma_{\mathbb{S}}(\mathrm{s}_0) \subseteq \gamma_{\mathbb{S}}(\mathrm{s}_1)$.*

The most important result on silhouette entailment check is that it is weaker than the rules for abstract states entailment:

Theorem 3 (Weak entailment)**.** *Let* $\mathrm{m}_0, \mathrm{m}_1 \in \mathbb{M}$. *Then:*

$$\mathrm{m}_0 \sqsubseteq_{\mathbb{M}} \mathrm{m}_1 \Longrightarrow \mathbf{leq}_{\mathbb{S}}(\Pi(\mathrm{m}_0), \Pi(\mathrm{m}_1)) = \mathbf{true}.$$

The proof of Th. 2 follows from the structure of silhouette entailment check whereas Th. 3 can be proved by induction on the derivations of $\sqsubseteq_{\mathbb{M}}$.

From this theorem follows the core principle of our approach: while the most direct way to check if $\gamma_{\mathbb{M}}(\mathrm{m}_0) \subseteq \gamma_{\mathbb{M}}(\mathrm{m}_1)$ consists in applying the (fairly expensive) $\mathbf{leq}_{\mathbb{M}}(\mathrm{m}_0, \mathrm{m}_1)$ proof search algorithm, an alternate approach consists in computing $\mathbf{leq}_{\mathbb{S}}(\Pi(\mathrm{m}_0), \Pi(\mathrm{m}_1))$ first and computing $\mathbf{leq}_{\mathbb{M}}$ only when $\mathbf{leq}_{\mathbb{S}}$ returns $\mathbf{true}$. Indeed, if $\mathbf{leq}_{\mathbb{S}}$ returns $\mathbf{false}$, then $\mathbf{leq}_{\mathbb{M}}$ will also definitely return $\mathbf{false}$, though at a much higher computational cost.

Example 2 (Entailment)**.** *We consider the example given in Sect. 2, and the states and silhouettes defined in Fig. 4. Then,* $\mathbf{leq}_{\mathbb{S}}(\mathrm{s}_0, \mathrm{s}_1) = \mathbf{false}$*, and indeed* $\mathrm{m}_0 \sqsubseteq_{\mathbb{M}} \mathrm{m}_1$ *does not hold. Moreover,* $\mathbf{leq}_{\mathbb{S}}(\mathrm{s}_1, \mathrm{s}_\sim) = \mathbf{true}$ *and* $\mathrm{m}_0 \sqsubseteq_{\mathbb{M}} \mathrm{m}_{1,2,3}$ *holds. On the other hand, among the states in Fig. 4, there is no case where* $\mathbf{leq}_{\mathbb{S}}$ *returns* $\mathbf{true}$ *and* $\sqsubseteq_{\mathbb{M}}$ *does not hold.*

4. Clumping Disjunctions of Abstract States

Silhouettes offer a weak characterization for feasible weakenings. We now put this characterization to work in order to determine when and how disjunctive abstract states may be clumped, without causing a severe loss in precision.

4.1 Existing Abstract States Join Operators and Challenges

To clump a disjunctive abstract state, the analysis should identify disjuncts that can be weakened into the same, precise result. This corresponds to a join operator over finite sets of abstract states. Thus, we first study how a join [9, 38] can be computed for a pair of abstract states m, m'. First, in order to take advantage of separation [32] and local reasoning, abstract states m, m' should be split into separating conjunctions of regions $\mathrm{m} = (\mathrm{m}_0 * \ldots * \mathrm{m}_k)$ and $\mathrm{m}' = (\mathrm{m}'_0 * \ldots * \mathrm{m}'_k)$ such that m_i and m'_i describe "similar" shapes. Second, for each pair $\mathrm{m}_i, \mathrm{m}'_i$ a *local* structural rule should produce a weaker $\mathrm{m}_i^{\sqcup}$, such that inclusions $\mathrm{m}_i \sqsubseteq_{\mathbb{M}} \mathrm{m}_i^{\sqcup}$ and $\mathrm{m}'_i \sqsubseteq_{\mathbb{M}} \mathrm{m}_i^{\sqcup}$ hold. Then, a valid over approximation of both m and m' is defined by $\mathrm{m}^{\sqcup} = \mathrm{m}_0^{\sqcup} * \ldots * \mathrm{m}_k^{\sqcup}$. During this second phase, two main kinds of rules are used:

- **Weakening guided by existing region predicates.** The first set of rules searches for cases where $\mathrm{m}_i \sqsubseteq_{\mathbb{M}} \mathrm{m}'_i$ (resp. $\mathrm{m}'_i \sqsubseteq_{\mathbb{M}} \mathrm{m}_i$) holds so that a valid choice for $\mathrm{m}_i^{\sqcup}$ is m'_i (resp., m_i). The most common case is when $\mathrm{m}_i = \mathrm{m}'_i$. Another important case is when $\mathrm{m}'_i = \mathbf{ind}(\alpha)$ and $\mathrm{m}_i \sqsubseteq_{\mathbb{M}} \mathbf{ind}(\alpha)$; then m_i gets weakened into a summary predicate, that was already present in m'.
- **Synthesis of summary predicates.** The second set of rules *synthesizes* new summary predicates, that weaken specific patterns. For instance, the introduction of a segment predicate is a particular case of summary predicate synthesis, that weakens an empty region into a segment:

$$\left.\begin{array}{r} \mathrm{m}_i = (\mathbf{emp} \wedge \alpha = \beta) \\ \mathrm{m}'_i \sqsubseteq_{\mathbb{M}} \mathbf{indseg}(\alpha, \beta) \end{array}\right\} \rightsquigarrow \mathrm{m}_i^{\sqcup} = \mathbf{indseg}(\alpha, \beta)$$

Example 3 (Abstract states join)**.** *As an example, we let* $\mathrm{m} = (\mathbf{tree}(\mathtt{x}) \wedge \mathtt{t} = \mathtt{x})$ *and* $\mathrm{m}' = (\mathtt{t} \cdot \mathtt{l} \mapsto \mathtt{x} * \mathtt{t} \cdot \mathtt{r} \mapsto \alpha * \mathbf{tree}(\mathtt{x}) * \mathbf{tree}(\alpha))$*. Then,* m' *can be split into two regions: first,* $\mathbf{tree}(\mathtt{x})$ *is also present into* m*; second, the remainder is included into* $\mathbf{treeseg}(\mathtt{t}, \mathtt{x})$*. Therefore, these two states can be joined into* $(\mathbf{treeseg}(\mathtt{t}, \mathtt{x}) * \mathbf{tree}(\mathtt{x}))$*.*

In practice, the splitting phase and the application of weakening rules are performed concurrently, as the search for a good splitting requires insight about what rules may apply.

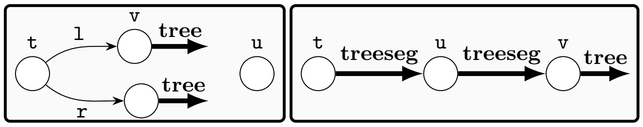


Figure 7. Join of abstract states and splitting.

Challenges. The main difficulty of the abstract join stems from the large search space that needs to be examined to determine whether an adequate splitting can be found. A first caveat is that in certain cases, there exists no splitting that produces a precise common approximation using the lattice defined in Fig. 6: the configurations of Fig. 1(a) and Fig. 1(b) provide examples of such pairs of states. Even when a solution exists, it may be non-unique, and there may be no universally best solution, as illustrated by the following example.

Example 4 (Non-unique splitting)**.** *We let* m, m' *be defined by:*

$$\begin{array}{r} \mathrm{m} = (\mathtt{t} \cdot \mathtt{l} \mapsto \mathtt{v} * \mathtt{t} \cdot \mathtt{r} \mapsto \alpha * \mathbf{tree}(\mathtt{v}) * \mathbf{tree}(\alpha) \wedge \mathtt{u} = \mathtt{t}) \\ \mathrm{m}' = (\mathtt{t} \cdot \mathtt{l} \mapsto \mathtt{v} * \mathtt{t} \cdot \mathtt{r} \mapsto \alpha * \mathbf{tree}(\mathtt{v}) * \mathbf{tree}(\alpha) \wedge \mathtt{u} = \mathtt{v}) \end{array}$$

There are two possible splittings (results are shown in Fig. 7):

1. *pairing each of the four edges of* m *with an edge of* m' *produces a join (shown in the left of Fig. 7) with very precise information about* $\mathtt{t}, \mathtt{v}$*, but that discards all information about* $\mathtt{u}$*;*
2. *introducing an empty segment between* $\mathtt{u}$ *and* $\mathtt{t}$ *in* m *and another empty segment between* $\mathtt{u}$ *and* $\mathtt{v}$ *in* m' *produces a result (shown in the right of Fig. 7) where the fact that* $\mathtt{u}$ *is a cursor in* $\mathtt{t}$ *is preserved, but the information that* $\mathtt{t} \cdot \mathtt{l}$ *points to* $\mathtt{v}$ *is lost.*

In this situation, to make the best clumping choice, the analysis should take into account future uses of $\mathtt{t}, \mathtt{u}, \mathtt{v}$. If $\mathtt{u}$ is unused after the point in the program where clumping is performed, the first splitting is the best. If the fact that $\mathtt{u}$ points somewhere in the tree matters, the second splitting is better. Last, if the fact that $\mathtt{v}$ is a child of $\mathtt{t}$ and the fact that $\mathtt{u}$ points somewhere in the tree both matter, then the best clumping strategy is *not* to join m, m'.

Unary weakening operators such as canonicalization [17] work in a similar manner (though without the guidance of a second argument), and face the same difficulties.

Silhouette-guided clumping of abstract states. In the rest of this section, we demonstrate that the silhouette abstraction brings a twofold gain to the clumping of abstract states. First, it allows us to rule out cases where no "good" splitting can be found, while taking into account information about the analysis goal (Sect. 4.2). Second, once a good splitting is found, it enhances the weakening rules that can be applied and eases the introduction of summary predicates (Sect. 4.3). Both of these advantages follow from the weak entailment decided at the silhouette level as shown in Theorem 3. The clumping proposed below is general and will also be used for widening in Sect. 5.

4.2 Clumping Silhouettes

While the silhouettes of abstract states defined in Sect. 3 provide a weak entailment check, they still adhere too closely to the structure of abstract states to highlight all possibilities for clumping. For instance, Fig. 8 shows excerpts $\mathrm{m}'_2, \mathrm{m}'_3$ of the abstract states $\mathrm{m}_2, \mathrm{m}_3$ of Fig. 4, and the silhouettes of $\mathrm{m}'_2, \mathrm{m}'_3$ are not equal, even though these abstract states could be clumped, as discussed in Sect. 2:

$$\begin{array}{l} \Pi(\mathrm{m}'_2) = \{(\mathtt{t}, \mathtt{l}, \alpha_0), (\alpha_0, (\mathtt{l} + \mathtt{r})^\star, \mathtt{x}), (\mathtt{t}, \mathtt{r}, \alpha_1)\} \\ \Pi(\mathrm{m}'_3) = \{(\mathtt{t}, \mathtt{r}, \beta_1), (\beta_1, (\mathtt{l} + \mathtt{r})^\star, \beta_2), (\beta_2, \mathtt{l}, \mathtt{x}), \ldots\} \end{array}$$

Taking advantage of the analysis goal. As remarked in Sect. 3.3, silhouettes can be made more concise and relevant by collecting only nodes that play an important role in the rest of the analysis. In

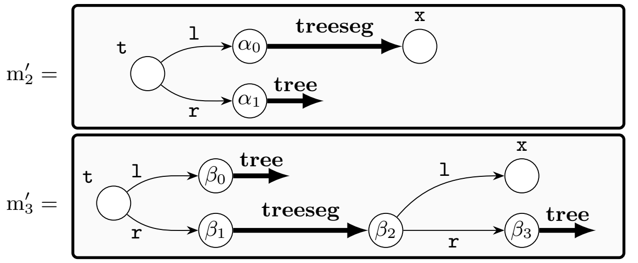


Figure 8. Abstract states candidate for clumping.

the case of the analysis shown in Sect. 2, variables $\mathtt{t}, \mathtt{x}, \mathtt{y}$ are needed in the rest of the analysis of the program, and there is no further dereference of fields of $\mathtt{t}$, thus the edges of the form $(\mathtt{t}, \mathtt{l}, \ldots)$ and $(\mathtt{t}, \mathtt{r}, \ldots)$ are not relevant, and could be folded into summary predicates without hurting the analysis.

Thus, to perform a precise clumping, one restricts the silhouette nodes to those that stand for *live* variables, i.e., variables that may be read later in the program being analyzed before they are overwritten (as computed by a standard liveness compiler analysis [1]).

Example 5. *If we let* $X = \{\mathtt{t}, \mathtt{x}\}$*, we obtain:*

$$\begin{aligned}\Pi(\mathrm{m}'_2, X) &= \{(\mathtt{t}, \mathtt{l} \cdot (\mathtt{l} + \mathtt{r})^\star, \mathtt{x})\} \\ \Pi(\mathrm{m}'_3, X) &= \{(\mathtt{t}, \mathtt{r} \cdot (\mathtt{l} + \mathtt{r})^\star \cdot \mathtt{l}, \mathtt{x})\}\end{aligned}$$

Though these silhouettes are not equal, they both resemble that of an abstract state with a segment between $\mathtt{t}$ *and* $\mathtt{x}$*.*

Generalizing silhouettes. To recognize silhouette similarities in configurations such as that of Ex. 5, constraints should be generalized so as to make segment patterns easier to recognize at the silhouette level. Since segments correspond to regular expressions of the form $(\mathtt{f}_0 + \ldots + \mathtt{f}_k)^\star$, the following generalization function makes such patterns appear more prominently:

Definition 3 (Silhouette generalization)**.** *The* generalization $\phi(\mathrm{e})$ *of a regular expression* e *of the form* $\mathtt{f}'_0 \cdot \ldots \cdot \mathtt{f}'_m \cdot (\mathtt{f}_0 + \ldots + \mathtt{f}_k)^\star \cdot \mathtt{f}''_0 \cdot \ldots \cdot \mathtt{f}''_n$*, where* $\{\mathtt{f}'_0, \ldots, \mathtt{f}'_m, \mathtt{f}''_0, \ldots, \mathtt{f}''_n\} \subseteq \{\mathtt{f}_0, \ldots, \mathtt{f}_k\}$*, is the regular expression* $\phi(\mathrm{e}) = (\mathtt{f}_0 + \ldots + \mathtt{f}_k)^\star$*. For all other regular expressions* e*, we let* $\phi(\mathrm{e}) = \mathrm{e}$*.*

The generalization *of a silhouette* s *is obtained by replacing any edge* $(\mathrm{n}, \mathrm{e}, \mathrm{n}')$ *of* s *by edge* $(\mathrm{n}, \phi(\mathrm{e}), \mathrm{n}')$*. It is noted* $\Phi(\mathrm{s})$*.*

This operation defines a weakening over silhouettes, since for all silhouettes s, we observe that $\gamma_{\mathbb{S}}(\mathrm{s}) \subseteq \gamma_{\mathbb{S}}(\Phi(\mathrm{s}))$.

Example 6. *After generalization, we obtain:*

$$\Phi(\Pi(\mathrm{m}'_2), \{\mathtt{t}, \mathtt{x}\}) = \{(\mathtt{t}, (\mathtt{l} + \mathtt{r})^\star, \mathtt{x})\} = \Phi(\Pi(\mathrm{m}'_3), \{\mathtt{t}, \mathtt{x}\})$$

Clumping relation. We now consider a criterion for the computing candidate groups of abstract states for clumping based on silhouettes. As observed in Sect. 4.1, a large part of the join rules, implicitly rely on $\sqsubseteq_{\mathbb{M}}$. Therefore, $\mathbf{leq}_{\mathbb{S}}$ can be used to prune out abstract state entailment checks that do not hold in the silhouette (Th. 3). Though, we note that, during the computation of a single join of two abstract states m, m', such weakening rules may need to be applied on both sides. Thus, we need a symmetric characterization of situations where a weakening may be performed either in m, in m′ or in both, which is the purpose of relation $\bowtie$:

Definition 4 (Silhouette association relation)**.** *Let* $\mathrm{s}_0, \mathrm{s}_1$ *be two silhouettes with the same set of nodes* N*. We let* $\mathrm{s}'_0 = \Phi(\mathrm{s}_0)$ *and* $\mathrm{s}'_1 = \Phi(\mathrm{s}_1)$*. We write* $\mathrm{s}_0 \bowtie \mathrm{s}_1$ *if and only if there exist* N_0, N_1 *such that* $N = N_0 \cup N_1$ *and:*

$$\begin{array}{rcl} & \mathrm{s}'_0 = \mathrm{s}'_{0\lceil N_0} \cup \mathrm{s}'_{0\lceil N_1} \quad \wedge & \mathbf{leq}_{\mathbb{S}}(\mathrm{s}'_{0\lceil N_0}, \mathrm{s}'_{1\lceil N_0}) = \mathbf{true} \\ \wedge & \mathrm{s}'_1 = \mathrm{s}'_{1\lceil N_0} \cup \mathrm{s}'_{1\lceil N_1} \quad \wedge & \mathbf{leq}_{\mathbb{S}}(\mathrm{s}'_{1\lceil N_1}, \mathrm{s}'_{0\lceil N_1}) = \mathbf{true} \end{array}$$

This relation is symmetric and reflexive, but not transitive:

Example 7. *Let* $\mathrm{s}_0, \mathrm{s}_1, \mathrm{s}_2$ *be defined by:*

$$\begin{array}{ll} \mathrm{s}_0 = \{(\mathtt{x}, \epsilon, \mathtt{y}), (\mathtt{y}, \mathtt{f}^\star, \mathtt{z})\} & \mathrm{s}_2 = \{(\mathtt{x}, \epsilon, \mathtt{y}), (\mathtt{y}, \mathtt{f}, \mathtt{z})\} \\ \mathrm{s}_1 = \{(\mathtt{x}, \mathtt{f}^\star, \mathtt{y}), (\mathtt{y}, \epsilon, \mathtt{z})\} & \end{array}$$

Then, $\mathrm{s}_0 \bowtie \mathrm{s}_1$ *and* $\mathrm{s}_0 \bowtie \mathrm{s}_2$ *hold, but* $\mathrm{s}_1 \bowtie \mathrm{s}_2$ *does not hold.*

The silhouette association relation soundly characterizes the cases where a precise join can be computed using rules that perform weakening guided by existing predicates:

Theorem 4 (Silhouette association and weakening)**.** *Let* m, m' *be two abstract states. We assume that* $\mathrm{m} = \mathrm{m}_0 * \mathrm{m}_1$ *and* $\mathrm{m}' = \mathrm{m}'_0 * \mathrm{m}'_1$*. Then:*

$$\mathrm{m}_0 \sqsubseteq_{\mathbb{M}} \mathrm{m}'_0 \wedge \mathrm{m}'_1 \sqsubseteq_{\mathbb{M}} \mathrm{m}_1 \Longrightarrow \Phi(\Pi(\mathrm{m})) \bowtie \Phi(\Pi(\mathrm{m}'))$$

This result is a direct consequence of Theorem 3. It implies that, whenever a join can be computed by weakening part of m into predicates present in m′ and part of m′ into predicates present in m, then relation $\bowtie$ holds. The contraposition entails that, when $\bowtie$ does not hold, no precise join can be found using only rules that weaken abstract states based on existing predicates. Therefore, Th. 4 will prevent clumping from attempting to compute some joins that will fail, but not all. Sect. 6 provides evidence that the characterization it provides is actually tight. Moreover, this result does not immediately apply to the cases where join requires the synthesis of summary predicates; Sect. 4.3 discusses such cases.

Example 8 (Clumping relation)**.** *We consider the states of Ex. 4. If* $X = \{\mathtt{t}, \mathtt{v}\}$*, then* $\Pi(\mathrm{m}, X) = \{(\mathtt{t}, \mathtt{l}, \mathtt{v})\} = \Pi(\mathrm{m}', X)$*. Thus,* $\Phi(\Pi(\mathrm{m}, X)) \bowtie \Phi(\Pi(\mathrm{m}', X))$ *holds. As observed in Ex. 4, joining these two states is precise with respect to* X*.*
On the other hand, if $X = \{\mathtt{t}, \mathtt{u}, \mathtt{v}\}$ *(we observed that joining them would incur a precision loss), we obtain* $\Pi(\mathrm{m}, X) = \{(\mathtt{t}, \epsilon, \mathtt{u}), (\mathtt{u}, \mathtt{l}, \mathtt{v})\}$ *and* $\Pi(\mathrm{m}', X) = \{(\mathtt{t}, \mathtt{l}, \mathtt{v}), (\mathtt{u}, \epsilon, \mathtt{v})\}$*, and* $\Phi(\Pi(\mathrm{m}, X)) \bowtie \Phi(\Pi(\mathrm{m}', X))$ *does not hold.*

4.3 Silhouette Guided Join of Abstract States

We now show that silhouettes may help synthesizing new summary predicates. For instance, when a segment predicate can be synthesized, it is always done after checking with relation $\sqsubseteq_{\mathbb{M}}$ that the new predicate actually defines a weakening of the arguments. Since Th. 3 offers a weak entailment check, the silhouette can be used in order to guide introduction of segment predicates.

Definition 5 (Silhouette join)**.** *We define the* join of silhouette edges $(\mathrm{n}_0, \mathrm{e}, \mathrm{n}_1)$ *and* $(\mathrm{n}_0, \mathrm{e}', \mathrm{n}_1)$ *as the edge* $(\mathrm{n}_0, \mathrm{e}^\sqcup, \mathrm{n}_1)$ *where* $\mathrm{e}^\sqcup$ *keeps the same left side as* e *and* e' *and joins the other part into the smallest and more general regular expression of the form* $(\mathtt{f}_0 + \ldots + \mathtt{f}_k)^\star$*. Given two silhouettes* $\mathrm{s} = (N, E), \mathrm{s}' = (N', E')$*, we define their* silhouette join *noted* $\mathbf{join}_{\mathbb{S}}(\mathrm{s}, \mathrm{s}')$ *as the silhouette* $\mathrm{s}^\sqcup = (N^\sqcup, E^\sqcup)$ *such that* $N^\sqcup = N \cap N'$ *and* $E^\sqcup$ *collect the pairwise join of the edges of* $\mathrm{s}_{\lceil N^\sqcup}$ *and of* $\mathrm{s}'_{\lceil N^\sqcup}$*.*

When two abstract states can be joined into a segment, their silhouette is a refinement of the silhouette of a segment (by Th. 3):

Theorem 5 (Guided segment synthesis weak characterization)**.** *Let* m, m' *be two abstract states such that* $\mathrm{m} \sqsubseteq_{\mathbb{M}} \mathbf{indseg}(\mathrm{n}_0, \mathrm{n}_1)$ *and* $\mathrm{m}' \sqsubseteq_{\mathbb{M}} \mathbf{indseg}(\mathrm{n}_0, \mathrm{n}_1)$*, then*

$$\begin{array}{c}\gamma_{\mathbb{S}}(\mathbf{join}_{\mathbb{S}}(\Pi(\mathrm{m}, \{\mathrm{n}_0, \mathrm{n}_1\}), \Pi(\mathrm{m}', \{\mathrm{n}_0, \mathrm{n}_1\}))) \\ \subseteq \gamma_{\mathbb{S}}(\{(\mathrm{n}_0, \mathcal{E}(\mathbf{ind}), \mathrm{n}_1)\})\end{array}$$

```
input:   disjunctive abstract state $\mathrm{m}_0 \vee \ldots \vee \mathrm{m}_n$
         set of live variables $X$
output:  clumped disjunctive abstract state $\mathrm{m}'_0 \vee \ldots \vee \mathrm{m}'_k$
$\mathrm{s}_0 \leftarrow \Pi(\mathrm{m}_0, X); \ldots; \mathrm{s}_n \leftarrow \Pi(\mathrm{m}_n, X)$;
$\mathrm{s}'_0 \leftarrow \Phi(\mathrm{s}_0); \ldots; \mathrm{s}'_n \leftarrow \Phi(\mathrm{s}_n)$;
computation of relation $\bowtie$ over $\mathrm{s}'_0, \ldots, \mathrm{s}'_n$
  and of the connected components $S_0, \ldots, S_k$ of $\bowtie$
for each connected component $S_j$ of $\bowtie$
    sort the elements of $S_j$ into $\mathrm{s}'_{i_0} \prec \ldots \prec \mathrm{s}'_{i_l}$
      such that $\forall p,\ \mathrm{s}'_{i_p} \bowtie \mathrm{s}'_{i_{p+1}}$
    $\mathrm{m}'_j \leftarrow \mathrm{m}_{i_0}; \mathrm{s}^{\sqcup} \leftarrow \mathrm{s}_{i_0}$;
    for $p = 1$ to $l$
        $\mathrm{s}^{\sqcup} \leftarrow \mathbf{join}_{\mathbb{S}}(\mathrm{s}^{\sqcup}, \mathrm{s}_{i_p})$;
        $\mathrm{m}'_j \leftarrow \mathbf{join}_{\mathbb{M}}(\mathrm{m}'_j, \mathrm{m}_{i_p}, \mathrm{s}^{\sqcup})$;
```

Figure 9. Clumping algorithm **clump**.

We derive the semantic guided segment introduction rule below:

$$\begin{array}{ll}
\text{let } \mathrm{s} = \Pi(\mathrm{m}, \{\mathrm{n}_0, \mathrm{n}_1\}) \text{ and } \mathrm{s}' = \Pi(\mathrm{m}', \{\mathrm{n}_0, \mathrm{n}_1\}) & \\
\quad \text{if} & \mathbf{join}_{\mathbb{S}}(\mathrm{s}, \mathrm{s}') = (\mathtt{f}_0 + \ldots + \mathtt{f}_k)^{\star} \\
\quad \text{and} & \mathbf{leq}_{\mathbb{S}}(\mathbf{join}_{\mathbb{S}}(\mathrm{s}, \mathrm{s}'), \{(\mathrm{n}_0, \mathcal{E}(\mathbf{ind}), \mathrm{n}_1)\}) = \mathbf{true}, \\
\quad\quad \text{then if} & \mathbf{leq}_{\mathbb{M}}(\mathrm{m}, \mathbf{indseg}(\mathrm{n}_0, \mathrm{n}_1)) = \mathbf{true} \\
\quad\quad \text{and} & \mathbf{leq}_{\mathbb{M}}(\mathrm{m}', \mathbf{indseg}(\mathrm{n}_0, \mathrm{n}_1)) = \mathbf{true}, \\
\quad\quad\quad \text{then,} & \mathbf{join}_{\mathbb{M}}(\mathrm{m}, \mathrm{m}') = \mathbf{indseg}(\mathrm{n}_0, \mathrm{n}_1)
\end{array}$$

Intuitively, this rule will only attempt to synthesize a segment when the join of the silhouettes can be weakened into the silhouette of a segment. Only in that case will it call the shape inclusion checking function $\mathbf{leq}_{\mathbb{M}}$ to determine if a segment can be introduced. This rule supersedes the classical syntactic rule and avoids many attempts to run the costly $\mathbf{leq}_{\mathbb{M}}$, which would fail (since $\mathbf{leq}_{\mathbb{S}}$ provides a cheaper, weak entailment test). In the following, we let $\mathbf{join}_{\mathbb{M}}$ denote a join operator for abstract states, that takes as input as a third argument for a silhouette that it uses as a guide to introduce segments.

Example 9. *We consider the abstract states* $\mathrm{m}'_2, \mathrm{m}'_3$ *of Fig. 8. Then:*

$$\mathrm{s}^{\sqcup} = \mathbf{join}_{\mathbb{S}}(\Pi(\mathrm{m}'_2), \Pi(\mathrm{m}'_3)) = \{(\mathtt{t}, (\mathtt{l} + \mathtt{r})^{\star}, \mathtt{x})\}$$

Thus, the above rule applies, and synthesizes a segment predicate:

$$\mathbf{join}_{\mathbb{M}}(\mathrm{m}'_2, \mathrm{m}'_3, \mathrm{s}^{\sqcup}) = \mathbf{treeseg}(\mathtt{t}, \mathtt{x})$$

This result is beyond the reach of algorithms based on local syntactic rules such as those of [9, 38].

4.4 Clumping Abstract States

Fig. 9 summarizes the algorithm **clump** for clumping abstract states. It takes as input a disjunctive abstract state and the set of live variables at the current location in the program being analyzed. It first computes the silhouette of all the abstract states and the generalized form of these silhouettes. The generalized forms are used in order to compute $\bowtie$. The groups of abstract states to be clumped together are defined by the connected components of this relation and is denoted $\sim$ in Sect. 2. Also the silhouette abstraction denoted $\theta(\mathrm{m})$ in Sect. 2 is computed as $\Phi(\Pi(\mathrm{m}, X))$. For each group, relation $\bowtie$ suggests a sequence of abstract state joins, that are expected to preserve silhouettes. These abstract state joins utilize the silhouettes $\mathrm{s}_0, \ldots, \mathrm{s}_n$ in order to enable semantic-guided synthesis of summary predicates as shown in Sect. 4.3.

This clumping algorithm is *sound*: it returns a disjunctive abstract state that over-approximates $\mathrm{m}_0, \ldots, \mathrm{m}_n$.

Theorem 6 (Clumping soundness)**.** *For all disjunctive abstract state* $\mathrm{m}_0 \vee \ldots \vee \mathrm{m}_n$ *and for all set of variables* X*, we have:*

$$\gamma_{\mathbb{D}}(\mathrm{m}_0 \vee \ldots \vee \mathrm{m}_n) \subseteq \gamma_{\mathbb{D}}(\mathbf{clump}(\mathrm{m}_0 \vee \ldots \vee \mathrm{m}_n, X))$$

Following Th. 4 and Th. 5, this algorithm will not attempt to compute abstract state joins that will not succeed to produce a precise result in the silhouette level. As observed previously, this does not mean all abstract state joins computed by the algorithm of Fig. 9 will keep adequate precision, though the experimental results of Sect. 6 did not show any occurrence of such a precision loss.

Example 10 (Clumping)**.** *When applied to the abstract states of Fig. 4, this algorithm clumps* $\mathrm{m}_1, \mathrm{m}_2, \mathrm{m}_3$ *into the abstract state* $\mathrm{m}_{1,2,3}$ *of Fig. 5. At the same time, it keeps* m_0 *separate.*

4.5 On the Essence of the Silhouette Abstraction

We now propose to discuss, in a general setting, the way the silhouette abstraction works. We started with an abstraction of heaps that is used in shape analysis, and identified a cause of precision loss when computing joins of abstract states. In general, when abstract states that are too dissimilar and do not join well, the analysis will lose critical properties about shapes, that cannot be recovered: indeed, when paths of two abstract states do not match, they cannot be joined into an abstract state that preserves paths, thus join will discard important pointer properties. The purpose of the silhouette is to offer a cheaper characterization of abstract states that the analysis should not attempt to join by the means of another abstraction, which focuses on paths (Definition 1). Theorem 3 illustrates this property: when paths do not match, silhouette inclusion does not hold, which entails that the inclusion of abstract states cannot hold either. Theorem 4 extends this property to union.

Remark 1 (Silhouettes in static analysis: a numerical example)**.** *We can extend this approach to other situations. We consider in this paragraph programs that manipulate integer variables, and perform basic arithmetic operations. A common goal for static analyses consists of verifying the absence of division by zero errors. Let us assume such an analysis relies on interval constraints over program variables: a division by* `x` *can be proved safe if the analysis computes for* `x` *either an interval of strictly negative numbers or an interval of strictly positive numbers. We note that joining together an interval of strictly negative numbers and an interval of strictly positive numbers will return an interval that contains zero. Thus, after such a join, a crucial information will be lost to verify that a division by* `x` *is safe.*

The sign abstraction provides an obvious characterization what interval unions will produce such an imprecise result: the sign abstract domain is made of four abstract values $\bot, \ominus, \oplus, \top$ *that respectively denote the empty set, any set of strictly negative integers, any set of strictly positive integers, and any set of integers. For instance, if two intervals have sign* $\ominus$*, they can be joined without losing information with respect to zero, whereas the join of an interval with sign* $\ominus$ *with an interval with sign* $\oplus$ *will cause a precision loss.*

Therefore, a principle similar to silhouettes allows us to enhance the analysis precision, with respect to the goal of proving variables are not equal to zero:

- *disjunctive states composed of finite disjunctions of intervals can keep more precise information about sets of states where a variable* `x` *may be positive or negative;*
- *the "silhouette" of intervals defined by their sign abstraction characterizes when joining two intervals will cause a precision loss, with respect to the property of interest.*

5. Widening Disjunctive Abstract States

To infer loop invariants from an abstract pre-condition, the analysis needs to use a *widening* operator [10] $\mathbf{widen}_{\mathbb{D}}$ over disjunctive abstract states. This operator should over-approximate concrete union, and ensure that any sequence $(\mathrm{d}_n)_n$ of the form $\mathrm{d}_{n+1} = \mathbf{widen}_{\mathbb{D}}(\mathrm{d}_n, \mathrm{d}'_n)$ terminates. In this section, we assume that $\mathbf{widen}_{\mathbb{M}}$ is a widening over abstract states, using similar algorithms as $\mathbf{join}_{\mathbb{M}}$. Such an operator can often be based on $\mathbf{join}_{\mathbb{M}}$ [9].

Widening based on silhouette-guided pairing. Intuitively, a widening of d with d′ should widen disjuncts of d with disjuncts of d′, using some sort of pairing to select which pairs of disjuncts are to be widened together. Since silhouettes aim at capturing abstract states that can be joined precisely, we build our widening around a pairing function that we shall build based on the silhouette.

Definition 6 (Pairing function)**.** *Given* $\mathrm{d} = \mathrm{m}_0 \vee \ldots \vee \mathrm{m}_n$ *and* $\mathrm{d}' = \mathrm{m}'_0 \vee \ldots \vee \mathrm{m}'_{n'}$*, a* pairing *of* d, d' *is a function* $\pi_{\mathrm{d},\mathrm{d}'}$ *from* $\{0, \ldots, n\}$ *into the powerset of* $\{0, \ldots, n'\}$ *such that* $i \neq j$ *implies* $\pi(i) \cap \pi(j) = \emptyset$*.*

If a pairing family $\pi_{\mathrm{d},\mathrm{d}'}$ is defined for all $\mathrm{d}, \mathrm{d}' \in \mathbb{D}$, we can define an operator over disjunctive abstract states that lets the pairing function define which disjuncts of the right argument are widened with each disjunct of the left argument, and preserves the disjuncts of the right argument that do not appear in the pairing.

Definition 7. *Let* $\mathrm{d} = \mathrm{m}_0 \vee \ldots \vee \mathrm{m}_n$ *and* $\mathrm{d}' = \mathrm{m}'_0 \vee \ldots \vee \mathrm{m}'_{n'}$*. We let* $\mathbf{widen}_{\mathbb{D}}(\mathrm{d}, \mathrm{d}')$ *be defined as the abstract state* $\mathrm{m}''_0, \ldots, \mathrm{m}''_{n+k}$*, where:*

- $\mathrm{m}''_i = \mathbf{widen}_{\mathbb{M}}(\mathrm{m}_i, \mathbf{join}_{\mathbb{M}}(\{\mathrm{m}'_l \mid l \in \pi_{\mathrm{d},\mathrm{d}'}(i)\}))$ *if* $i \leq n$*;*
- $\{\mathrm{m}''_{n+i} \mid 1 \leq i \leq k\} = \{\mathrm{m}'_j \mid \forall i,\ j \notin \pi_{\mathrm{d},\mathrm{d}'}(i)\}$*.*

This operator always returns a sound over-approximation of its arguments. Yet, depending on the pairing family, it may fail to guarantee termination. In the following, we define a pairing family that ensures termination so that $\mathbf{widen}_{\mathbb{D}}$ indeed defines a widening.

A good pairing should map abstract states that produce a precise widening, in the same way as clumping for join. It should also drive the introduction of summary predicates. As observed in Sect. 4, silhouettes hold information capable of guiding this process. However, for termination, we need to bound silhouettes. Thus, we define a pairing that is parameterized by an integer bound b, and that associates abstract states with silhouettes that are equivalent when regular expressions are smashed upon exceeding length b (in the following, we let $b = 1$):

Definition 8 (Silhouette-based pairing)**.** *We let* $\mathbb{F}$ *denote the set of all field names, and we define the regular expression bounded abstraction by* $\omega_b(\mathrm{e}_1 \cdot \ldots \cdot \mathrm{e}_k) = \mathrm{e}_1 \cdot \ldots \cdot \mathrm{e}_b \cdot \mathbb{F}^\star$ *if* $k > b$ *and* $\omega_b(\mathrm{e}_1 \cdot \ldots \cdot \mathrm{e}_k) = \mathrm{e}_1 \cdot \ldots \cdot \mathrm{e}_k \cdot \mathbb{F}^\star$ *otherwise. We let function* Ω_b *abstract a silhouette by replacing each edge* $(\mathrm{n}, \mathrm{e}, \mathrm{n}')$ *by* $(\mathrm{n}, \omega_b(\mathrm{e}), \mathrm{n}')$*.*

Last, if s, s' *are silhouettes, then we write* $\mathrm{s} \rtimes_b \mathrm{s}'$ *if and only if* $\mathbf{leq}_{\mathbb{S}}(\Omega_b(\mathrm{s}'), \Omega_b(\mathrm{s})) = \mathbf{true}$*.*

Theorem 7 (Widening)**.** *The operator* $\mathbf{widen}_{\mathbb{D}}$ *defined using the pairing of Def. 8 enforces termination of abstract iterates.*

The proof relies on the finiteness of the image of Ω_b, which entails that, for any sequence $(\mathrm{d}_n)_n$ of widened iterates (such that $\mathrm{d}_{n+1} = \mathbf{widen}_{\mathbb{D}}(\mathrm{d}_n, \mathrm{d}'_n)$), the disjuncts in d_n eventually stabilize to a set corresponding to a fixed, finite set of silhouettes.

Example 11. *Fig. 10 displays a few of the disjuncts that arise in the second and third iteration over the first loop in the analysis of the code presented in Fig. 2. Disjunct* m_0 *occurs in the second iteration whereas disjuncts* $\mathrm{m}'_0, \mathrm{m}'_1, \mathrm{m}'_2$ *arise in the third iteration. For clarity, we only show the nodes on the path between* `t`*,* `x` *and* `y`*. In the table below, we show their silhouettes before and after applying the bounded abstraction:*

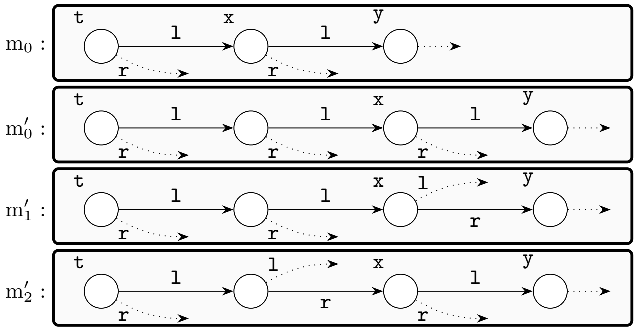


Figure 10. Widening disjunctive abstract states.

	silhouette	*bounded abstraction*
m_0	$\{(\mathtt{t}, \mathtt{l}, \mathtt{x}), (\mathtt{x}, \mathtt{l}, \mathtt{y})\}$	$\{(\mathtt{t}, \mathtt{l} \cdot \mathbb{F}^\star, \mathtt{x}), (\mathtt{x}, \mathtt{l} \cdot \mathbb{F}^\star, \mathtt{y})\}$
m'_0	$\{(\mathtt{t}, \mathtt{l} \cdot \mathtt{l}, \mathtt{x}), (\mathtt{x}, \mathtt{l}, \mathtt{y})\}$	$\{(\mathtt{t}, \mathtt{l} \cdot \mathbb{F}^\star, \mathtt{x}), (\mathtt{x}, \mathtt{l} \cdot \mathbb{F}^\star, \mathtt{y})\}$
m'_1	$\{(\mathtt{t}, \mathtt{l} \cdot \mathtt{l}, \mathtt{x}), (\mathtt{x}, \mathtt{r}, \mathtt{y})\}$	$\{(\mathtt{t}, \mathtt{l} \cdot \mathbb{F}^\star, \mathtt{x}), (\mathtt{x}, \mathtt{r} \cdot \mathbb{F}^\star, \mathtt{y})\}$
m'_2	$\{(\mathtt{t}, \mathtt{l} \cdot \mathtt{r}, \mathtt{x}), (\mathtt{x}, \mathtt{l}, \mathtt{y})\}$	$\{(\mathtt{t}, \mathtt{l} \cdot \mathbb{F}^\star, \mathtt{x}), (\mathtt{x}, \mathtt{l} \cdot \mathbb{F}^\star, \mathtt{y})\}$

Then, $\Pi(\mathrm{m}_0) \rtimes_b \Pi(\mathrm{m}'_0)$ *and* $\Pi(\mathrm{m}_0) \rtimes_b \Pi(\mathrm{m}'_2)$ *hold whereas* $\Pi(\mathrm{m}_0) \rtimes_b \Pi(\mathrm{m}'_1)$ *does not. Thus, the pairing is defined by* $\pi(0) = \{0, 2\}$*. The widening will thus preserve the information which says whether* `y` *is the left or right child of* `x`*.*

Static analysis. We now summarize the whole analysis. As a forward abstract interpretation [10], it starts with an abstract precondition that over-approximates initial memory states with a single disjunct (either the set of all memory states, with no structure allocated for whole program analysis, or a pre-condition describing the valid call states for the analysis of a library function) and computes abstract post-conditions for each statement. Post-conditions of assignment, test, allocation and deallocation statements are computed locally on each disjunct in parallel, as discussed in Sect. 3.1. When the analysis computes disjunctive abstract states d, d' before a control flow join, it applies clumping (Sect. 4) to $\mathrm{d} \vee \mathrm{d}'$. For loops, standard fixpoint approximation techniques [10] apply, and the silhouette guided $\mathbf{widen}_{\mathbb{D}}$ operator (Th. 7) enforces convergence of disjunctive abstract states sequences. The analysis is sound and returns abstract post-conditions that over-approximate final states.

6. Experimental Evaluation

Research hypotheses. In this section, we empirically evaluate whether or not semantic-directed clumping is effective in improving the static analysis of data structures from real-world C libraries. We implemented clumping inside the MemCAD analyzer [36]. We seek to provide evidence for or against the following hypotheses:

RH1 (Clumping is effective). *Semantic-directed clumping with guided join is necessary and effective for analyzing data structure operations from existing, real-world libraries.* The underlying premise of this paper is that inferring disjunctive loop invariants is necessary to effectively analyze real-world data structures. While certain code may be reasonably adapted to avoid the need for disjunctive invariants, analyzing existing, real-world code typically involves handling corner cases in separate disjuncts. We assess the impact of silhouettes on analysis and compare with other techniques.

RH2 (Guided join is necessary). *Guided join is necessary to avoid unacceptable precision loss.* Semantic-directed clumping uses silhouette abstraction to select the disjuncts to join. Guided join then subsequently uses these same silhouettes to perform the shape join. We seek to evaluate the need for this latter step.

RH3 (Clumping has low overhead). *The overhead of semantic-directed clumping is reasonable.* We hope and expect that the additional overhead in computing and comparing silhouettes is outweighed by the benefit of increased precision and improved scalability. This aspect must be tested empirically as the number of silhouette comparisons is quadratic in the number of disjuncts.

RH4 (Clumping limits disjunctive explosion). *Semantic-directed clumping improves the scalability of disjunctive analysis by limiting the number of disjuncts in the abstract state.* Without clumping, the number of disjuncts grows exponentially from control-flow paths and unfolding of inductive predicates. The scalability of disjunctive analysis (and thus most shape analyses) is limited by the exponential growth in the number of disjuncts in the abstract state. Thus the technical challenge is to avoid piling up more and more unnecessary disjuncts while analyzing sequences of operations.

Experimental methodology. To evaluate the effectiveness of semantic-based clumping, we consider 26 benchmarks of varying implementation styles and degrees of difficulty to analyze. These include operations over singly-linked and doubly-linked lists, as well as various binary trees with different kinds of invariants and pointer patterns (search, splay, red black, or AVL and with various sharing patterns). The operations consist of variants of finding an element, inserting, deleting, reversing, and sorting. Notably 21 benchmarks are from external sources. Some of these come with typically simple, user-specified assertions (e.g., x != NULL). Each benchmark consists of a top-level function implementing a data structure operation with a pre- and a post-condition that specify the preservation of precise shape invariants. Some routines are recursive whereas the others contain nested loops. Table 1 lists these benchmarks with metrics to get a sense of the difficulty to analyze, including presence of recursion, lines of code, numbers of sub-routines, numbers of loops and acyclic paths. The latter gives the number of disjuncts that a naïve path and context-sensitive analysis would have at the exit point. The final metric is the maximum number of simultaneous pointers into a data structure instance (column Simult. Pointer). The summary row gives total counts of LOCs, assertions, functions, loops and paths, and the average number of simultaneous pointers. As discussed in Section 1, the number of simultaneous pointers into an instance is related to the number of disjuncts needed to represent the program invariant.

The analysis is parameterized by the definition of **ind** (or infers it for basic list and tree cases). It attempts to prove memory safety, any user-specified assertions, and the (more complex) post-condition given a C program and optionally inductive predicates summarizing memory regions. Clumping is applied at the beginning and end of functions, at loop heads, at loop exits, and at the end of branches.

We evaluate and compare the following clumping strategies:

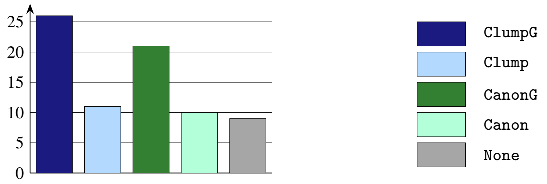
- `ClumpG` and `Clump` do silhouette-guided clumping (Sect. 4.4); `ClumpG` uses guided join (Sect. 4.3), whereas `Clump` does not;
- `None` is the baseline technique, that does not compute silhouettes to perform clumping or guided joining;
- `Canon` and `CanonG` conservatively model canonicalization operators [17, 35]: they compute silhouettes but only join abstractions when their silhouettes are exactly the same (after folding nodes which are not pointed to by live variables); `CanonG` uses guided joining whereas `Canon` does not (we expect these strategies will still compute fewer disjunctions than a purely syntactic canonicalization would).

Benchmark	LOC	User Assert	Fun	Loop	Path	Simult. Pointer
singly-linked list						
sll-delmin	25	0	1	1	12	5
sll-delmin†	26	0	2	1†	6	5
sll-delminmax	49	0	1	1	248	7
sll-delminmax†	52	0	2	1†	124	7
binary search tree						
bstree-find	26	0	1	1	4	3
bstree-find†	26	0	2	1†	4	3
Predator singly-linked list						
psll-reverse	11	0	1	1	2	3
psll-isort	20	0	1	2*	5	5
psll-bsort	25	0	1	2*	10	4
GDSL doubly-linked list (back pointers) with sentinel head and tail						
gdll-findmin	49	14	8	1	3	5
gdll-index	55	14	9	2	24	2
gdll-findmax	58	14	8	1	3	5
gdll-find	78	26	10	1	18	5
gdll-delete	107	26	12	1	72	5
GDSL binary search tree with leaf-to-root and back pointers						
gbstree-find	53	8	7	1	20	3
gbstree-insert	133	15	12	1	7680	5
gbstree-delete	165	9	15	1	23040	10
BSD splay tree						
bsplay-find	81	0	4	1	56	5
bsplay-delete	95	0	4	2	448	5
bsplay-insert	101	0	4	1	43	5
BSD red black tree with back pointers						
brbtree-find	29	0	3	1	4	2
brbtree-insert	177	0	4	2	3036	7
brbtree-delete	329	0	5	3	$1.e+8$	12
JSW AVL tree						
javl-find	25	0	3	1	26	2
javl-free	27	0	3	1	3	3
javl-insert	95	0	6	2	$1.e+8$	6
summary	1917	126	123	34	$2.e+8$	5.0

Table 1. List of benchmarks, divided into internal, microbenchmarks (top) and benchmarks from external sources (bottom) including the Predator test suite [18], the GNU Data Structure Library (GDSL), the BSD library, and a tutorial implementation of AVL trees (JSW) [37]. Some external libraries include typically simple, user-specified assertions (User Assert). A † indicates the routine is recursive, while a $*$ indicates the loops are nested. The subsequent columns provide metrics for the complexity of the code, including the total number of functions (Fun), the number of loops (Loop), the number of acyclic paths (Path), and the maximum number of simultaneous pointers into a data structure instance (Simult. Pointer).

RH1: Clumping is effective. Our analyzer attempts to infer complex disjunctive shape invariants in loops, which is particularly challenging considering the benchmarks shown in Table 1 with not only forward and back pointers but also possibly unbounded sharing patterns. Back pointers (as in doubly-linked lists or trees with parent pointers) require not only forward unfolding of inductive summaries but also backward unfolding [8]. The BSD red-black tree has parent pointers, which is heavily used in rebalancing. The leaves of the GDSL binary search tree all point back to the root node.

We observe a number of implementation idioms that drive a need for disjunctions. First, the maximum number of simultaneous pointers into a data structure instance (Simult. Pointer) is one driving factor as mentioned above, which typically increases as the operations get more complex (e.g., 12 for brbtree-delete). Second, the GDSL doubly-linked list uses sentinel nodes for the list head and tail. Because they are different than normal nodes, they cannot be summarized into the normal list segment, which yields more disjuncts to represent the possible points-to relationships between



(a) Number of benchmarks verified using each strategy.

Benchmark	ClumpG	Clump	CanonG	Canon	None
sll-delmin	0.04	$\top$	0.05	$\top$	$\top$
sll-delmin†	0.04	0.05	$\top$	$\top$	$\top$
sll-delminmax	0.12	$\top$	0.42	$\top$	$\top$
sll-delminmax†	0.20	0.20	$\top$	$\top$	$\top$
bstree-find	0.03	$\top$	0.05	$\top$	$\top$
bstree-find†	0.04	0.04	0.11	0.11	$\top$
psll-reverse	0.02	0.02	0.02	0.02	0.02
psll-isort	0.03	0.03	0.04	0.04	$\top$
psll-bsort	0.04	0.04	0.04	0.04	0.06
gdll-findmin	0.61	0.61	0.62	$\top$	0.61
gdll-index	0.61	$\top$	0.62	$\top$	0.61
gdll-findmax	0.61	0.61	0.62	$\top$	0.60
gdll-find	0.62	0.62	0.63	0.65	$\top$
gdll-delete	0.62	0.63	0.63	0.64	$\top$
gbstree-find	0.59	$\top$	0.59	$\top$	0.58
gbstree-insert	0.65	$\top$	$\top$	$\top$	$\top$
gbstree-delete	1.64	$\top$	1.71	$\top$	1.39
bsplay-find	0.28	$\top$	0.56	0.56	$\top$
bsplay-delete	0.48	$\top$	1.08	1.07	$\top$
bsplay-insert	0.30	$\top$	0.62	0.62	$\top$
brbtree-find	0.36	$\top$	0.36	$\top$	$\top$
brbtree-insert	1.07	$\top$	$\top$	$\top$	$\top$
brbtree-delete	6.06	$\top$	$\top$	$\top$	$\top$
javl-find	0.19	$\top$	0.19	$\top$	0.19
javl-free	0.18	0.19	0.19	0.18	0.19
javl-insert	1.84	$\top$	5.22	$\top$	$\top$
average (all)	0.66	0.28	0.68	0.39	0.47
verified	26	11	21	10	9

(b) Analysis times are only reported if all assertions are successfully verified, including the preservation invariant in post-conditions. $\top$ indicates failure to verify (due to precision loss). Run times (in seconds) measured on one core of a 3.20GHz Intel Xeon with 16GB of RAM. Times are reported as an average of three runs.

Figure 11. Successful verification times with different analysis strategies. The clumping with guided join strategy can verify significantly more benchmarks than any other strategy and over 3x more than the None baseline—and all at similar cost in analysis time.

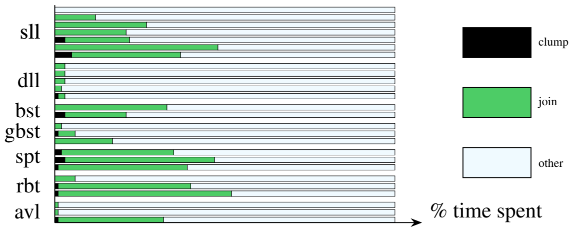
those nodes and the internal list segment. Third, nodes sometimes need to remain materialized between loops. Disjunctions are needed to represent the cases where the materialized node can occur but the number of disjuncts explodes exponentially in the number of materialized nodes in a tree. For example, delete in a red-black tree (brbtree-delete) requires three loops in sequence: find the node n to delete, find the minimum node m in the right subtree of n (i.e., the next in-order node to preserve the binary search invariant), rebalance the tree from the right subtree of that minimum node m. The node to delete n must be kept materialized between the first and second loop (to be able to track the swap of m and n) but becomes irrelevant between second and third loop (after the swap). While maintaining a disjunct, or trace partition, for every acyclic path may be sufficiently precise in many or even most cases, it is clear from the path counts (column Path) that this choice is utterly infeasible in practice. The essence of semantic-directed clumping is to identify this sort of relevance or irrelevance of a materialized node by computing silhouettes (i.e., "abstracting the abstraction"). In Fig. 11(a), we compare clumping with guided join with the other strategies showing the number of benchmarks that can be successfully verified memory safe (i.e., free of null or dangling pointer dereference), the user-specified asserts, and the shape preservation post-condition using the different strategies (also see the bottom line of Fig. 11(b)). The bottom line is that clumping with guided join (ClumpG) is significantly more effective (able to verify benchmarks) than the baseline and also more capable than any other strategy. CanonG fails to verify, for example, the particularly complex red-black tree insertion and deletion operations.

Clumping with guided join is able to verify these additional benchmarks with comparable analysis times as any other configuration. The analysis time is larger for more complex benchmarks (e.g., for red-black tree delete brbtree-delete), which as expected raises its average analysis time, but ClumpG is also the *only* strategy that succeeds on this code. ClumpG is also notably faster than CanonG on javl-insert. Fig. 11(b) shows that ClumpG takes comparable or even less time than other strategies on benchmarks where other strategies succeed (e.g., in the bsplay benchmarks).

Another measure of effectiveness of ClumpG is that analysis logs show this strategy led to no precision loss in joins.

RH2: Guided join is necessary. Fig. 11(a) and 11(b) also show that not only is clumping effective but guided join is also necessary. While Clump does improve on the baseline (11 versus 9), the guided join strategies (ClumpG and CanonG) are significantly better (26 and 21, respectively). Clump and Canon tend to fail more in the tree benchmarks, as there is a larger search space for the join; guiding appears more critical when the search space for join is larger.

RH3: Clumping has low overhead. From the discussion above, we see that ClumpG is comparable in analysis times with any other configuration providing some evidence that clumping has a reasonable overhead. To see this more directly, we show in the graph below, the percentage of the analysis time spent on clumping, join, and other for each benchmark. We find that in all cases the time spent on clumping (which includes the time to compute silhouettes) is a very small percentage of the total analysis time (no more than a few percent), and much smaller than the time spent in join.



RH4: Clumping limits disjunctive explosion. Fig. 12(a) drills further down on the verification times from Fig. 11(b) by considering the maximum number of disjuncts produced at a loop head, any program point, and at the exit point of the operation. First, we observe that interestingly, the None baseline configuration only succeeds when the number of disjuncts at loop head is 1. When more than one disjunct is needed for the loop invariant, the silhouette abstraction seems crucial to guiding the inference of a sufficiently precise invariant. Second, in all cases where both ClumpG and CanonG succeed, ClumpG uses fewer disjuncts in the loop invariant, and the difference can be quite significant (e.g., for javl-insert, 3 versus 18). And interestingly, ClumpG is always able to get to a single disjunct at the exit point while proving the post-condition.

	`ClumpG`			`CanonG`			`None`		
Benchmark	Fix	Max	Post	Fix	Max	Post	Fix	Max	Post
psll-reverse	1	1	1	1	1	1	1	1	1
psll-isort	1	2	1	1	3	2	1	⊤	⊤
psll-bsort	1	5	1	1	4	1	1	8	1
gdll-findmin	2	3	1	4	7	2	1	3	2
gdll-index	1	7	1	1	5	2	1	5	5
gdll-findmax	2	3	1	4	7	2	1	3	2
gdll-find	2	6	1	2	6	2	⊤	⊤	⊤
gdll-delete	2	6	1	2	6	2	⊤	⊤	⊤
gbstree-find	1	3	1	1	3	3	1	3	3
gbstree-insert	2	4	1	⊤	⊤	⊤	⊤	⊤	⊤
gbstree-delete	1	69	1	2	68	1	1	54	54
bsplay-find	3	42	1	5	89	1	⊤	⊤	⊤
bsplay-delete	3	42	1	5	89	1	⊤	⊤	⊤
bsplay-insert	3	42	1	5	89	1	⊤	⊤	⊤
brbtree-find	1	13	1	2	8	2	⊤	⊤	⊤
brbtree-insert	3	51	1	⊤	⊤	⊤	⊤	⊤	⊤
brbtree-delete	3	108	1	⊤	⊤	⊤	⊤	⊤	⊤
javl-find	1	3	1	1	3	1	1	3	3
javl-free	1	2	1	1	2	1	1	⊤	⊤
javl-insert	3	120	1	18	240	1	⊤	⊤	⊤
max	3	120	1	18	240	3	1	54	54

(a) The maximum number of disjuncts at a loop head (Fix), at any program point (Max), and at the exit point (Post) produced by the `ClumpG`, `CanonG`, and `None` analysis configurations.

	`ClumpG`		`CanonG`		`None`	
Benchmark	Time	Post	Time	Post	Time	Post
gbstree-find	3.3	1	2.18	1	40.79	158
brbtree-insert	60.2	1	⊤	⊤	⊤	⊤
javl-find	0.63	1	⊤	⊤	3.77	158
javl-insert	129.9	1	526.17	1	⊤	⊤
average (all)	48.5075	1	264.175	1	22.28	158

(b) Synthesize a new benchmark by sequentially composing a data structure operation 32 times to simulate multiple operations in sequence (times in seconds).

Figure 12. Clumping limits disjunctive explosion.

From these observations, we hypothesize that being able to clump into the minimal number of necessary disjuncts is crucial to analyzing data structure operations in a client program that makes several such calls. To test this hypothesis, we perform a controlled experiment by sequentially composing a given operation with itself 32 times and trying to prove the shape preservation invariant at the end. The resulting analysis is notably more complex than for the initial code, due to growingly complex abstract states over call sequences. Results in Fig. 12(b) show that `ClumpG` keeps the number of disjuncts constant and scales well, whereas `CanonG` and `None` suffer a significant slowdown on the cases where they do not fail.

7. Related Works

Several generic techniques are used to handle disjunctions in static analyses. Disjunctive completion [11] adds support for disjunctions to an abstract domain, but never collapses them, thus is too expensive in practice. Similarly, [19] introduces the least disjunctive basis of an abstract domain as a compact domain with the same disjunctive completion, though this construction does not minimize the size of the representation of the abstract elements. State partitioning [12, 23] attaches abstract states to specific sets of states, which effectively allows one to support disjunctive properties while providing a way to control disjunct numbers via the definition of partitions. Trace partitioning [22, 33] achieves a similar result using information about traces. However, these works provide frameworks, and do not address the problem of finding a criterion to clump disjuncts. Silhouettes provide such a criterion, based on the properties of join.

Shape analyses based on three-valued logic [35], and on separation logic [4, 5, 7, 9, 18] are known to require disjunctions. The same goes for array analyses [21]. Several strategies have been designed to limit the number of disjunctions. When summary predicates denote non-empty regions, possibly empty regions create an additional need for disjunctions. Thus, several proposals have been made to let summary predicates denote possibly empty regions [9, 13, 38]. Canonicalization operators [4, 17, 25, 35] collapse abstract states into a smaller, finite lattice, and hereby bound the number of disjuncts. While the analysis may use a larger lattice, precision after application of this operation is limited by that of the smaller lattice. Join operators [9, 38] do not require a smaller lattice (hence can keep more information), but lack a mechanism to bound the cardinality of disjunctions. To apply join or canonicalization operators efficiently, several strategies have been designed. Arnold [2] groups abstract states that satisfy some inclusion relation. Moreover, [38] performs a partial join that groups disjuncts only when it syntactically verifies the absence of precision loss. By contrast, our work proposes a criterion based on an abstraction of abstract states, and on the fact that this abstraction is cheaper to compute. A more related work is that of Manevich [27], that extends TVLA [25] with a grouping of three-valued abstract states based on a partial graph isomorphism.

Techniques to merge disjuncts have been developed in static analyses for numerical properties as well. For instance, a notion of affinity between polyhedra is used in [31] so as to decide whether they can be joined without too significant a precision loss. This approach is extended in [30] to deal with set properties. Bagnara [3] proposed a widening over disjunctions of polyhedra, that tests for implication to better bound the precision loss.

Existing off-line approaches for the parameterization of static analyses and abstract domains (such as the selection of the abstract domain to use, and of the abstract values to keep) include syntactic heuristics based on code patterns [6], machine learning techniques [26, 29], and semantic impact pre-analysis methods [28]. The disjunct clumping problem is tied to the abstract states that arise during the analysis, thus unsurprisingly calls for using an on-line abstraction of these at analysis time.

Finally, we remark that other analyses that abstract structures with summaries [14], heap abstractions [15], rich type systems [24, 34] or quantified logical assertions [20] may benefit from adapted forms of silhouette abstraction when facing the disjunction problem.

8. Conclusion

In this paper, we introduced *silhouettes* that abstract abstract states. The information enclosed in the silhouettes proves useful not only to clump disjunctions of abstract states, but also to compute better abstract joins. These results were achieved by selecting a definition of silhouettes that provides a weak entailment check over abstract states, and hereby characterizes accurately abstract states that are likely to join well. This characterization is conservative with respect to the standard analysis algorithms: while it will always suggest clumping abstract states that can be joined precisely, it may suggest clumping abstract states that cannot be joined precisely, though we never observed this behavior in the experiments. This situation is quite similar to that of a static analysis that can be proven sound but is conservative in theory, yet computes very precise results in practice. Our experimental evaluation confirms the effectiveness of the silhouette abstraction, which allows our implementation to verify a large collection of challenging benchmarks at a reasonable cost. The overhead inherent in computing silhouettes is outweighed by the benefit of increased precision and improved scalability.

Acknowledgments.

We are particularly grateful to Arlen Cox for discussions on the use of silhouettes in analysis and his comments on the paper. We also thank Hugo Illous, Yoonseok Ko, Jiangchao Liu, Kwangkeun Yi, and the anonymous referees for comments helping us improve this paper.

References

[1] A. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 2008.

[2] G. Arnold. Specialized 3-valued logic shape analysis using structure-based refinement and loose embedding. In *SAS*, pages 204–220. Springer, 2006.

[3] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. In *VMCAI*, pages 135–148. Springer, 2004.

[4] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68. Springer, 2005.

[5] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, pages 178–192. Springer, 2007.

[6] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.

[7] A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *VMCAI*, pages 1–22. Springer, 2012.

[8] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260. ACM, 2008.

[9] B.-Y. E. Chang, X. Rival, and G. Necula. Shape analysis with structural invariant checkers. In *SAS*, pages 384–401. Springer, 2007.

[10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[12] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

[13] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118, 2011.

[14] A. Cox, B.-Y. E. Chang, and X. Rival. Automatic analysis of open objects in dynamic language programs. In *SAS*, pages 134–150, 2014.

[15] I. Dillig, T. Dillig, and A. Aiken. Symbolic heap abstraction with demand-driven axiomatization of memory invariants. In *OOPSLA*, pages 397–410. ACM, 2010.

[16] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *POPL*, pages 187–200, 2011.

[17] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302. Springer, 2006.

[18] K. Dudka, P. Peringer, and T. Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *CAV*, pages 372–378. Springer, 2011.

[19] R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Science of Computer Programming*, 32(1):177–210, 1998.

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

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

[22] M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *SAS*, pages 200–214. Springer, 1998.

[23] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *SAS*, pages 39–50. Springer, 1999.

[24] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI*, pages 304–315. ACM, 2009.

[25] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, pages 280–301. Springer, 2000.

[26] P. Liang, O. Tripp, and M. Naik. Learning minimal abstractions. In *POPL*, pages 31–42. ACM, 2011.

[27] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *SAS*, pages 265–279. Springer, 2004.

[28] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi. Selective context-sensitivity guided by impact pre-analysis. In *PLDI*, pages 475–484. ACM, 2014.

[29] H. Oh, H. Yang, and K. Yi. Learning a strategy for adapting a program analysis via bayesian optimisation. In *OOPSLA*, pages 572–588. ACM, 2015.

[30] T. Pham, M. Trinh, A. Truong, and W. Chin. Fixbag: A fixpoint calculator for quantified bag constraints. In *CAV*, pages 656–662. Springer, 2011.

[31] C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In *ASIAN*, pages 331–345. Springer, 2006.

[32] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE, 2002.

[33] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *TOPLAS*, 29(5), 2007.

[34] P. M. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, pages 131–144. ACM, 2010.

[35] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.

[36] P. Sotin and X. Rival. Hierarchical shape abstraction of dynamic structures in static blocks. In *APLAS*, pages 131–147, 2012.

[37] J. Walker. AVL balanced tree library, 2003. `http://www.eternallyconfuzzled.com/libs/jsw_avltree.zip`.

[38] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398. Springer, 2008.