FESA: Fold- and Expand-Based Shape Analysis*

Holger Siegel and Axel Simon

Technische Universität München, Institut für Informatik II, Garching, Germany
firstname.lastname@in.tum.de

Abstract. A static shape analysis is presented that can prove the absence of `NULL`- and dangling pointer dereferences in standard algorithms on lists, trees and graphs. It is conceptually simpler than other analyses that use symbolically represented logic to describe the heap. Instead, it represents the heap as a single graph and a Boolean formula. The key idea is to summarize two nodes by calculating their common points-to information, which is done using the recently proposed *fold* and *expand* operations. The force of this approach is that both, *fold* and *expand*, retain relational information between points-to edges, thereby essentially inferring new shape invariants. We show that highly precise shape invariants can be inferred using off-the-shelf SAT-solvers. Cheaper approximations may augment standard points-to analysis used in compiler optimisations.

1 Introduction

Performing a shape analysis can help optimizing compilers to perform certain program specializations, such as inlining of virtual functions, partial evaluation (e.g. for fusing producers and consumers in functional languages [4]), or simply to eliminate `NULL` pointer tests.

In order to make shape analysis amenable for compilers, an analysis should be flexible in its precision to allow tuning its scalability. In this work we present a shape analysis whose performance depends on a numeric domain that can be implemented using off-the-shelf convex approximations [20], although we represent the state exactly using a SAT solver to illustrate its precision.

Our shape analysis can be seen as an instance of TVLA with one crucial simplification: The logic inherent in our analysis avoids a recursive transitive closure operator (RTC), as it is used in TVLA to encode reachability. For instance, a linked list with head H and a summarized tail T starting in a stack variable `x` is commonly encoded in TVLA as $x(H) = t_n(H,H) = t_n(H,T) = 1 \wedge t_n(T,T) = \frac{1}{2}$, meaning that `x` points to H, and from H we can reach H and T by following the n field zero or more times, from T we may reach T by following the n field zero or more times. The t_n predicate models transitive reachability and is defined using the RTC operator. Retaining this linked list invariant requires nontrivial integrity constraints on t_n, since the $\frac{1}{2}$ value of the $t_n(T,T)$ predicate proliferates during the evaluation of transfer functions. Rather than encoding reachability

* This work was supported by DFG Emmy Noether programme SI 1579/1.

K. De Bosschere and R. Jhala (Eds.): CC 2013, LNCS 7791, pp. 82–101, 2013.

a)
```
x = y = new_node();
while(rnd()) {
  *y = new_node();
  y = *y;
}
```

b)

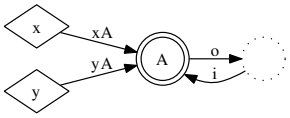


Fig. 1. allocating a linked list

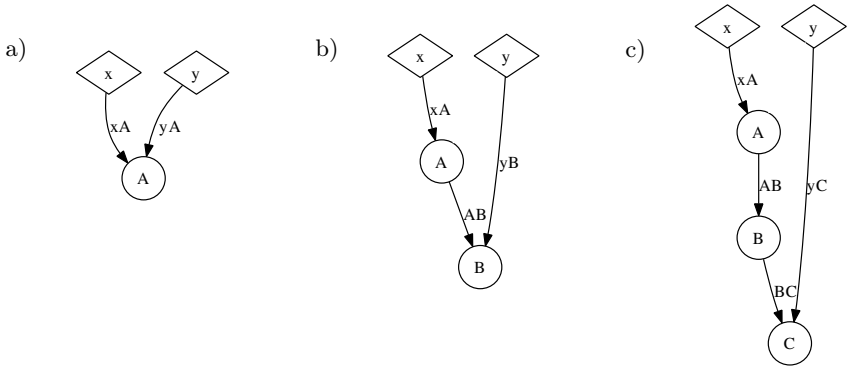


Fig. 2. calculating the state in the `while`-loop

directly, we only encode node-local information: (1) the existence of nodes and points-to edges, (2) the number of incoming nodes and (3) the number of outgoing edges. We show that these three properties suffice. In fact, we replace the set of graphs and their three-valued interpretation used in TVLA by a set of two-valued interpretations of a single graph. Thus, the omission of the RTC operator allows us to define a property-based shape analysis where a state is simply described by one graph and a single Boolean formula. The contribution of our work lies in describing how to perform summarization and materialization of nodes using sets of two-valued interpretations. Specifically, the insight in this paper is that invariant relations between neighboring nodes can be inferred during summarization using a relational *fold* operation that was recently defined for numeric domains [19]. A symmetric *expand* operation duplicates these relations when a summary cell is materialized back into two heap cells, one of which is a concrete non-summary cell. While the semantic information aggregated and duplicated by these two functions is nontrivial, their implementation is straightforward. This simplicity stands in contrast to previous work on using Boolean formulae as interpretation in the context of predicate abstraction [15] that requires sophisticated abstraction refinement techniques to obtain sufficient precision. Although our analysis does not explicitly maintain the linked list invariant, by virtue of the inferred relations between these predicates, it is precise enough to distinguish between lists, trees and graphs, which is not possible in TVLA without defining specific properties. We now demonstrate how inference of relational information replaces the validation of TVLA's linked list invariants.

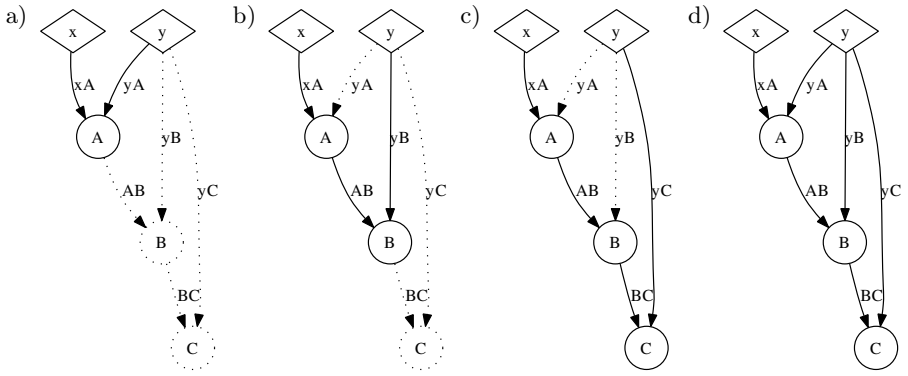


Fig. 3. Joining states

Consider the C program in Fig. 1a) that constructs a singly linked list. We will first show how the allocated cells can be summarized into a single summary node A, resulting in the heap in Fig. 1b). For the sake of exposition, we only show information pertaining to points-to edges.

Before the loop in Fig. 1a) is entered for the first time, `x` and `y` both point to a single node allocated by `new_node()`. The corresponding heap in Fig. 2a) depicts program variables as diamonds and heap-allocated cells as circles. Figure 2b) shows the state after one iteration. Here, `y` points to the newly allocated heap cell B. Figure 2c) shows the state after another iteration.

In order to represent the three states in a single abstract state, we represent heaps using a points-to map (a graph) and a numeric state. The points-to map takes each heap cell or program variable to a set of points-to edges and is shown as a directed graph. Each points-to edge is then further qualified by a flag that is mapped to zero or to one by the numeric domain. In particular, a flag f_{AB} maps to one iff the edge from node A to node B exists. For instance, the state before the loop consists of the points-to map shown in Fig. 2a) and the numeric state $\langle f_{xA}, f_{yA}\rangle \in \{\langle 1,1\rangle\}$. The state after one iteration consists of the points-to map shown in Fig. 2b) and the numeric state $\langle f_{xA}, f_{yB}, f_{AB}\rangle \in \{\langle 1,1,1\rangle\}$. After another iteration, the state consists of the points-to map in Fig. 2c) and the numeric state $\langle f_{xA}, f_{yC}, f_{AB}, f_{BC}\rangle \in \{\langle 1,1,1,1\rangle\}$.

These three states cannot be merged directly, since their sets of edges and nodes are different. We therefore make them *compatible* to each other by adding missing edges and nodes. Figure 3a) shows how adding nodes B and C and edges yB, yC, AB and BC to Fig. 2a) gives a compatible points-to graph with numeric state $\langle f_{xA}, f_{yA}, f_{yB}, f_{yC}, f_{AB}, f_{BC}\rangle \in \{\langle 1,1,0,0,0,0\rangle\}$. Figure 3b) and c) show how the states in Fig. 2b) and c) are made compatible with the corresponding numeric states $\{\langle 1,0,1,0,1,0\rangle\}$ and $\{\langle 1,0,0,1,1,1\rangle\}$, respectively. The merge of these compatible states is completed by joining the three numeric states into a single state $b := \{\langle 1,1,0,0,0,0\rangle\, \langle 1,0,1,0,1,0\rangle\, \langle 1,0,0,1,1,1\rangle\}$. The benefit of this representation is that the three heap configurations are all encoded by the graph in Fig. 3d) and the numeric state b.

a)
```
do {
    z = x;
    x = *x;
    free(z);
} while(x);
```

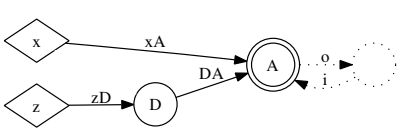
b)


Fig. 4. Deallocating a linked list

The latter graph and state b can now be transformed into a graph where all list nodes are summarized into one node as shown in Fig. 1b). To this end, our analysis uses the points-to graph to determine which edges to overlay and then applies a relational *fold* operation from [19] in order to summarize the corresponding dimensions. As a result, the dimensions $\langle f_{xA}, f_{yA}, f_{yB}, f_{yC}, f_{AB}, f_{BC}\rangle$ corresponding to Fig. 3d) are mapped to the dimensions $\langle f_{xA}, f_{yA}, f_i, f_o\rangle$ corresponding to Fig. 1b). This summary state represents the common points-to properties of all nodes, where the edges f_i and f_o represent the in- and outgoing edges connecting A with another instance of A. This instance is drawn with a dotted circle and is henceforth called the *ghost node* of A.

Interestingly, if we were to summarize a list with up to four nodes, we would obtain the same summarized state. Indeed, the summarized state is a fixpoint for the loop. This, in turn, implies that the summary represents a singly linked list of arbitrary size. The analysis has thus inferred that the loop constructs a singly linked list that commences in `x` and ends in the node pointed to by `y`. The numeric state of the fixpoint is quite subtle and describes the various rôles the summary node can take on: the list head A in Fig. 3a); the list head A in b), c); the unreachable nodes B, C in a) and C in b); the final node B in b) and C in c); and the inner node B in c). The key observation is that the relational *fold* is able to automatically synthesize these rôles and that the Boolean function maintains the distinction between them, thereby inferring very strong shape invariants. Note, though, that the invariant does not state which roles of A are possible. Thus, it might be that A only occurs in its role as a middle element of a list, thereby forming a cyclic list in the heap. Only the fact that `x` points to A and the relational information on A forces A to take on the role as list head and as list tail, thereby stating that it is an acyclic list. The inference of a precise relational summary using *fold* therefore replaces the verification of a linked-list invariant done by TVLA.

We now consider the loop in Fig. 4a) that deallocates a linked list. In our analysis, a summary node is materialized each time it is accessed. Thus, all modifications to heap nodes are performed on materialized nodes, which ensures that the abstract transformers are easy to derive since they closely follow the concrete transformers. Consider the assignment `x = *x` in the third line. Since the dereference `*x` would access a summary node, our analysis materializes a node D from the summary before executing the assignment. This materialization results in a state with the points-to set depicted in Fig. 4b).

In this particular case it is possible to free the node D by simply removing it from the graph. After `z` goes out of scope, the resulting graph is already compatible with that at the loop head. A fixpoint for the numeric domain is observed after one further iteration.

When evaluating the expression `*x`, the analysis checks that the numeric domain maps at least one edge in the points-to set of `x` to one. If this is not the case, a warning is emitted, stating that `x` can be `NULL`. In the example, our analysis is precise enough to verify the absence of `NULL`-pointer dereferences. Indeed, it can infer invariants that distinguish lists from trees from graphs. In contrast to other analyses [7,12], no extra effort is needed to make our analysis robust with respect to variations of these basic data structures (position of pointer fields, use of sentinel nodes instead of `NULL` values or the use of back pointers).

Overall, our shape analysis lies at a sweet-point between precision and simplicity by building on the following contributions presented in this work:

- a shape representation using points-to relations qualified by {0, 1}-vectors, thus substituting the common approach of representing heaps using a logic with simple transformers operating on a single graph and a numeric state
- a shape analysis for which transformers are easy to derive since they never operate on summary nodes and thus directly follow the concrete transformers
- the use of relational *fold* and *expand* operators [19] to summarize and materialize heap cells, allowing for a highly precise *inference* of new shapes; indeed, we can synthesize the strongest invariant when summarizing two heap cells
- the observation that only two extra properties suffice to distinguish common classes of data structures [21] as long as these are tracked with high precision; allowing us to verify programs operating on lists, trees and graphs

We present the principles of our shape analysis before Sect. 3 formalizes it for a *C*-like language. An analysis of trees is presented in Sect. 4 before Sect. 5 details our implementation. Section 6 discusses related work before Sect. 7 concludes.

2 Shape Analysis Using Numeric Domains

A shape analysis finitely summarizes a set of potentially unbounded, *concrete shape graphs* into an abstract representation. Before we define the abstract domain of points-to sets and numeric states, we consider concrete heap shapes.

Let $\mathcal{A}$ be a set of symbolic addresses. With each memory cell M we associate a unique symbolic address $A_M \in \mathcal{A}$. A *concrete shape graph* is a partial map $c : \mathcal{A} \to \wp(\mathcal{A})$ that maps each allocated memory cell to its *points-to set* $c(A_M)$: When $c(A_M) = \{A_N\}$ and $A_N \in \mathrm{dom}(c)$ then memory cell M contains a pointer to memory cell N. When $c(A_M) = \emptyset$ then the memory cell M contains `NULL`. A points-to graph may also contain cells whose content does not represent a proper pointer value: When $|c(A_M)| > 1$ or $c(A_M) = \{A_N\}$ with $N \notin \mathrm{dom}(c)$ then M contains an *invalid pointer*, namely, one that should not be dereferenced.

The remainder of the section presents the abstract states that represent sets of concrete shape graphs; it discusses summarization and materialization of nodes and addresses the correctness of these operations.

2.1 Abstract Shape Graphs

The building block of our shape analysis is a points-to set that is further qualified by a set of {0, 1}-vectors. In particular, we define a flow-sensitive points-to

analysis [11] by associating each memory cell M with a points-to set of the form $\{\langle f_1, A_1\rangle, \ldots \langle f_k, A_k\rangle\} \subseteq \mathcal{X} \times \mathcal{A}$ where $\mathcal{X}$ is a set of flags. A numeric domain associates each flag with a value drawn from $\{0, 1\}$. The idea is that $A_i \in c(A_M)$ if the flag f_i is mapped to one [20]. Thus, a memory cell can be dereferenced if exactly one flag in its points-to set is mapped to one by the numeric domain. In order to define this combined domain, we first present the numeric domain.

The Numeric Domain. The possible configurations of flags are given by a numeric domain $\mathcal{B}_n := \wp(\{0, 1\}^n)$ that holds sets of $\{0, 1\}$-vectors of dimension n. As a numeric domain, $\mathcal{B}_n$ can be modified by removing, adding and swapping dimensions or by restricting their values. The removal of dimension i from $b \in \mathcal{B}_n$ is defined by $\mathit{drop}_i(b) = \{\langle v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n\rangle \mid \langle v_1, \ldots v_n\rangle \in b\}$. We write $\mathit{drop}_{i_1 \ldots i_k} = \mathit{drop}_{i_1} \circ \cdots \circ \mathit{drop}_{i_k}$ for ascending sequences $i_1 \ldots i_k$ of indices. A dimension is added to $b \in \mathcal{B}_n$ using $\mathit{add}_i(b) = \{\langle v_1, \ldots, v_{i-1}, v, v_i, \ldots, v_n\rangle \mid \langle v_1, \ldots, v_n\rangle \in b,\ v \in \{0, 1\}\}$. We write $\mathit{add}_{i_1 \ldots i_k} = \mathit{add}_{i_k} \circ \cdots \circ \mathit{add}_{i_1}$ for ascending sequences $i_1 \ldots i_k$ of indices. A swap of two dimensions i and j in $b \in \mathcal{B}_n$ is denoted as $\mathit{swap}_{i,j}(b)$. It lifts naturally to two sequences of equal length.

For presentational purposes, we use flag names as synonyms of vector indices and assume sequences of dimensions to be in ascending order wherever they occur. Moreover, we write $\mathit{add}_f(b)$ to associate a new dimension with the flag name f. We omit the number of dimensions of the numeric domain, as it is equal to the number of stored flags. Using this convention, restricting a numeric state $b \in \mathcal{B}$ is denoted by $b[\![\mathit{expr}]\!] = \{\boldsymbol{b} \in b \mid \mathit{expr}\}$. For instance, $b[\![f = 0]\!]$ denotes all vectors $\boldsymbol{b} \in b$ in which the dimension associated with flag f maps to zero, and $\mathit{add}_{f_2}(b)[\![f_2 = 1 - f_1]\!]$ denotes an assignment of $1 - f_1$ to a fresh dimension f_2.

The Points-to Domain. A points-to state $p : \mathcal{A} \to \wp(\mathcal{X} \times \mathcal{A})$ maps (the address of) each memory cell to its points-to set. The set of points-to states is denoted by P. Writing a points-to set v to a memory cell at address A in state $p \in P$ is denoted by the update $p[A \mapsto v]$. We use a combined abstract state $p \rhd b \in P^{\mathcal{B}}$ where $P^{\mathcal{B}}$ is a new abstract points-to domain in which p is refined (qualified) by a numeric domain $b \in \mathcal{B}$. An operation on a state $p \rhd b$ adjusts p, thereby modifying the set of flags in the points-to sets, which, in turn, requires adjustments to $b \in \mathcal{B}$.

An abstract heap description must be able to represent concrete heaps c_1, c_2 with different numbers of cells, that is, $\mathrm{dom}(c_1) \neq \mathrm{dom}(c_2)$. For example, a heap cell may be deallocated in one state but not in another. In order to ensure that accessing a non-allocated region can be flagged as "dangling pointer" error, the numeric domain tracks an additional flag $f_{\exists M}$ for each heap-allocated cell M that is one iff M is allocated, that is, if $A_M \in \mathrm{dom}(c)$. For brevity, we generally omit the $f_{\exists X}$ flags in the presentation of the upcoming examples whenever they are constant one in the numeric state $b \in \mathcal{B}$, but we consider this example first:

Example 1. The points-to domain of the linked list in Fig. 5a) can be given by $p = [A_x \mapsto \{\langle f_{xA}, A_A\rangle\}, A_A \mapsto \{\langle f_{AB}, A_B\rangle\}, A_B \mapsto \{\langle f_{BC}, A_C\rangle\}]$ where A_x, A_A, A_B and A_C are the addresses of memory cells `x`, A, B and C, respectively. The numeric domain $\langle f_{xA}, f_{AB}, f_{BC}, f_{\exists A}, f_{\exists B}, f_{\exists C}\rangle \in \{\langle 1, 1, 1, 1, 1, 1\rangle\}$

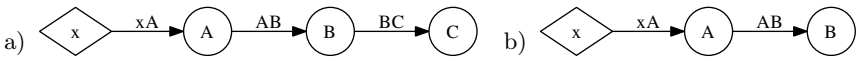


Fig. 5. Two linked lists

assures that dereferencing the contents of x, A and B is safe since in each points-to set exactly one flag has value one and all heap cells are allocated.

Lattice Operations on Abstract States. Whenever the control flow merges with states $p_1 \rhd b_1$ and $p_2 \rhd b_2$, the analysis continues with the joined state $p_1 \rhd b_1 \sqcup p_2 \rhd b_2$, which is $p_1 \rhd b_1 \cup b_2$ if $p_1 = p_2$. Checking for fixpoints when analyzing loops requires a subset test $p_1 \rhd b_1 \sqsubseteq p_2 \rhd b_2$ that reduces to $b_1 \subseteq b_2$ if $p_1 = p_2$. An analogous definition for $\sqcap$ yields the complete lattice $\langle P^{\mathcal{B}}, \sqsubseteq, \sqcup, \sqcap \rangle$.

In general, the points-to domains p_1 and p_2 may be different: Before they can be joined or compared, they must be made *compatible* by adding edges and nodes to p_1, p_2: When p_i contains a node M that is not present in p_j, the missing node is added to p_j using $p_j[A_M \mapsto \emptyset]$ and $f_{\exists M} = 0$ is added to b_j, indicating that M is not allocated. When p_i contains an edge from some node M to some node N that is not present in p_j, the edge is added to the points-to set of M in p_j and the corresponding numeric flag f_{MN} is introduced in b_j with value zero. Further adjustments are necessary for summary nodes and instrumentation predicates which are defined analogously. Consider again the lists in Fig. 5:

Example 2. Joining the heaps of Fig. 5a) and Fig. 5b) yields the points-to state of Fig. 5a) with $\langle f_{xA}, f_{AB}, f_{BC}, f_{\exists A}, f_{\exists B}, f_{\exists C} \rangle \in \{\langle 1, 1, u, 1, 1, u \rangle \mid u \in \{0, 1\}\}$.

2.2 Summarizing Nodes

This section illustrates how the previously defined operations are used to summarize two heap cells, which is a three-step process: First, both nodes are made *compatible*, then the edges between them are removed, and finally the numeric flags of one cell have to be merged with those of the other cell. Materialization applies the last two steps in reverse order. We consider each step in turn.

Making Nodes Compatible. Suppose that we summarize nodes A and B of the three-element linked list in Fig. 5. In order to merge the numeric information stored for the points-to information of A and B, both must have the same set of incoming and outgoing edges. To this end, they are made *compatible* by complementing the edge from x to A with a new edge from x to B and adding the flag $f_{xB} = 0$. Similarly, the edges from B to C and from A to B are complemented with edges from A to C and from C to B, yielding the state in Fig. 6a).

Disconnecting the Nodes. The points-to information between A and B can no longer be represented as edges between different nodes of the graph once they are summarized. We therefore introduce a *ghost node* to which these edges point, indicating that they point to a different instance of the summarized node. To

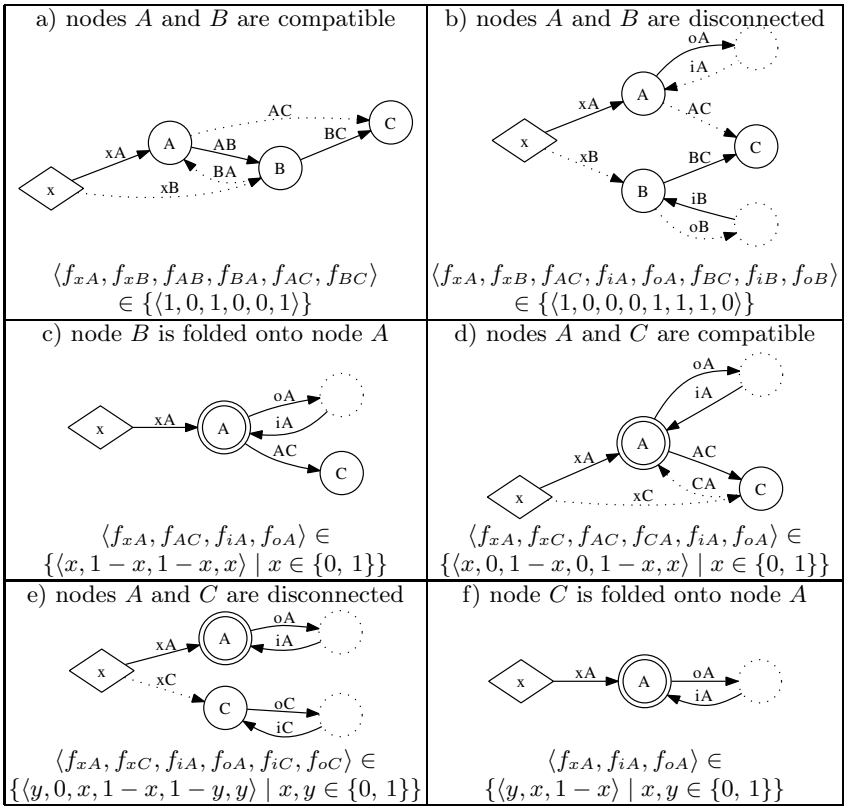


Fig. 6. Folding and expanding a linked list

this end, we assign the value of f_{AB} to the flag f_{oA} that represents the out-edge from A to its ghost node and to flag f_{iB} that represents the in-edge from its ghost node to B. In the same way, we assign the value of f_{BA} to f_{iA} and to f_{oB}. Finally the edges between A and B are discarded, giving the state in Fig. 6b).

Summarizing Numeric Information. Note that the points-to sets of A and B now contain the same set of addresses, so that each flag associated with B can be merged with the respective flag of A. Merging and duplication of flags is based on *fold* and *expand* which is defined as follows [19]:

Definition 1. *Given the k dimensions $i_1 \dots i_k$ and k dimensions $j_1 \dots j_k$, define $fold_{i_1 \dots i_k, j_1 \dots j_k} : \mathcal{B}_{n+k} \to \mathcal{B}_n$ and $expand : \mathcal{B}_n \to \mathcal{B}_{n+k}$ as follows:*

$$fold_{i_1 \dots i_k, j_1 \dots j_k}(b) = drop_{j_1 \dots j_k}(b \cup swap_{i_1 \dots i_k, j_1 \dots j_k}(b))$$
$$expand_{i_1 \dots i_k, j_1 \dots j_k}(b) = add_{j_1 \dots j_k}(b) \cap swap_{i_1 \dots i_k, j_1 \dots j_k}(add_{j_1 \dots j_k}(b))$$

Intuitively, the *fold* operation merges the information over $j_1 \dots j_k$ with that over $i_1 \dots i_k$ and removes $j_1 \dots j_k$. This process discards all information between

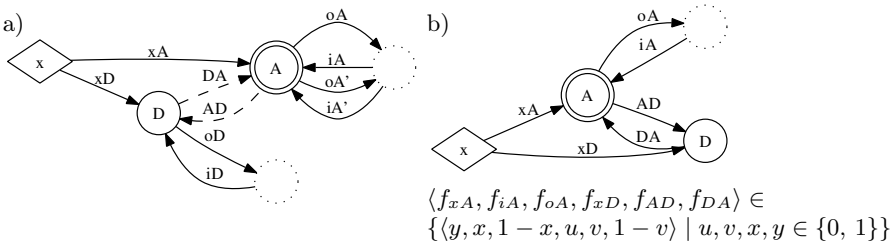


Fig. 7. Materializing D from A, given the state of Fig. 6f)

$i_1 \dots i_k$ and $j_1 \dots j_k$ but retains any relational information that holds for both $i_1 \dots i_k$ and $j_1 \dots j_k$. Symmetrically, *expand* retains relations within $i_1 \dots i_k$ when duplicating them to $j_1 \dots j_k$ but, unlike an assignment $j_1 := i_1$, induces no equality between j_1 and i_1. In the example, the flags $f_{xB}, f_{BC}, f_{iB}, f_{oB}$ are folded onto the flags $f_{xA}, f_{AC}, f_{iA}, f_{oA}$ by applying $fold_{f_{xA}f_{AC}f_{iA}f_{oA}, f_{xB}f_{BC}f_{iB}f_{oB}}$ to the numeric state. Note that this operation retains the relations that exist in each group of flags as shown in Fig. 6c). Here, the summarized state retained that the node that points to C is not the one pointed to by `x` since $f_{xA} \neq f_{AC}$.

Summarizing Summaries. Now we merge the summary node A with node C which has to be made compatible first as shown in Fig. 6d). Since A already has a ghost node, the flags f_{iA}, f_{oA} are already present in the numeric domain. Therefore, the information of f_{iA}, f_{oA} has to be merged with that of f_{AC}, f_{CA}. The intuition is that after summarizing A and C, there is no distinction between the edges from A to C and edges from A to nodes that have been previously merged with A. On the numeric state, flags f_{iC}, f_{oC} are introduced with $f_{iC} = f_{AC}$, $f_{oC} = f_{CA}$ and flags f_{CA}, f_{AC} are folded onto f_{iA}, f_{oA} by operation $fold_{f_{iA}f_{oA}, f_{CA}f_{AC}}$, yielding the state in Fig. 6e). Now all flags of node C are folded onto those of A by operation $fold_{f_{xA}f_{iA}f_{oA}, f_{xC}f_{iC}f_{oC}}$, giving the state in Fig. 6f).

2.3 Materializing Nodes from Summaries

In this section we discuss how to materialize a node from a summary node in two steps: first, the summary node is duplicated by applying *expand* to its numeric flags; second, the edges between the summary and the new node are synthesized using the edges to the ghost node. Consider expanding D from node A in Fig. 6f).

The first step applies $expand_{f_{xA}f_{iA}f_{oA}, f_{xD}f_{iD}f_{oD}}$ to the numeric state, resulting in $\langle f_{xA}, f_{iA}, f_{oA}, f_{xD}, f_{iD}, f_{oD}\rangle \in \{\langle y, x, 1-x, u, v, 1-v\rangle \mid u, v, x, y \in \{0, 1\}\}$. After that, it remains to reconstruct the edges between A and D: since node D is concrete and node A is a summary, we may assume that the flags f_{DA}, f_{AD} are equal to flags f_{oD}, f_{iD} and also equal to flags that have been summarized with f_{iA}, f_{oA} in the process of summarization. Thus, we first add the edges between A and D and their corresponding flags f_{AD}, f_{DA} by renaming flags f_{iD}, f_{oD} in Fig. 7a). In order to assert the equality with an *instance* of the edges f_{iA}, f_{oA}, we duplicate them to f'_{iA}, f'_{oA} using $expand_{f_{iA}f_{oA}, f'_{iA}f'_{oA}}$ and enforcing the equality

by restricting the numeric state b to $b[\![f_{AD} = f'_{oA} \wedge f_{DA} = f'_{iA}]\!]$. The flags f'_{iA}, f'_{oA} are removed using *drop*, resulting in the state shown in Fig. 7b).

In contrast to the initial state in Fig. 6a), in the materialized state variable `x` may also contain `NULL` (since $f_{xA} = f_{xD} = 0$ is possible) or an invalid pointer (since $f_{xA} = f_{xD} = 1$ is possible). This precision loss is caused by summarizing the flags f_{xB} and f_{xC} onto f_{xA}, which makes them indistinguishable. Sect. 4.1 details how the validity of pointers can be maintained despite summarization. We conclude this section with an observation on the soundness of summarization.

2.4 Soundness of Summarization

We assume the existence of a function *summarize* that summarizes two nodes and a function *materialize* that expands summary nodes as described above and leaves non-summary nodes unchanged:

Definition 2. *Define* $summarize : \mathcal{A} \times \mathcal{A} \times P^{\mathcal{B}} \to P^{\mathcal{B}}$ *such that in state* $summarize(A_M, A_N, p \rhd q)$ *node* M *is a summary of* M *and* N *of state* $p \rhd q$.

Define $materialize : \mathcal{A} \times P^{\mathcal{B}} \to P^{\mathcal{B}}$ *such that for* $p' \rhd b' = materialize(A_N, p \rhd b)$, *some new address* $A_C \in \mathrm{dom}(p') \setminus \mathrm{dom}(p)$ *points to the materialized concrete node in state* $p' \rhd b'$ *if* N *is a summary in* $p \rhd b$, *and* $p' \rhd b' = p \rhd b$ *otherwise.*

Note that we assume that *materialize* recognizes a node M as summary node by observing the existence of the flags f_{iM}, f_{oM} in the numeric domain. As shown in [19], the pair $(fold_{i_1 \ldots i_k, j_1 \ldots j_k}, expand_{i_1 \ldots i_k, j_1 \ldots j_k})$ forms a Galois connection, from which follows that folding dimensions $j_1 \ldots j_k$ onto dimensions $i_1 \ldots i_k$ of a numeric state and then re-expanding dimensions $j_1 \ldots j_k$ yields an over-approximation of the initial numeric state. The following observations states that this extends to summarizing and then re-expanding a pair of nodes in the abstract shape graph.

Observation 1. *For all states* $p \rhd b \in P^{\mathcal{B}}$ *and addresses* $A_1, A_2 \in \mathrm{dom}(p)$ *there is* $p \rhd b \sqsubseteq materialize(A_1, summarize(A_1, A_2, p \rhd b))$ *up to the renaming of the materialized node.*

We will allow our analysis to summarize nodes in one sequence and afterwards materialize them in a different sequence. To ensure soundness, the result of summarizing and expanding a set of nodes must therefore be independent of the chosen sequence. It suffices that changing the order in which two nodes are materialized does not change the result, which is asserted as follows:

Observation 2. *For all states* $p \rhd b \in P^{\mathcal{B}}$ *and addresses* $A_1, A_2 \in \mathrm{dom}(p)$, $materialize(A_1, materialize(A_2, p \rhd b)) = materialize(A_2, materialize(A_1, p \rhd b))$ *holds up to the renaming of the materialized nodes.*

As a result, a shape analysis that relies on the operations *summarize* and *materialize* may summarize heap cells at any point and re-expand them on demand, namely when accessing the content of a summarized heap cell. The next section puts this into practice by defining an abstract interpreter for a C-like language which is later enriched to work on summaries.

$$
\begin{aligned}
[\![\texttt{x=NULL}]\!]^\natural c &= c[A_x \mapsto \emptyset] \\
[\![\texttt{x=y}]\!]^\natural c &= c[A_x \mapsto c(A_y)] \\
[\![\texttt{x=\&y}]\!]^\natural c &= c[A_x \mapsto \{A_y\}] \\
[\![\texttt{x=malloc()}]\!]^\natural c &= c[A_x \mapsto \{A_N\}, A_N \mapsto \emptyset] \text{ where } A_N \text{ fresh} \\
[\![\texttt{x=*y}]\!]^\natural c &= c[A_x \mapsto c(A)] \text{ where } \{A\} = c(A_y) \\
[\![\texttt{*x=y}]\!]^\natural c &= c[A \mapsto c(A_y)] \text{ where } \{A\} = c(A_x) \\
[\![\texttt{free(x)}]\!]^\natural c &= c \setminus A \text{ where } \{A\} = c(A_x) \\
[\![\texttt{if (x==NULL) t else e}]\!]^\natural c &= \begin{cases} [\![\texttt{t}]\!]^\natural c \text{ if } c(A_x) = \emptyset \\ [\![\texttt{e}]\!]^\natural c \text{ otherwise} \end{cases}
\end{aligned}
$$

Fig. 8. Concrete semantics of a C-like language

3 Shape Analysis of a C-Like Language

We present a language with explicit memory management using `malloc` and `free` which mimic their C counterparts. In contrast to C, we assume that program variables and heap cells are initialized to `NULL` and that they hold only one value; a simplification for the sake of clarity which is not present in our implementation.

The semantics of our language in Fig. 8 operates on concrete heap shapes as defined in Sect. 2. In a heap c, a cell M is allocated iff $A_M \in \mathrm{dom}(c)$. Thus $[\![\texttt{x=malloc()}]\!]^\natural$ adds a mapping for a new address A_N to c and $[\![\texttt{free(x)}]\!]^\natural$ removes the mapping for A, denoted by $c \setminus A$. Analogously, stack variables `x`,`y`,... are stored at $A_x, A_y, \ldots$ where $A_x \in \mathrm{dom}(c)$ iff `x` is in scope. Note that a state c containing an invalid pointer in cell M (with $|c(A_M)| > 1$ or $c(A_M) = \{A_N\}$ with $A_N \notin \mathrm{dom}(c)$) is not an error, only dereferencing M is. This is addressed by the rules $[\![\texttt{x=*y}]\!]^\natural$, $[\![\texttt{*x=y}]\!]^\natural$, and $[\![\texttt{free(x)}]\!]^\natural$ that are undefined if the dereferenced cell does not hold exactly one pointer or if the pointed-to cell at A is not allocated, that is, if $A \notin \mathrm{dom}(c)$. The goal of the analysis is to prove the absence of undefined behavior. Note that invalid pointers cannot arise in the concrete semantics, but may arise due to approximations in the abstract semantics, which is presented next.

3.1 Abstract Semantics

An abstract interpretation is sound if the abstract semantics $[\![\mathtt{s}]\!]^\sharp$ of each statement `s` approximates its concrete semantics $[\![\mathtt{s}]\!]^\natural$, that is, if $\overline{[\![\mathtt{s}]\!]^\natural} \circ \gamma \subseteq \gamma \circ [\![\mathtt{s}]\!]^\sharp$ with $\overline{[\![\mathtt{s}]\!]^\natural}(C) := \{[\![\mathtt{s}]\!]^\natural(c) \mid c \in C\}$ [9]. In order to derive this semantics, we first define a concretization function γ_0 that relates each abstract cell M to one concrete memory cell if A_M exists (that is, if its $f_{\exists M}$ flag is one):

Definition 3. *Function* $\gamma_0 : P^{\mathcal{B}} \to \wp(\mathcal{A} \to \wp(\mathcal{A}))$ *is given by*

$$\gamma_0(p \rhd b) = \left\{ [A_M \mapsto \{A_i \mid \langle f_i, A_i \rangle \in p(A_M) \wedge \boldsymbol{b}(f_i) = 1\}]_{A_M \in \mathrm{dom}(p) \wedge \boldsymbol{b}(f_{\exists M}) = 1} \mid \boldsymbol{b} \in b \right\}$$

where $\boldsymbol{b}(f_i)$ *denotes dimension* f_i *of a vector* $\boldsymbol{b}$.

$$
\begin{array}{ll}
[\![\texttt{x=NULL}]\!]^\sharp(p \rhd b) & = p[A_x \mapsto \emptyset] \rhd \mathit{drop}_{\{f \mid \langle f,\_\rangle \in p(x)\}}(b) \\
[\![\texttt{x=y}]\!]^\sharp(p \rhd b) & = p[A_x \mapsto \{\langle f_{x1}, A_1\rangle, \ldots, \langle f_{xn}, A_n\rangle\}] \rhd \\
 & \quad \mathit{add}_{f_{x1}\ldots f_{xn}}(\mathit{drop}_{\{f \mid \langle f,\_\rangle \in p(x)\}}(b))[\![f_{x1} = f_{y1}, \ldots, f_{xn} = f_{yn}]\!] \\
 & \quad \text{where } \texttt{x} \neq \texttt{y} \text{ and } p(y) = \{\langle f_{y1}, A_1\rangle \ldots, \langle f_{yn}, A_n\rangle\} \\
[\![\texttt{x=\&y}]\!]^\sharp(p \rhd b) & = p[A_x \mapsto \{\langle f_{xy}, A_y\rangle\}] \rhd \mathit{add}_{f_{xy}}(\mathit{drop}_{\{f \mid \langle f,\_\rangle \in p(x)\}}(b))[\![f_{xy} = 1]\!] \\
[\![\texttt{x=malloc()}]\!]^\sharp(p \rhd b) & = [\![\texttt{x=\&N}]\!]^\sharp(p[A_N \mapsto \emptyset] \rhd \mathit{add}_{f_{\exists N}}(b)[\![f_{\exists N} = 1]\!]) \text{ where } A_N \text{ fresh} \\
[\![\texttt{*x=y}]\!]^\sharp(p \rhd b) & = \bigsqcup_{\langle A_z, p' \rhd b'\rangle \in \mathit{deref}(A_x, (p \rhd b))} [\![\texttt{z=y}]\!]^\sharp(p' \rhd b') \\
[\![\texttt{x=*y}]\!]^\sharp(p \rhd b) & = \bigsqcup_{\langle A_z, p' \rhd b'\rangle \in \mathit{deref}(A_y, (p \rhd b))} [\![\texttt{x=z}]\!]^\sharp(p' \rhd b') \\
[\![\texttt{free(x)}]\!]^\sharp(p \rhd b) & = \bigsqcup_{\langle A_z, p' \rhd b'\rangle \in \mathit{deref}(A_x, (p \rhd b))} p' \rhd b'[f_{\exists z} \mapsto 0] \\
[\![\texttt{if (x==NULL) t else e}]\!]^\sharp(p \rhd b) & = [\![\texttt{t}]\!]^\sharp(p \rhd b[\![f_{x1} = 0 \wedge \ldots \wedge f_{xn} = 0]\!]) \\
 & \quad \sqcup [\![\texttt{e}]\!]^\sharp(p \rhd b[\![f_{x1} = 1 \vee \ldots \vee f_{xn} = 1]\!]) \\
 & \quad \text{where } p(A_x) = \{\langle f_{x1}, A_1\rangle, \ldots \langle f_{xk}, A_k\rangle\}
\end{array}
$$

Fig. 9. Abstract semantics

The abstract semantics $[\![\texttt{s}]\!]^\sharp$ that lifts the concrete semantics $[\![\texttt{s}]\!]^\natural$ of a statement `s` to the abstract domain $P^{\mathcal{B}}$ is shown in Figure 9. Here, $[\![\texttt{x=NULL}]\!]^\sharp$ removes all previous flags f of the points-to set of `x` from b using *drop* and empties $p(A_x)$. These stale flags are also removed in $[\![\texttt{x=y}]\!]^\sharp$ before the new flags $f_{x1}, \ldots f_{xn}$ are set to the values of $f_{y1}, \ldots f_{yn}$ of `y`. Analogously for $[\![\texttt{x=\&y}]\!]^\sharp$. Allocating heap cells with $[\![\texttt{x=malloc()}]\!]^\sharp$ chooses a fresh address A_N. The fact that A_N is a valid cell is stated by adding the flag $f_{\exists N}$ with value one to b. The result `x` is set to point to the new address A_N using $[\![\texttt{x=\&N}]\!]^\sharp$.

The next three statements dereference pointers. Each statement dereferences the pointer contents using function $\mathit{deref} : \mathcal{A} \times P^{\mathcal{B}} \to \wp(\mathcal{A} \times P^{\mathcal{B}})$. This function partitions the passed-in state depending on the address that is pointed to:

$$\mathit{deref}(A, p \rhd b) = \bigcup_{(\_, B) \in p(A)} \phi(A, B, p \rhd b) \setminus \{\mathit{warn}\}$$

Here, a call $\phi(A, B, p \rhd b)$ to the auxiliary function $\phi : \mathcal{A} \times \mathcal{A} \times P^{\mathcal{B}} \to \wp(\mathcal{A} \times P^{\mathcal{B}}) \cup \{\mathit{warn}\}$ returns a state in which the cell at A points to the address B:

$$
\begin{aligned}
\phi(A, B, p \rhd b) = {} & \{(B,\ p \rhd b[\![f_1 + \ldots + f_n = 1 \wedge f_{AB} = 1 \wedge f_{\exists B} = 1]\!])\} && (1) \\
& \cup \{\mathit{warn} \mid b[\![f_1 + \ldots + f_n \neq 1]\!] \neq \emptyset\} && (2) \\
& \cup \{\mathit{warn} \mid b[\![f_1 + \ldots + f_n = 1 \wedge f_{AB} = 1 \wedge f_{\exists B} = 0]\!] \neq \emptyset\} && (3)
\end{aligned}
$$

where $p(A) = \{\langle f_1, A_1\rangle, \ldots, \langle f_n, A_n\rangle\}$. Value *warn* indicates a possible run-time error that has to be reported by the abstract interpreter: it is issued whenever a points-to set may point to none or more than one memory cell (Eqn. 2), or when the heap cell pointed to is not allocated due to, for instance, `free()` (Eqn. 3).

For each tuple $\langle B, p' \rhd b'\rangle$ returned by function *deref*, $[\![\texttt{*x=y}]\!]^\sharp$ writes `y` to address B, $[\![\texttt{x=*y}]\!]^\sharp$ reads from address B, and $[\![\texttt{free(x)}]\!]^\sharp$ deallocates address B in state $p' \rhd b'$, and the results are joined. Finally, the rule for the conditional partitions the state $p \rhd b$ such that `x` is `NULL` and non-`NULL`.

We note that our abstract semantics fulfills the soundness condition from above:

Observation 3. *For any statement* s*, there is* $\overline{[\![s]\!]^\natural} \circ \gamma_0 = \gamma_0 \circ [\![s]\!]^\sharp$*. Given* $\alpha_0 : \wp(\mathcal{A} \to \wp(\mathcal{A})) \to P^{\mathcal{B}}$ *with* $\alpha_0(c) = \sqcap\{a \mid c \subseteq \gamma(a)\}$ *there is* $\alpha_0 \circ \overline{[\![s]\!]^\natural} = [\![s]\!]^\sharp \circ \alpha_0$.

Moreover, the observation states that the abstract semantics exactly mirrors the concrete semantics. However, abstract states can grow arbitrarily large, and therefore fixpoint computations must apply a widening to ensure termination, as detailed in the next section.

3.2 Widening and Materialization on Access

An infinite growth of the abstract state space can be avoided by inserting a widening operator into every loop of the program [9] which ensures that any increasing sequence of states eventually stabilizes. We implement widening by summarization. By materializing memory cells on access, we are able to retain our abstract semantics as is. We detail both operations in turn.

Widening by Summarization. A challenge in analyzing loops is to ensure that the set of heap cells remains finite. We address this by summarizing nodes that are *abstract reachable* from the same program variables. This *abstract reachability* only considers the points-to graph and not the information in the numeric domain. For instance, the nodes A and B in Fig. 3a) are abstract reachable from `x` and `y`, even though B is not reachable according to the numeric state.

Our widening consists of two steps and is applied to a single state. In the first step, a partitioning of the heap-allocated nodes is calculated: nodes that are *abstract reachable* from the same set of stack variables are put into one partition. This ensures finiteness of fixpoint computation since the set of partitions is determined by the number of stack variables in scope. In the second step, each partition of heap nodes is collapsed into one summary node by summarizing all its members with the oldest member. This ensures that the symbolic addresses remain stable under the fixpoint calculations which enables the detection of a fixpoint using $\sqsubseteq$.

Materialization on Access. Materialization is performed as part of dereferencing pointers. To this end, each use of *deref* in Fig. 9 is replaced by *deref′* (defined below). For every summary node, *deref′* returns a state in which the node is materialized and a state in which the node is turned into a concrete node by simply removing the ghost node. The latter is necessary in case a summary node represents only one concrete node. The two cases are reflected by two calls to ϕ:

$$\mathit{deref}'(A, p \rhd b) \;=\; \bigcup_{(f,\,B)\in p(A)} \phi(A, B', p_1 \rhd b_1) \cup \phi(A, B, p_2 \rhd b_2) \setminus \{\mathit{warn}\}$$

where $p_1 \rhd b_1 = \mathit{materialize}(B, p \rhd b)$ is the result of materializing node B to $\{B'\} = \mathrm{dom}(p_1) \setminus \mathrm{dom}(p)$ and $p_2 \rhd b_2 = p \rhd \mathit{drop}_{f_{iB}, f_{oB}}(b)$ is the state in which B is no longer a summary node. Since the state $p_1 \rhd b_1$ returned by function

materialize is unchanged if B is not a summary, $deref'(A, p \rhd b)$ is identical to $deref(A, p \rhd b)$ when the points-to set of A only refers to concrete nodes.

We conclude this section by defining a concretization γ that takes an abstract state in $P^{\mathcal{B}}$ to concrete states in $\wp(\mathcal{A} \to \wp(\mathcal{A}))$.

Definition 4. *Concretization* $\gamma : P^{\mathcal{B}} \to \wp(\mathcal{A} \to \wp(\mathcal{A}))$ *is given by* $\gamma = \gamma_0 \circ \tau$ *where* $\tau : P^{\mathcal{B}} \to P^{\mathcal{B}}$ *is given by* $\tau(s) = fix_s(\lambda t \,.\, t \sqcup \bigsqcup_{A \in \mathcal{A}} \{u \mid A \in \mathcal{A},\ (\_, u) \in materialize(A, t)\}))$ *where* $fix_s(f)$ *is the least fixpoint of* f *greater than* s.

Here, function τ calculates the reflexive transitive closure of applying arbitrary materializations. Note that for abstract states with at least one summary node, τ returns a points-to domain with an infinite number of nodes paired with a numeric domain with an infinite number of dimensions. Materialization on access retains the following property which implies that all precision loss is incurred by widening:

Observation 4. *For any statement* s, *there is* $\overline{[\![s]\!]^\natural} \circ \gamma = \gamma \circ [\![s]\!]^\sharp$.

We now enhance this basic analysis by two predicates that are sufficient to distinguish between list, trees and graphs.

4 Two Simple Instrumentations

For the sake of presentation, we generalize the numeric domain from $\{0, 1\}$-vectors to $\mathbb{N}^n$. We define two simple numeric instrumentation variables and describe how they can be approximated by $\{0, 1\}$-flags.

4.1 Counting Outgoing Edges

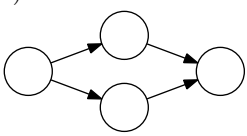
The shape graph in Fig. 7b) shows the result of materializing a heap cell D from a summary A. The obtained numeric state is $\langle f_{xA}, f_{iA}, f_{oA}, f_{xD}, f_{AD}, f_{DA}\rangle \in \{\langle y, x, 1-x, u, v, 1-v\rangle \mid u, v, x, y \in \{0, 1\}\}$, which allows flags f_{xA} and f_{xD} to be zero or one at the same time, indicating `NULL` or an invalid pointer value in variable `x`. The reason for this precision loss is that when summarizing two nodes M and N the flags f_{xM} and f_{xN} become indistinguishable due to folding.

We rectify this precision loss by tracking the number of outgoing edges of each node N in a numeric counter c_N^{out}. As this counter reflects the number of outgoing edges in the concrete graph, summarizing two nodes does not change this counter. However, when summarizing node O onto P, the flags f_{NO} and f_{NP} are merged into f'_{NP} which may be smaller than $f_{NO} + f_{NP}$. Thus, the sum $s_N := f_{N1} + \cdots + f_{Nk}$ of N's points-to flags may be smaller than c_N^{out}. We apply the information in c_N^{out} by adding the assertion $s_N = c_N^{out} = 1$ to $deref'$ and by enforcing $s_M \leq c_M^{out}$ whenever $p(A_M)$ receives new edges due to materialization.

Since the concrete semantics in Fig. 8 precludes points-to sets with more than one element, the invariant $c_N^{out} \leq 1$ can be enforced on the abstract state without losing soundness. This reduction with respect to the concrete semantics has been dubbed "hygiene condition" [18]. As a consequence, we implement only the lowest bit of the c_N^{out} counter as $\{0, 1\}$-flag.

a)
```
while(a != NULL) {
  tree * b = a->l;
  if(b==NULL) {
    b = a->r;
    free(a);
    a = b;
  } else {
    a->l = b->r;
    b->r = a->r;
    a->r = b; }}
```

b)



c)

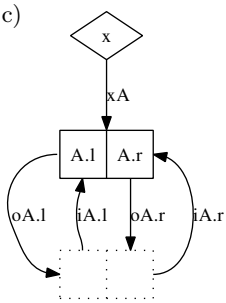


Fig. 10. Expanding a binary tree

4.2 Counting Incoming Edges

Fig. 10a) shows a C program that iteratively deallocates a binary tree pointed to by variable `a` by tree rotation. It requires that `a` holds a tree, that is, neither cycles nor diamond-shaped subgraphs as in Fig. 10b) are reachable from `a`, i.e., no heap cell can be accessed via more than one path. We guarantee the latter by simple *reference counting*: every memory cell M is equipped with a reference counter c^{in}_M that counts the references coming from *heap-allocated* cells. Whenever an entry $\langle f_{NM}, A_M \rangle$ is added to the points-to set of a node N, counter c^{in}_M is incremented by the value of f_{NM}. Analogously, the counter is decremented by the respective flag when an entry is removed from the points-to set. We then assert $c^{in}_M \geq 0$.

Our analysis is easily extended to compound memory cells like, for example, the nodes of a binary tree: summarization is done by making the components compatible one-by-one and then folding the compound cells onto each other. Fig. 10c) shows the abstract representation of an arbitrary binary tree. Summarization can infer the invariant that $f_{xA} = 1 - c^{in}_A \in \{0, 1\}$, that is, every heap cell has exactly one incoming edge. Indeed, this instrumentation is nearly sufficient to prove the absence of double-free-errors for the algorithm in Fig. 10a).

Since the numeric domain of our implementation is restricted to $\{0, 1\}$-values, we implement the counters c^{in}_N by a saturating binary 2-bit counter. We now show how the analysis can be efficiently implemented using a SAT solver.

5 Implementation and Experimental Results

The numeric state $b \in \mathcal{B}_n$ can be represented by a Boolean formula over n variables. We therefore implemented the functor domain $p \rhd b \in P^{\mathcal{B}_n}$ as a tuple consisting of a map for the points-to sets $p \in P$ and a single formula φ for $b \in \mathcal{B}_n$ that is either in conjunctive or in disjunctive normal form.

Since *expand* calculates an intersection of states, it can be straightforwardly implemented on a CNF formula by duplicating clauses. Analogously, *fold* has a straightforward implementation on a DNF formula. Join (resp. meet, in particular $b[\![\cdot]\!]$) is calculated by merging the DNF (resp. CNF) representations. The semantics of most statements requires the removal of dimensions from the vector set using $drop_{f_i}$, which corresponds to removing the quantifier in the formula

Table 1. Evaluation of our implementation

	instr. pred.	projections	avg. size vars/clauses	time	mem	loop iterations	warnings
three-element list	−	0	–	2 ms	10 MB	-	1
	+	0	–	3 ms	10 MB	-	0
singly linked list	−	33	20 / 36	30 ms	12 MB	4 / 3	6
	+	42	36 / 55	132 ms	14 MB	5 / 3	0
doubly linked list	−	12	23 / 37	13 ms	10 MB	2 / 2	7
	+	15	47 / 76	49 ms	13 MB	3 / 2	0
summarize tree	−	0	–	2 ms	12 MB	-	0
	+	0	–	2 ms	13 MB	-	0
access tree	−	38	15 / 15	48 ms	10 MB	3	6
	+	12	131 / 298	32 ms	17 MB	2	0
deallocate tree	−	95	32 / 45	380 ms	25 MB	5	28
	+	80	52 / 95	1.511 ms	26 MB	5	1
deallocate graph	−	96	34 / 51	375 ms	23 MB	5	44
	+	139	57 / 104	1.425 ms	26 MB	5	1

Table 2. Comparison with TVLA

TVLA	predicates	time	mem	iterations	warnings
singly linked list	16	176 ms	14 MB	40	0
deallocate tree	19	3,391 ms	20 MB	66	26

$\exists f_i \,.\, \varphi$. Rather than removing each dimension using *drop*, f_i is simply renamed to a fresh variable. Unused variables are eventually removed during the conversion between a CNF and a DNF formula. For the latter, we use the projection method of Brauer et al. [5], which is based on SAT-solving and calculates a minimal DNF representation from a CNF formula and vice versa.

Our implementation is written in Java using MINISAT v2.2. Table 1 shows the results for: summarizing a three-elemented list and accessing its first element as described in Sect 2.2; allocating and then deallocating a singly linked list; dto. for a doubly linked list of arbitrary length; summarizing a seven-element binary tree; accessing the leftmost innermost element of a binary tree; running the algorithm of Fig. 10a) on the summarized tree; running it on the summary of a diamond-shaped graph as in Fig. 10b). Each task is shown with instrumentation flags enabled (+) and disabled (−).

The columns show the number of calls to the projection function and the average size of the formulae (number of variables / number of clauses) at each projection. Running times and memory consumption on an Intel Core i7 with 2,66 GHz on Mac OS X 10.6 follow. The next two columns show how many iterations are required to find a fixpoint for the loop(s) in the examples. A "-" indicates that the example has no loop. For all examples except the last two, we could verify the absence of `NULL`- and dangling pointer dereferences. The warnings that occur in the deallocation examples arise when accessing a freed

heap node N through a stack variable. In the *tree* deallocation example, $c_N^{in} = 1$ but no summary node M pointing to N can be materialized with $f_{MN} = 1$. That is, this state is contradictory in itself and could be removed by a reduction step. Thus, while our analysis is expressive enough, an explicit reduction is required to remove this spurious warning. Since this reduction is orthogonal to the shape analysis itself, it has been omitted. Note that the warning in the last row of Table 1 is a true error.

We compare our prototype implementation with the latest TVLA 3 analyzer. Table 2 shows the running time of two of our examples when reformulated with TVLA predicates. The verification of the list example is slightly slower. A TVLA tree example that mimics our deletion algorithm for trees is shown in the second row. Although TVLA provides predicates that can express the invariant, it is unable to prove that a node is only freed once and, hence, emits warnings. This is curious, since TVLA provides several tree invariants that are silently enforced by integrity constraints.

6 Related Approaches to Shape Analysis

Using Boolean functions for analyzing points-to sets is a re-occurring theme in the literature, although they are often represented as binary decision diagrams rather than CNF formulae [3]. Our work shows how points-to analysis can be enriched with summaries of heap structures, thereby giving a new answer to the question of how to merge the regions created at call sites of `malloc` [14]. Moreover, our analysis could replace ad-hoc forms of shape analysis such as summarizing all heap cells but the last one allocated [1]. The latter is used to allow strong updates on heap cells that are allocated in a loop. By materializing on access and summarizing through widening, our analysis refines this ad-hoc strategy by a dynamic, semantics-driven strategy.

One peculiarity of our analysis is the ability to distinguish lists, trees, and graphs using only relational information associated with each node. This design simplifies the abstract transfer functions in that they only have to update information of the nodes that are actually accessed, we call this a node-local analysis. One motivation for using separation logic for shape analysis is exactly this ability, namely that an update in separation logic retains a so-called frame that describes the part of the heap that is not being accessed. Moreover, the inductively defined predicates of separation logic [17] are also local in that they relate each node to a fixed number of neighboring nodes. Using these predicates, a linked list is specified as $\mathit{list}(x) \equiv \mathbf{emp} \vee (\exists y \,.\, x \mapsto y \bullet \mathit{list}(y))$ where the *separating conjunction* $h_1 \bullet h_2$ expresses that heaps h_1 and h_2 do not overlap. Interestingly, a summary node N and its ghost node live at different addresses and, thus, they can be seen as being separated by the separating conjunction.

While automatic analysis using separation logic relies on a symbolic representation of heap shapes, our approach is based on a Boolean domain whose join operation can infer new invariants. Inferring new invariants (predicates) in the context of separation logic is in general not yet possible [8]. However, templates,

namely higher-order predicates, have been automatically instantiated in order to infer hierarchical data structures [2]. Since our current approach already infers any node-local invariant, future work should address the inference of hierarchical shapes, for instance, to verify operations on a list of independent circular lists.

A different approach to shape analysis is the TVLA framework [18] where the shape of the heap is described by a set of core predicates such as $n(x, C)$ (indicating that $x.n$ points to C). Other, so-called instrumentation predicates, such as $r_x(C)$ (node C is reachable from x) are defined in terms of core predicates. Transfer functions that describe the new value after a program statement must be given at least for all core predicates. Two heap cells A_1 and A_2 are summarized by merging the truth values of the predicates that mention them: for example, if $v_i = n(x, A_i)$, then the merged value is v_1 if $v_1 = v_2$ or $\frac{1}{2}$ if $v_1 \neq v_2$. Indeed, TVLA's three-valued interpretation approximates our set of Boolean vectors b by using the value $\frac{1}{2}$ for a predicate p iff $\boldsymbol{b}(p)$ is not constant for all $\boldsymbol{b} \in b$. This is troublesome when, for example, re-evaluating $r_x(C)$ after summarizing two nodes that lie on a path from x to node C: although C is still reachable from x, its re-evaluation on the core predicates yields $\frac{1}{2}$. Indeed, devising precise transfer functions for instrumentation predicates in TVLA is considered a "black art" [16, Sect. 4]. This triggered work on automatic synthesis of transfer function [16].

For certain predicates it is particularly challenging to define a precise transfer function, one of them being $r_x(C)$. The reason is that these predicates use a recursive, transitive closure operator, whose calculation in general requires the whole heap state and, hence, incurs the imprecision of core predicates over summary nodes. In contrast, our analysis only requires node-local information, thereby eschewing the need to perform calculations using information in summary nodes. This strength comes at the cost of rather unintuitive invariants: For instance, in TVLA a predicate would directly state that a summary node A represents an acyclic list, whereas in our analysis the relation $f_{oA} \neq f_{iA}, f_{xA} \neq f_{iA}$ with $[A_x \mapsto \{\langle f_{xA}, A_A\rangle\}]$ states that A is a list which is acyclic if and only if `x` is pointing to it (here f_{iA} and f_{oA} decorate the edges to the ghost node of A).

The flag $f_{\exists N}$ (node N is allocated) resembles the TVLA property *present* of [13]. While our c_N^{in} counter can be seen as a generalization of TVLA's $is(N)$ predicate (is shared, indicating that N has more than one incoming edge), it is actually motivated by work on classifying data structures by Tsai [21]. Indeed, our diamond-shaped subgraph in Fig. 10 can be classified as a shared, acyclic set of heap nodes. We follow Tsai in using reference counting to detect this sharing.

The use of Boolean formulae for a TVLA-style shape analysis was also advocated by Wies et al. [15]. Updates in their predicate abstraction approach are also node-local. However, their analysis does not consider summary nodes. Furthermore, due to the lack of an adequate projection algorithm [5] they deliberately destroy relational information using cartesian abstraction.

An interesting approach to shape analysis is given by Calcagno et al. [7] who propose to perform a backward analysis using abduction. While it is well-known [6] that Boolean functions lend themselves to this kind of task, future

work has to address if the combined points-to and Boolean domain also allows for abduction.

A natural extension is the use of a more generic numeric domain like polyhedra [10] in which our instrumentation counters c_N^{in} and c_N^{out} require no special encoding. This would also raise the question of how relational numeric invariants between a summary and its ghost node can be inferred, for instance, to deduce that a list is sorted [12,8]. Future work will address these challenges.

7 Conclusion

We proposed and formalized a fully automatic shape analysis that expresses the heap shape using a single graph and a Boolean function. Our analysis is highly precise by exploiting the ability of Boolean formulae to express relations between heap properties. Due to this relational information, our analysis distinguishes lists from trees from graphs by using only predicates pertaining to the existence of nodes and edges and the number of incoming and outgoing edges.

The key insight is that this relational information can be precisely inferred using a relational *fold* and *expand* [19] that we adapted to Boolean functions. Using these operations, our shape analysis has the ability to infer new shape invariants automatically. We have shown how an efficient implementation of the analysis is possible using SAT solving.

References

1. Balakrishnan, G., Reps, T.: Recency-Abstraction for Heap-Allocated Storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006)
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape Analysis for Composite Data Structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
3. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. In: Programming Language Design and Implementation, pp. 103–114. ACM, San Diego (2003)
4. Boquist, U., Johnsson, T.: The Grin Project: A Highly Optimising Back end for Lazy Functional Languages. In: Kluge, W.E. (ed.) IFL 1996. LNCS, vol. 1268, pp. 58–84. Springer, Heidelberg (1997)
5. Brauer, J., King, A., Kriener, J.: Existential Quantification as Incremental SAT. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 191–207. Springer, Heidelberg (2011)
6. Brauer, J., Simon, A.: Inferring Definite Counterexamples through Under-Approximation. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 54–69. Springer, Heidelberg (2012)
7. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional Shape Analysis by means of Bi-Abduction. In: Principles of Programming Languages, Savannah, Georgia, USA, ACM (2009)
8. Chang, B.-Y.E., Rival, X.: Relational Inductive Shape Analysis. In: Principles of Programming Languages, pp. 247–260. ACM (2008)

9. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Principles of Programming Languages, pp. 269–282. ACM, San Antonio (1979)
10. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Constraints among Variables of a Program. In: Principles of Programming Languages, Tucson, Arizona, USA, pp. 84–97. ACM (1978)
11. Hind, M., Pioli, A.: Which Pointer Analysis Should I Use? In: International Symposium on Software Testing and Analysis, Portland, Oregon, USA, pp. 113–123. ACM (2000)
12. McCloskey, B., Reps, T., Sagiv, M.: Statically Inferring Complex Heap, Array, and Numeric Invariants. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 71–99. Springer, Heidelberg (2010)
13. McCloskey, B.: Practical Shape Analysis. PhD thesis, EECS Department, University of California, Berkeley (May 2010)
14. Nystrom, E.M., Kim, H.S., Hwu, W.W.: Importance of Heap Specialization in Pointer Analysis. In: Flanagan, C., Zeller, A. (eds.) Program Analysis for Software Tools and Engineering. ACM, Washington, DC (2004)
15. Podelski, A., Wies, T.: Boolean Heaps. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 268–283. Springer, Heidelberg (2005)
16. Reps, T., Sagiv, M., Loginov, A.: Finite Differencing of Logical Formulas for Static Analysis. Transactions on Programming Languages and Systems 32, 24:1–24:55 (2010)
17. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, Copenhagen, Denmark, pp. 55–74. IEEE (2002)
18. Sagiv, M., Reps, T., Wilhelm, R.: Parametric Shape Analysis via 3-Valued Logic. Transactions on Programming Languages and Systems 24(3), 217–298 (2002)
19. Siegel, H., Simon, A.: Summarized Dimensions Revisited. In: Mauborgne, L. (ed.) Workshop on Numeric and Symbolic Abstract Domains, ENTCS, Venice, Italy. Springer (2011)
20. Simon, A.: Splitting the Control Flow with Boolean Flags. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 315–331. Springer, Heidelberg (2008)
21. Tsai, M.-C.: Categorization and Analyzing Linked Structures. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, Illinois, USA (1994)