Shape Analysis with Connectors[1]

Holger Siegel and Axel Simon

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

Abstract. We extend off-the-shelf shape analyses with the ability to infer numeric relations between directly or indirectly connected heap cells. Specifically, we introduce the concept of *connectors*, an instrumentation that retains relations between heap cells even if these cells are merged into summary nodes. Managing connectors is based on applying generic *fold* and *expand* operations on a numeric abstract domain. Connectors are thus a universal tool to enhance shape analyses with any numeric analysis. We show how connectors provide the ability to infer invariants of non-trivial heap structures such as sorted/skip lists and search trees.

1 Introduction

Proving the absence of `NULL`- and dangling pointer dereferences in heap-manipulating programs requires the ability to deal with an a priori unbounded number of heap cells. The problem of summarizing these heap cells into a finite set of abstract heap cells is known as shape analysis. Traditionally, the shape of the heap is described using a logic such as separation logic [14] or three-valued logic analysis (TVLA) [15]. The challenge of both approaches is the synthesis of appropriate predicates that are able to express the invariants in the program.

While this challenge has been addressed quite successfully for the synthesis of shape predicates, the synthesis of numeric invariants remains a stronghold of numeric abstract domains. This paper therefore addresses how to combine an analysis of heap shapes with the inference of numeric relations on the contents of heap cells. Such a combination is a prerequisite to expressing invariants for sorted lists, search trees and other common data structures. The presented analysis extends any shape analysis that summarizes and materializes nodes in a graph that represents the abstracted heap. We therefore consider the actual shape analysis to be an oracle that informs our analysis about nodes that are combined into summaries and about accesses to nodes. In the latter case, we assume that a summarized node is being materialized. For the sake of the introduction, we also assume that the underlying shape analysis removes infeasible points-to edges.

Consider the sorted list in Fig. 1a) that is summarized into a single summary node in Fig. 1b). When analyzing the numeric content using the abstract domain of polyhedra [2], the cell content reduces to $1 \leq x \leq 3$ that we abbreviate by the interval notation $x \in [1, 3]$. Note that summarizing the heap has discarded

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

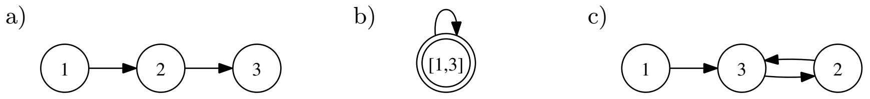

Fig. 1: summarizing a linked list

any information about the order of list elements. Figure 1c) shows one possible materialization of this summary where the order of list elements is not preserved.

The central idea is to construct so-called connectors between heap nodes that allow numeric domains to infer numeric relations between the nodes. Figure 2a) shows the connectors attached to the same three-element list as in Fig. 1a). Each connector holds copies of the contents of the cells that it connects. When summarizing the cells containing one and two, the connector between them stays on the edge that thereby points from the summary node to itself. When folding the next node onto the summary, the cell content and the connector are folded in unisono, yielding the state in Fig. 2b). The connector retains the invariant that the successor of each node contains the next larger value. When a cell is materialized from the summary, the corresponding connector is expanded too, as done in Fig. 2c). Once it is certain that one cell points to another, the content of the corresponding connector is equated to the two cells it connects. For instance, upon accessing the second list node, another cell and connector is materialized before the first connector $a, a+1$ is equated with the first and the new second heap cell. The equated connector can now be removed, yielding the heap in Fig. 2d). Note how equating the connector has restricted the content of the first cell to $a \in [1,2]$ and that of the second cell to $a+1 \in [2,3]$.

The connectors presented so far are *one-step* connectors. They are sufficient to express relations between neighboring nodes like sortedness properties, but they cannot express invariants of search trees, namely, that e.g. all elements reachable via the right branch of a node contain a greater value than the node itself. transitive properties like reachability. In order to express these reachability invariants, we additionally introduce *transitive* connectors that express a numeric relation between a cell and all cells that are transitively reachable from this cell.

In summary, we present an inference of numeric heap invariants that is generic in the shape analysis and in the numeric domains, based on these novel ideas:

– We use generic relational *fold* and *expand* operators [16] to infer numeric relations using any numeric domain. This improves over the state-of-the-art of tracking a fixed set of predicates as part of the shape analysis.
– We introduce the concepts of one-step and transitive connectors that express the relation between nodes on the heap. The handling of connectors is linked to the operations of a shape analysis which is a parameter of our framework.
– We illustrate how several important data structure invariants can be inferred.

After presenting an interface to existing shape analyses and numeric domains, Sect. 3 introduces one-step connectors and details the interaction between the shape analysis and our connectors. Section 4 presents transitive connectors, followed by an example of their application to complex data strutures. Section 6 presents experimental results before Sect. 7 discusses related work and concludes.

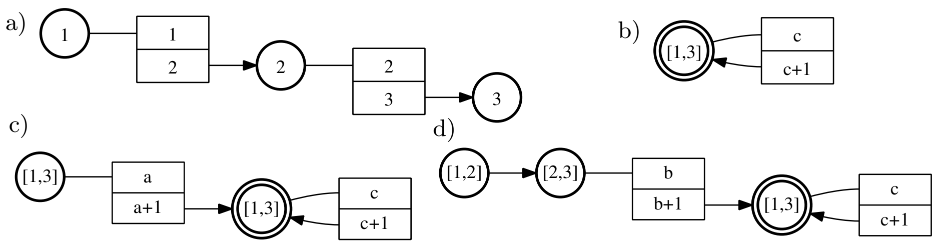

Fig. 2: summarizing a linked list with one-step connectors

2 Preliminaries

This section introduces the interface to an off-the shelf shape analysis and details necessary operations on the numeric domain.

2.1 Interface to the Shape Analysis

Let $\mathcal{M}$ denote a set of non-overlapping memory regions, consisting of heap-allocated cells and stack variables. Each region contains of a set of non-overlapping fields that are written $A.l$ with $A \in \mathcal{M}$ and $l \in \Sigma$ where Σ denotes the set of all field names. As a convention, we write $A.x$, $A.y$ for fields containing numeric values and $A.p$, $A.q$ for fields containing pointers, although a field may contain either, a pointer or a value. For brevity, a stack variable $S.x$ is also written as x.

In order to be agnostic to the employed shape analysis, we define an interface that the shape analysis must implement and that is sufficient to infer connectors. Let S denote the set of abstract heap shapes and N denote an abstract numeric domain. Each state $s \in S$ of the shape analysis is paired with a numeric state $n \in N$ that holds valuations for all fields that contain numeric values, that is, $A.x$ and $A.y$ are variables in the numeric domain. We assume that the transfer function of assignment, tests etc. modify the numeric domain accordingly.

The interface between the shape analysis and our connector inference is given by only four functions. These may already be implemented by the shape analysis in order to summarize the numeric content of a cell. In this case, they have to be redirected to our connector inference. The four operations are as follows:

- Operation $\mathit{fold}_{S,C}$ merges the content of C (usually a concrete node) with that of S (usually a summary node), thereby removing all information on C. In the numeric domain, all fields $C.x_i$ are folded onto the respective fields $S.x_i$. The connector inference translates each call $\mathit{fold}_{S,C}$ to a call $\mathit{fold}_{S\,S_1 \ldots S_n,\,C\,C_1 \ldots C_n}$ that simultaneously folds nodes S and C and the nodes S_i, C_i that are stored in the connectors attached to S and C. By this, the numeric invariant of Fig. 2b) is inferred.
- Operation $\mathit{expand}_{S,C}$ is the inverse of operation *fold*. It duplicates the information in memory region S to a new memory region C. Its effect on the numeric domain and the connectors are symmetrical to those of *fold*, allowing

to restore the numeric properties of the expanded memory region as shown in Fig. 2c). Both, the *fold* and *expand* mechanisms, are detailed below.
- Operations $assume(C.f \mapsto D)$ and $assume(C.f \not\mapsto D)$ informs the numeric domain (and the connector inference) that field $C.f$ definitely contains (respectively: does not contain) a pointer to region D.

Moreover, we require that the shape analysis provides a set $\mathcal{E} \subseteq Edges$ of possible edges where $Edges = \{A.l \mapsto B \mid A, B \in \mathcal{M}, l \in \Sigma\}$. In other words, $\mathcal{E}$ over-approximates all possible points-to configurations of the current abstract state. We now detail how two common shape analyses can provide the interface of *fold*, *expand* and *assume*.

2.2 Separation Logic

Separation logic [14] describes a heap using a first order logic formula over heap assertions $A.l \mapsto B$, stating that the heap consists of a cell A with field l pointing to a cell B that is not necessarily part of this heap. A *separating conjunction* $h_1 * h_2$ states that the heap cells in h_1 and h_2 live in *separate* heaps. Thus, the heap described by formula $A.l \mapsto B * B.l \mapsto A$ consists of two different nodes pointing to each other. In contrast, both arguments of a logical disjunction describe the *same* heap, so that a expression $A.l \mapsto C \vee B.l \mapsto \textbf{null}$ stands for a single heap node which is named A or B. In the following, we associate each heap node with a distinct and unique name, so that each node in the heap is accompanied by exactly one node A in the numeric state. A set of possible edges $\mathcal{E}$ is then given by the set of all points-to terms $A.l \mapsto B$ in the formula.

Separation logic allows to summarize sets of similar heap cells by recursively defined predicates. For instance, a list from node A ending at B can be characterized by the predicate $ls(A, B) = (A.n \mapsto B) \vee (\exists C \,.\, A.n \mapsto C * ls(C, B))$. Each occurence of predicate $ls(A, \cdot)$ in a heap formula represents a list that is summarized into one summary node A. In the numeric domain, node A holds the summarized numeric content of this list. When a list element is accessed, the predicate $ls(A, B)$ is *unfolded* by replacing it with its right hand side. Since the right hand side consists of a disjunction $(A.n \mapsto B) \vee (\exists C \,.\, A.n \mapsto C * ls(C, B))$ of different heap shapes, both alternatives have to be considered separately in the numeric domain: replacing $ls(A, B)$ with $(A.n \mapsto B)$ just turns the summary node into a concrete node, so we do not need to adjust the numeric state. Replacing $ls(A, B)$ with $(\exists C \,.\, A.n \mapsto C * ls(C, B))$ introduces a new node C, therefore a new node is also introduced in the numeric domain via operation $expand_{A,C}$. Symmetrically, a formula that matches the right hand side of a predicate definition can be replaced by its left hand side. In this case a corresponding operation $fold_{A,C}$ is executed on the numeric domain. This mechanism allows to summarize any heap shape for which a recursive separation logic predicate exists.

Whenever a points-to relation $A.n \mapsto B$ is unconditionally true, operation $assume(A.n \mapsto C)$ should be executed on the numeric domain. Analogously, if a points-to relation $A.n \mapsto B$ is definitely unsatisfiable, then an $assume(A.n \not\mapsto B)$ operation should be executed.

2.3 Three-Valued Logic Analysis (TVLA)

The TVLA framework for shape analysis [15] represents the heap by a finite set of abstract summary nodes, each of them representing a set of concrete nodes. The shape of the heap is characterized by a set of user-supplied predicates. Usually, at least a $\mathit{next}(A.l, C)$ predicate is given, indicating whether a points-to edge $A.l \mapsto C$ exists. The set of *next*-predicates that occur in the current state defines the set of edges $\mathcal{E}$. A valuation function takes each predicate to a value in $\{0, \frac{1}{2}, 1\}$. Whenever the valuation of $\mathit{next}(A.l, C)$ changes to zero or one, a corresponding $\mathit{assume}(A.l \not\mapsto C)$ or $\mathit{assume}(A.l \mapsto C)$ operation is performed on the numeric state. Folding and expanding corresponds to summarizing and materializing a node in TVLA. During updating a cell, all involved predicates must map to zero or one in order to determine the semantics of an access. To this end, the state is temporarily duplicated for each involved predicate with value $\frac{1}{2}$. The task of updating a numeric state in the presence of a set of formulae has been addressed elsewhere [3]. Variants of TVLA refine the three-valued mapping with a Boolean formula, avoiding the duplication of the TVLA formula for $\frac{1}{2}$ values [13]. Moreover, the information in the Boolean formula can be folded and expanded in the same way as the numeric information [17], thereby giving a unified formulation of shape analysis as an analysis over numeric variables.

Given the interface to the shape analysis, we now turn to numeric domains.

2.4 Numeric Domain Operations

We illustrate our analysis on $N_{\mathcal{X}} := \mathit{Poly}_{\mathcal{X}}$, the numeric domain of convex polyhedra [2] over variables $\mathcal{X}$, although any other abstract domain may be used. In the context of our analysis, $\mathcal{X} = \{A.x, A.y, \ldots \mid A \in \mathcal{M}\}$, and we write N instead of $N_{\mathcal{X}}$ where the support set is clear from the context. We define the following three functions that return a new numeric state:

– Function $\mathit{copy}_{A,B} : N \to N$ adds a copy B of region A to a numeric state $n \in N$ by performing the assignment $B.l := A.l$ for all $A.l$ currently in n.
– Function $\mathit{drop}_A : N \to N$ eliminates all variables $A.l$ from a state $n \in N$. In the abstract domain of polyhedra, elimination of a variable x corresponds to a projection onto the Euclidian subspace without dimension x.
– Function $[\![A = B]\!] : N \to N$ restricts a state $n \in N$ by adding the constraint $A.l = B.l$ for every field $l \in \Sigma$, such that $A.l$ or $B.l$ occurs in n.

We require two more numeric domain functions that mirror the effect of *fold* and *expand* on memory regions. To this end, we recall the following definition [16]:

Definition. *Given the m dimensions $ab\ldots \in \mathcal{X}^m$ and m dimensions $a'b'\ldots \in \mathcal{X}^m$, define $\mathit{fold}_{ab\ldots,a'b'\ldots} : N \to N$ and $\mathit{expand} : N \to N$ as follows:*

$$\begin{aligned} \mathit{fold}_{ab\ldots,a'b'\ldots}(n) &= \mathit{drop}_{a'b'\ldots}(n \sqcup \mathit{swap}_{ab\ldots,a'b'\ldots}(n)) \\ \mathit{expand}_{ab\ldots,a'b'\ldots}(n) &= n \sqcap \mathit{swap}_{ab\ldots,a'b'\ldots}(n) \end{aligned}$$

where operation $\mathit{drop}_{a'b'\ldots}$ eliminates the given variables (analogous to drop_A) and $\mathit{swap}_{ab\ldots,a'b'\ldots}$ exchanges the given primed and unprimed identifiers.

The *fold* function merges the information over $a'b'\dots$ with that over $ab\dots$ and then removes $a'b'\dots$. Intuitively, *fold* discards all relations between $ab\dots$ and $a'b'\dots$ but retains any relational information that $ab\dots$ and $a'b'\dots$ have with other variables. Symmetrically, *expand* duplicates relations within $ab\dots$ to $a'b'\dots$ but, unlike an assignment $a := a'$, induces no equality between a and a'. For instance, $\mathit{expand}_{ab,a'b'}(n)$ on a polyhedron n defined by the vertices $\langle a, b\rangle \in \{\langle 0,0\rangle, \langle 1,1\rangle\}$ yields $\langle a,b,a',b'\rangle \in \{\langle 0,0,0,0\rangle, \langle 0,0,1,1\rangle, \langle 1,1,0,0\rangle, \langle 1,1,1,1\rangle\}$. Note that the equality $a = b$ in the input state n has been preserved and duplicated to $a' = b'$ in the expanded state, but that $a = a'$ does not hold.

Applying *fold* and *expand* on memory regions S and C naturally translates to applying the corresponding numeric operations on the sequences of numeric variables $S.x\,S.y\dots$ and $C.x\,C.y\dots$. Care must be taken that both sequences contain the same sequence of field indices by possibly inserting lacking fields. In the sequel, we write $n := \mathit{fold}_{S,C}(n)$ to update a numeric state and assume that the translation to sequences of numeric variables is done implicitly.

These preliminary definitions suffice to introduce the connector inference.

3 One-step Connectors

The connector inference tracks relational information between heap cells that preserves certain information that may otherwise be lost when heap cells are summarized. This section addresses the key questions about connectors, namely how to create them and how to use their information, by first introducing *one-step connectors* that are generalized to *transitive connectors* in the next section.

A one-step connector tracks the relation between two cells that are directly connected via a points-to edge. Thus, we define the state of the connector inference as a partial map $o : \mathit{Edges} \nrightarrow \mathcal{M} \times \mathcal{M}$. A connector between a field $A.l$ and a memory region B exists iff $o(A.l \mapsto B)$ is defined. In this case the tuple $\langle A', B'\rangle := o(A.l \mapsto B)$ is a connector that describes the contents of A and B in all states where the edge $A.l \mapsto B$ exists. Valuations for all fields $A'.x_i$, $B'.x_i$ are held in the numeric domain, so that relations between these fields can be inferred. Suppose that the summary node in Fig. 2b) is S and that its self-edge is $S.\mathit{next}$, the heap consists of a summary node S and a connector $S.\mathit{next} \mapsto S$ with $o(A.l \mapsto B) = \langle\langle S'.x, S'.\mathit{next}\rangle, \langle S''.x, S''.\mathit{next}\rangle\rangle$. Then the numeric domain holds the constraints $S.x \in [1,3]$, $S'.x = c$ and $S''.x = c+1$ for some $c \in [1,2]$.

Creating Connectors. In order to curtail an information loss due to summarization, connectors have to be created before a memory region C is folded onto region S by intercepting the operation $\mathit{fold}(S,C)$. Based on the set of may-edges $\mathcal{E}$, we identify all edges $E \subseteq \mathcal{E}$ that are connected to S or to C and for which no connector exists. For each such edge $A.l \mapsto B \in E \setminus \mathit{dom}(o)$, we create two memory regions A', B', update the numeric state to $n := \mathit{copy}_{A,A'}(\mathit{copy}_{B,B'}(n))$ and extend o with the mapping from $A.l \mapsto B$ to $\langle A', B'\rangle$.

Folding Cells. When two regions S and C are folded using $\mathit{fold}(S,C)$, the connectors attached to S and C also have to be folded onto each other. A connector

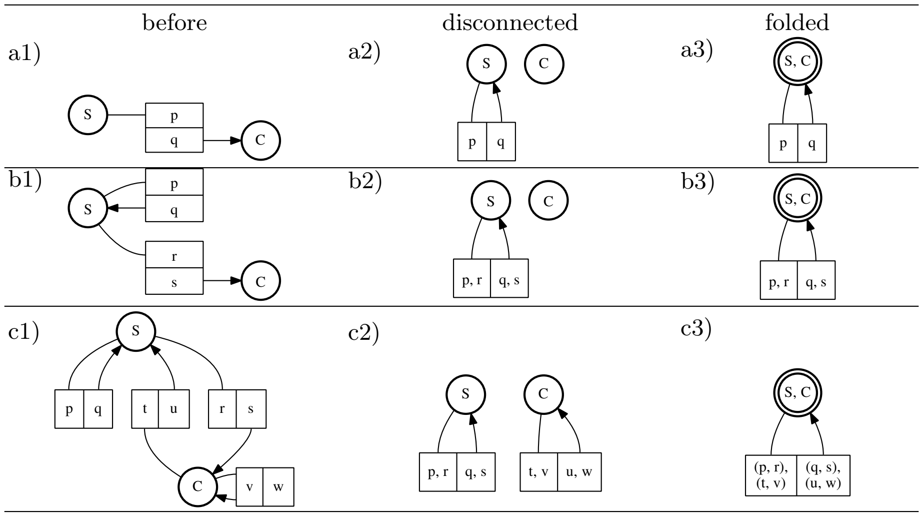

Fig. 3: Folding two cells. The result of simultaneously folding r onto p and s onto q is denoted by p, r and q, s. Folding t, v onto p, r is denoted by $(p, r), (t, v)$.

represents values of nodes *in case they are connected*, so for a non-existing edge arbitrary connectors may be introduced, as they will never be applied. Thus, when two nodes A and B are summarized where a connector $C.l \mapsto A$ but no corresponding connector $C.l \mapsto B$ exists, the connector $o(C.l \mapsto A)$ becomes the connector of the summarized node. Analogously for connectors $A.l \mapsto C$ and $B.l \mapsto C$. Connectors may be absent in two contexts, namely for connectors to other nodes than S or C and for edges between S and C, as detailed now.

The first step in folding two regions S and C is to ensure that no connectors between S and C exist, as these would be dangling once C is removed. Hence, any connector on the edge $C.l \mapsto S$ is "bent" to the edge $C.l \mapsto C$ as shown in Fig. 3a2). If a connector already exists for edge $C.l \mapsto C$, it is folded onto the bent connector as shown in Fig. 3b2). The edges from S to C are treated symmetrically, thereby giving a strategy for removing all edges between S and C. Figure 3 illustrates this strategy: In a2), the connector from S to C of the heap in a1) is bent to S, giving the heap in a3). In b1), the connector reaching from S to C is turned into a self-looping connector that reaches from S to S. This requires its numeric content to be folded onto the already existing connector from S to itself, giving the heap b2). In c1), both the connectors from S to C and from C to S have to be folded onto the connector from S to S and the connector from C to C, respectively, giving the heap in c2).

The intermediate result are two nodes S and C that are not connected. Now regions S and C have to be folded onto each other together with the connectors attached to them. The goal is to update the numeric state to $n := \mathit{fold}_{S\,S_1 \ldots S_n,\,C\,C_1 \ldots C_n}(n)$ where $S_1 \ldots S_n$ and $C_1 \ldots C_n$ are sequences of connectors that are folded alongside the actual cells S and C. To this end, define sequences s and c as $s := S$ and $c := C$ and extend them with connectors:

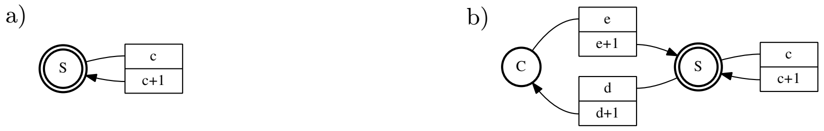

Fig. 4: expanding a list element

- For each node $A \not\in \{S, C\}$ for which connectors $\langle A', S' \rangle := o(A.l \mapsto S)$ and $\langle A'', C' \rangle := o(A.l \mapsto C)$ exist, add them to s and c by updating $s := sA'S'$ and $c := cA''C'$ and remove connector $A.l \mapsto C$ from o.
- For each node $A \not\in \{S, C\}$ for which connectors $\langle S', A' \rangle := o(S.l \mapsto A)$ and $\langle C', A'' \rangle := o(C.l \mapsto A)$ exist, add them to s and c by updating $s := sA'S'$ and $c := cA''C'$ and remove connector $C.l \mapsto A$ from o.
- If there exist connectors $\langle S', S'' \rangle := o(S.l \mapsto S)$ and $\langle C', C'' \rangle := o(C.l \mapsto C)$, add them to s and c by updating $s := sS'S''$ and $c := cC'C''$ and remove connector $C.l \mapsto C$ from o.

Finally, the numeric content is folded by updating the numeric state to $n := \mathit{fold}_{s,c}(n)$. Now all connectors attached to C that had a corresponding connector attached to S are removed from o and their numeric contents are summarized.

Note that there may still be connectors attached to C that have no corresponding connector attached to S. These are "moved" to region S by updating o so that every connector $\langle A', C' \rangle := o(A.l \mapsto C)$ is replaced with a connector $A.l \mapsto S$ such that $\langle A', C' \rangle = o(A.l \mapsto S)$. Analogously, every connector $\langle A', C' \rangle := o(C.l \mapsto A)$ originating from C is replaced by a connector $S.l \mapsto A$ originating from S. The third column of Fig. 3 illustrates this situation. In a3) and b3) the connectors attached to S remain unchanged during summarization, whereas in c3) the connector attached to C is folded onto the connector attached to S together with the numeric contents of the nodes.

Expanding Summaries. Expanding a summary node S to a concrete node C is done by simply reverting the steps of folding two nodes to one summary node: First, S is expanded to an exact copy C, then each self-looping connector at S is expanded to a connector from S to C and finally each self-looping connector at C is bent to S. Figure 4a) shows a summary S of a consecutively numbered list. In Fig. 4b), a node C has been materialized and the connector $S.\mathit{next} \mapsto S$ has been expanded to connectors $S.\mathit{next} \mapsto C$ and $C.\mathit{next} \mapsto S$, describing every edge from C to S or vice versa, provided that it exists, points to the next higher value.

Applying Connectors. The information contained in a connector can be utilized whenever the shape analysis determines that an edge $A.l \mapsto B$ definitely exists. In this case, the connector $\langle A', B' \rangle := o(A.l \mapsto B)$ can be applied by updating state n to state $n := [\![A = A', B = B']\!](n)$. This operation is triggered by the shape analysis via operation $\mathit{assume}(A.l \mapsto B)$. In contrast, operation $\mathit{assume}(A.l \not\mapsto B)$ informs the connector inference that a connector is spurious

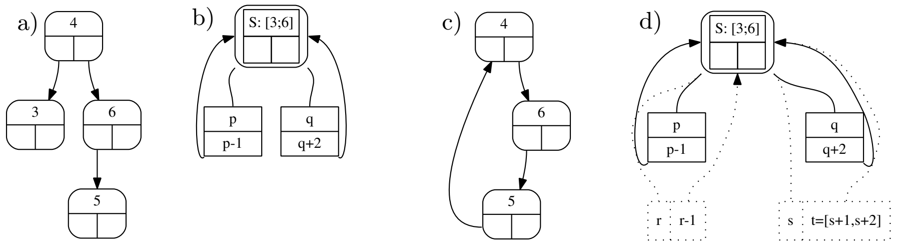

Fig. 5: summarizing a binary search tree

and that the relations it expresses do not hold. In the first case, the information in the connector is redundant after applying it, whereas in the second case its information is not applicable. Hence, in both cases we remove the connector $A.l \mapsto B$ from o and A', B' from n using $drop_{A'} \circ drop_{B'}$.

Since relations between neighboring nodes are not always sufficient for describing complex heap structures, we now turn to *transitive* connectors.

4 Transitive Connectors

Consider summarizing the search tree in Fig. 5a). One-step connectors can express relations between a node and its left or right child. This is shown in Fig. 5b) where connector $S.l \mapsto S$ indicates that the left child is exactly one smaller and connector $S.r \mapsto S$ indicates that the right child is exactly one greater. However, the concrete heaps modeled by this abstraction include the heap shown in Fig. 5c), which obviously violates the search tree invariant as it is cyclic.

Since one-step connectors are insufficient for expressing certain invariants such as that of a search tree, we now address how to generalize them to connectors that relate nodes that are indirectly connected via arbitrary paths. Considering the set of field names Σ as an alphabet, then the words over Σ describe possible paths that may connect one node with another. Suppose that we annotate each connector with the formal language that describes the set of possible paths for which the connector models a relation. Then a one-step connector $A.l \mapsto B$ can be considered as tracking a relation for the one-elemented language $\{l\}$.

In principle, one can track connectors for any set of paths. For simplicity, we will confine this presentation to connectors that capture the relation between node A and all nodes reachable from A via some arbitrary path beginning in field $A.l$. Thus, we partition the possible paths into languages $\omega_l = l\Sigma^*$ for all field names $l \in \Sigma$. In the following, we will denote the set of paths connecting node A with node B via ω_l by $A.l \rightsquigarrow B$. The operations presented for these paths ω_l can straightforwardly be adopted to more specific paths.

Figure 5d) shows the summarized tree of Fig. 5a) with a dotted transitive connector $\langle r, r-1\rangle$ for $S.l \rightsquigarrow S$, stating that, for each tree element, the nodes in the left subtree are exactly one smaller (implying at most one node), and a transitive connector $S.r \rightsquigarrow S$ whose content $\langle s, t\rangle$ with $t \in [s+1, s+2]$ states that all contents of a node's right subtree are strictly greater than that of S.

a) $y \in \{x+1,\, x+2\}$
$s \in \{r+1,\, r+2\}$

b) $y \in \{x+1,\, x+2\}$
$t = q-1 \wedge t \in \{x+1,\, x+2\}$

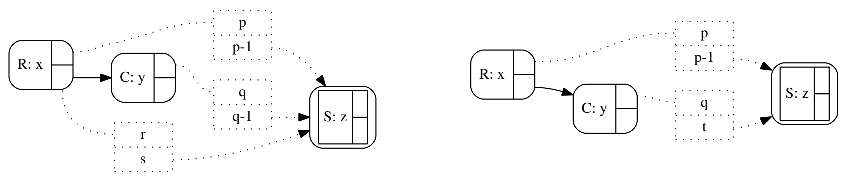

Fig. 6: expanding a summarized tree with transitive connectors

Creating Transitive Connectors. Transitive connectors are created in the same way as one-step connectors: For every pair of heap cells A and B and field $l \in \Sigma$ for which the set $\mathcal{E}$ of possible edges contains a path $p \in l\Sigma^*$ from A to B starting with field l, a transitive connector $A.l \rightsquigarrow B$ is created by copying the numeric contents of A and B into regions A' and B' in the numeric state, and a mapping from $(A.l \rightsquigarrow B)$ to $\langle A', B' \rangle$ is tracked. Since overapproximating the set of possible paths introduces connectors between unrelated heap cells, the precision of the transitive connectors crucially depends on the accuracy of set $\mathcal{E}$.

Folding and Expanding. Transitive connectors are folded and expanded in the same way as one-step connectors. Figure 6a) shows the tree of Fig. 5d) after expanding the summary two times and then following path r, that is, accessing contents of the root node's right child. The connector of path $R.r \rightsquigarrow C$ has been applied to the numeric content of R and C, forcing the numeric content y of C to be greater than the numeric content x of R. For the sake of presentation, we have omitted connectors originating at S and those targeting R.

Applying Transitive Connectors. In Fig. 6a), the connector for $R.r \rightsquigarrow S$ states that those nodes summarized in S that are reachable via the r field of R have greater numeric content that node R itself. We cannot apply this information directly to summary node S, since at this point the nodes abstracted in S that are reachable from node R via field r cannot be distinguished from those that are reachable from $R.l$. However, we know that $R.r$ definitely points to C, and so all fields reachable via $R.r$ are also reachable via node C, in particular, via fields $C.l$ and $C.r$. More generally, every node A that is reachable from C via paths $C.l \rightsquigarrow A$ or paths $C.r \rightsquigarrow A$ is also reachable via paths $R.r \rightsquigarrow A$. Thus, the information in connector $R.r \rightsquigarrow S$ is valid for $C.l \rightsquigarrow S$ and $C.r \rightsquigarrow S$.

Hence, we restrict the connectors $C.l \rightsquigarrow S$ and $C.r \rightsquigarrow S$ by expanding the connector $R.r \rightsquigarrow S$ as necessary and equating the instance. (Expansion is not necessary in our example, as there is only $C.l \rightsquigarrow S$.) This is done by equating the "source" value r of connector $R.r \rightsquigarrow S$ with x and equating the "target" value s of connector $R.r \rightsquigarrow S$ with the target value $q-1$ of connector $C.l \rightsquigarrow S$. The resulting state is shown in Fig. 6b), where target value t of connector $C.l \rightsquigarrow S$ is now restricted by $t = q-1$ and $t \in \{x+1,\, x+2\}$, maintaining the invariant that the values of the left subtree of C are between the values of R and C.

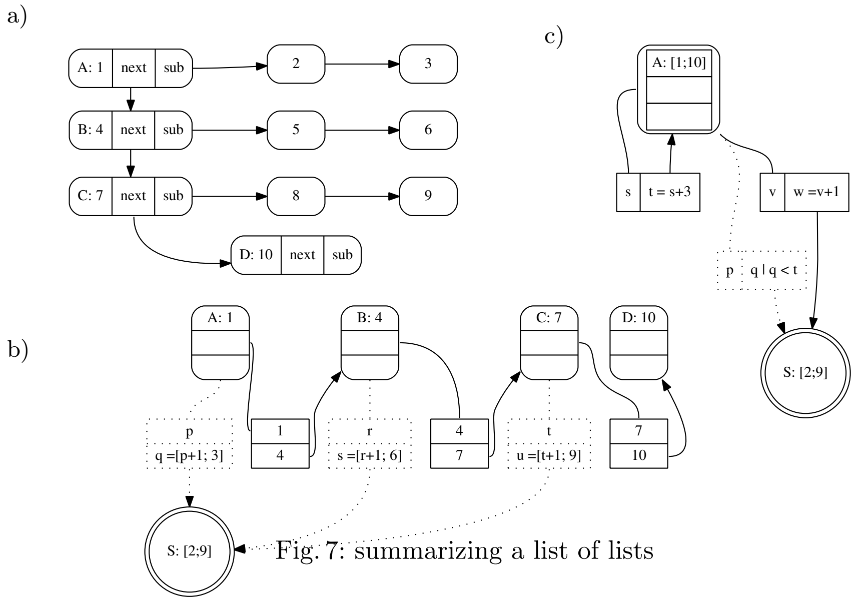

Fig. 7: summarizing a list of lists

5 Inferring Relational Heap Invariants

Since the numeric contents of all connectors are stored in a single domain, connector inference does not only infer relations between the heap cells a connector is attached to, but also between different connectors. This allows to infer invariants for more complex shapes like lists of lists. This section illustrates one such example.

Figure 7a) shows a linked list of sub-lists, each of them containing a numeric field, a pointer to the next list element and a pointer to a sub-list with numeric content. Similar to a skip list, this data structure fulfills the invariant that the entries of each sub-list are between the values of its own and the next list head. Figure 7b) shows the data structure with one-step connectors for the `next` fields and transitive connectors for the `sub` fields of the outer list after the sub-lists are summarized into one node S. For node A, the invariant restricting the entries of its sub-list is expressed by the transitive connector originating in A. Analogously for B, C and D. Summarizing node A with node B *simultaneously* folds node B and the two connectors attached to B onto A and the two connectors attached to A, thereby inferring the relation between the two connectors originating from each node. Summarizing the outer list yields the heap shown in Fig. 7c) where $q < t$ is inferred for the transitive connector $\langle p, q\rangle$ of $A.sub \rightsquigarrow S$ where t is in the one-step connector $A.next \mapsto A$, thereby enforcing the desired sub-list invariant. This example shows that, although being agnostic of the analyzed heap shapes, the combination of connectors and relational numeric domains can even infer properties of hierarchical data structures.

	connectors	time (s)	verified
summarize sorted list	—	0.48	✗
	one-step	0.88	✓
	transitive	1.64	✓
unroll sorted list	—	0.73	✗
	one-step	2.68	✓
	transitive	6.49	✓
summarize numbered list	—	0.52	✗
	one-step	0.86	✓
	transitive	1.70	✗
unroll numbered list	—	0.76	✗
	one-step	2.73	✓
	transitive	6.61	✗
summarize tree	—	0.69	✗
	one-step	2.21	✗
	transitive	4.38	✓
access leaf	—	1.20	✗
	one-step	64,84	✗
	transitive	673,60	✓

Fig. 8: evaluation of our implementation

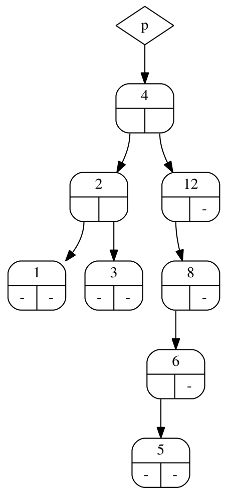

Fig. 9: the search tree

6 Experimental Work

Our prototype implementation consists of the shape analysis described in [17], extended with one-step and transitive connectors. As the numeric domain, we use the implementation of the Octagon domain [11]. While the shape analysis is written in Java, the Octagon domain is taken from the Apron library [8].

Figure 8 shows the experimental results measured on an Intel Core i7 with 1.6 Ghz. All programs are evaluated in three variations: without connectors, with one-step connectors and with transitive connnectors. The first program creates a six-elemented linked list with values in ascending order that is abstracted into one summary node by the analyzer. By inspecting the resulting state, we have verified, whether the sortedness of the list is still maintained by the summarized heap. Without connectors, all relational information between the summarized nodes is lost, whereas in the variants with one-step and transitve connectors, a connector establishes the relation between each node and its successor. The second program accesses and thereby materializes the first three nodes of a summarized sorted list, verifying that they are indeed in ascending order. As before, both one-step and transitive connectors are able to maintain sortedness.

Similarly to the first and second program, the third and fourth program of Fig. 8 summarize and expand a list of consecutive numbers. By inspecting the summarized state and by materializing list elements, we have checked whether the invariant that all list elements are increasing by one is inferred. Here, transitive connectors are not sufficient to verify that the difference between each node and its successor is exactly one, because the transitive connector models the numeric relation between a node and *all* following nodes.

The last two programs of Fig. 8 summarize the binary search tree of Fig. 9. Although the one-step connectors express that the left child of each node has a lower value and, respectively, that the right child of each node has greater value, they cannot express the crucial invariant that all nodes in the right subtree are greater than the root node. The last program descends into the right subtree and checks whether its leftmost leaf has value 5, which amounts to evaluating the C-like assertion `assert(p->r->l->l->l->val == 5)`. Indeed, transitive connectors succeed in proving this property. The timings show a drawback of our implementation: The complexity of the Octagon domain is quadratic in the number of variables, so that the runtime becomes cubic in the number of connectors that have to be applied. Moreover, computing the set of all transitive connectors is costly, since in the best case it is based on the Floyd-Warshall algorithm that runs in n^3 steps when creating the n^2 connectors where n is the number of reachable nodes.

7 Related Work

Ferrara et al. [3] extend a TVLA shape analysis with a numeric abstract domain. They address the challenge of modifying the single numeric domain when the TVLA analysis has inferred several possible heaps at a given program point. Their work thereby shows how TVLA can be combined with our connector inference. However, they use a one-dimensional *fold* and *expand* operation [5] that cannot retain relations between variables in a heap cell. In our context, this is particularly critical, as a connector needs to express relations between at least two variables to be of use. Thus, the use of relational *fold* and *expand* operations [16] is a prerequisite for tracking summarized relations.

Magill et al. [9] enhance a separation logic based shape analysis with a numeric analysis. The idea is to generate an arithmetic program from counterexamples that the shape analysis provides. Thus, the result of the shape analysis is refined whenever the numeric analysis infers that a counterexample is spurious.

Nguyen et al. [12] use predefined numeric predicates in a separation-logic based verification tool. They can verify various various length and sortedness properties, but they cannot infer any numeric relations and rely on user-supplied predicates. Inlining (unfolding) shape definitions creates a new instance of a numeric relation and thereby corresponds to our *expand* operation. Analogously, their fold operation corresponds to our *fold* operation. However, note that a fold operation in separation logic may fail if no predicate matches the current formula. The advantage of using *fold* on numeric domains is that new relations and hence new invariants can be inferred.

Bouajjani et al. [1] introduce *data words abstract domains* that model the numeric contents of linked lists. While their approach is able to infer numeric invariants like sortedness, it is limited to sequential data structures.

McCloskey et al. [10] implement a similar invariant checker based on canonical abstraction [15]. While the set of user-defined predicates is fixed, they infer numeric invariants using numeric abstract domains. Their approach is able to handle complex hierarchical shapes such as lists of trees. Creating hierarchical invariants amounts to summarizing a group of heap nodes onto another group. While the generic *fold* and *expand* operations would, in principle, allow this, our handling of edges between objects that are to be folded currently only caters for two single objects to be folded. Future work will address this shortcoming.

Further afield is the work of Fu [4] that focuses on scalability. Here, an upfront points-to analysis is run to obtain an approximation of possible heap structures. He shows that on this approximated structure a numeric analysis can be implemented efficiently.

Gulwani et al. [6] infer sizes and offsets into memory partitions by combining a generic shape analysis with relational numeric abstract domains. Their approach is based on an on-demand reduction between both domains. Our method of combining shape analysis with numeric analysis merely refines the shape analysis and in particular does not need to use a reduction operator in order to propagate information. Instead, all information is propagated towards the numeric domain. This alleviates the implementor from deciding when to apply the reduction.

Halbwachs et al. [7] verify algorithms that iterate over some arrays A and B simultaneously. They abstract each array A as a current element $A[i]$ at position i and summaries for the partitions before and after position i. Whenever index variable i changes, the summaries are weakly updated *simultaneously* with the current values of $A[i]$ and $B[i]$. In our approach, this simultaneous weak update corresponds to folding a connectors that spans between $A[i]$ and $B[i]$ onto a summarized connector.

Several analyses address the inference of metaproperties such as the length of lists [6, 9, 12]. Our connector inference is a universal refinement technique that also applies to these analyses. For instance, the size of a list can be analyzed by tracking the length of the remaining list in each list element. Analogously, the depth of a tree can be tracked by annotating each node with the distance to its leaves.

8 Conclusion

We proposed an analysis of the numeric contents of heap shapes that is able to infer numeric relations between different but related heap cells. Based on generic relational *fold* and *expand* operations, any numeric abstract domain can be used for this inference. We illustrated that invariants of complex data structures can be inferred automatically and described how established shape analyses like separation logic and TVLA can be enhanced with the proposed connector inference.

References

1. A. Bouajjani, C. Dragoi, C. Enea, A. Rezine, and M. Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. In *Computer Aided Verification*, volume 6174 of *LNCS*, pages 72–88. Springer, 2010.

2. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Constraints among Variables of a Program. In *Principles of Programming Languages*, pages 84–97, Tucson, Arizona, USA, January 1978. ACM.
3. P. Ferrara, R. Fuchs, and U. Juhasz. TVAL+ : TVLA and Value Analyses Together. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Software Engineering and Formal Methods*, volume 7504 of *LNCS*, pages 63–77. Springer, 2012.
4. Z. Fu. Modularly Combining Numeric Abstract Domains with Points-to Analysis, and a Scalable Static Numeric Analyzer for Java. In K. L. McMillan and X. Rival, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 8318 of *LNCS*, pages 282–301. Springer, 2014.
5. D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric Domains with Summarized Dimensions. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, pages 512–529, Barcelona, Spain, March 2004. Springer.
6. S. Gulwani, T. Lev-Ami, and M.Sagiv. A Combination Framework for Tracking Partition Sizes. In *Principles of Programming Languages*, Savannah, Georgia, USA, January 2009. ACM.
7. N. Halbwachs and M. Péron. Discovering Properties about Arrays in simple Programs. In R. Gupta and S. P. Amarasinghe, editors, *Programming Language Design and Implementation*, pages 339–348, Tucson, Arizona, USA, June 2008. ACM.
8. B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, volume 5643 of *LNCS*, pages 661–667, Grenoble, France, June 2009. Springer.
9. S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic Strengthening for Shape Analysis. In H. R. Nielson and G. Filé, editors, *Static Analysis Symposium*, volume 4634 of *LNCS*, pages 419–436. Springer, 2007.
10. B. McCloskey, T. Reps, and M. Sagiv. Statically Inferring Complex Heap, Array, and Numeric Invariants. In *Static Analysis Symposium*, volume 6337 of *LNCS*, pages 71–99, Perpignan, France, 2010. Springer.
11. A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.
12. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated Verification of Shape and Size Properties Via Separation Logic. In B. Cook and A. Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *LNCS*, pages 251–266. Springer, 2007.
13. A. Podelski and T. Wies. Boolean Heaps. In C. Hankin and I. Siveroni, editors, *Static Analysis Symposium*, volume 3672 of *LNCS*, pages 268–283, London, UK, September 2005. Springer.
14. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, 2002. IEEE.
15. M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
16. H. Siegel and A. Simon. Summarized Dimensions Revisited. In L. Mauborgne, editor, *Workshop on Numeric and Symbolic Abstract Domains*, ENTCS, Venice, Italy, September 2011. Springer.
17. H. Siegel and A. Simon. FESA: Fold- and Expand-based Shape Analysis. In *Compiler Construction*, volume 7791 of *LNCS*, pages 82–101, Rome, Italy, March 2013. Springer.