Shape Analysis of Sets

Jan Reineke

Saarland University
Im Stadtwald - Gebäude E1 3
66041 Saarbrücken, Germany
reineke@cs.uni-sb.de

Abstract. Shape Analysis is concerned with determining "shape invariants", i.e. structural properties of the heap, for programs that manipulate pointers and heap-allocated storage. Recently, very precise shape analysis algorithms have been developed that are able to prove the partial correctness of heap-manipulating programs. We explore the use of shape analysis to analyze abstract data types (ADTs). The ADT Set shall serve as an example, as it is widely used and can be found in most of the major data type libraries, like STL, the Java API, or LEDA. We formalize our notion of the ADT Set by algebraic specification. Two prototypical C set implementations are presented, one based on lists, the other on trees. We instantiate a parametric shape analysis framework to generate analyses that are able to prove the compliance of the two implementations to their specification.

1 Introduction

This paper deals with the Shape Analysis of the Abstract Data Type (ADT) Set. Its main goal is to use Shape Analysis to prove that Set implementations written in C comply to an algebraic specification of the ADT Set. The paper summarizes major results from the author's Master's thesis [Rei05].

Shape Analysis [CWZ90,GH96,SRW99,SRW02] is concerned with determining "shape invariants", i.e. structural properties of the heap, for programs that manipulate pointers and heap-allocated storage. Formerly, it was primarily used to aid compilers. Knowledge about the structure of the heap allows to carry out several optimizations, for instance, compile-time garbage collection, better instruction scheduling and automatic parallelization.

Recently, more precise shape analysis algorithms have been developed that are able to prove the partial correctness of heap-manipulating programs. In [LARSW00] bubble-sort and insertion-sort procedures are analyzed. The analyses were able to infer that the procedures indeed returned sorted lists. They also successfully analyzed destructive list reversal and the merging of two sorted lists. The analyses of [LARSW00] and our analyses are based on the Shape Analysis Framework presented in [SRW02]. Logical structures are used to represent the program state in this framework. The concrete semantics is specified in first-order logic. By interpreting the concrete semantics in a 3-valued domain sound and precise abstractions can be extracted automatically.

Set implementations are widely used and can be found in most of the major data type libraries, like STL [MS96], the Java API [Mic04], or LEDA [MN99]. The ADT Set shall serve as an example of abstract data types. The main goal of this paper is to show the partial correctness of set implementations using Shape Analysis. For this purpose we formally define the ADT Set using algebraic specification [EM85,EM90,LEW97]. It shall serve as a reference for the implementations described later. Algebraic Specification allows us to express the intended behaviour independently of possible concrete implementations. The following two axioms are taken from our definition:

$$a \in s.\texttt{insert}(b) \leftrightarrow a =_{el} b \lor a \in s, \ (3)$$
$$a \in s.\texttt{remove}(b) \leftrightarrow a \neq_{el} b \land a \in s \ \ (4)$$

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/698

They capture the effect of the $\cdot$`.insert`$(\cdot)$- and $\cdot$`.remove`$(\cdot)$-functions on the $\in$-predicate. Notice that they do not make any statement about the concrete data structures or algorithms employed.

We present two prototypical C implementations, one based on singly-linked lists, the other on binary trees. Using Shape Analysis, we demonstrate that these implementations comply to our specification of the data type. This involves creating precise analyses using the framework of [SRW02] and linking the results to the specification of the ADT.

2 Sets as Data Abstractions

The formal definition of the ADT Set will serve as a reference for the implementations introduced later. The definition should be independent of possible implementations. Notice that a concrete implementation would also constitute a formal specification. It would however contain many design decisions that are not specific to the data type itself.

A method widely used for the specification of data types is known as *Algebraic Specification of Data Types* [EM85,EM90,LEW97]. Here, a specification consists of a signature and axioms. The signature introduces operations on the data type, while the axioms capture the meaning of the given operations. Data Types defined in this way are often called Abstract Data Types. This is for three reasons:

- The specification is concerned with the data type itself as an abstract mathematical object and not with its implementation by a concrete program in a particular programming language.
- Specifications may be incomplete by only partially specifying the meaning of operations.
- They maybe defined in terms of other data types that serve as parameters. This is also called generic specification.

While we easily grasp an intuitive meaning of these specifications, it is of course profitable to give a formalization of the concept. We will not go into detail about this since we do not rely on the precise definitions in the following chapters. The semantics of such a specification is a set of many-sorted algebras. An algebra belongs to this set if it is a model of the axioms of the specification. The axioms are implicitly universally quantified. Usually, there are many non-isomorphic models of a given specification reflecting the incompleteness of the definition. The interested reader may consult [EM85] and [LEW97] for an in-depth treatment of the topic.

The full specification of the ADT Set is displayed in Table 1. Our specification is parameterized by an *element* type. This could also be instantiated with a *set* itself, building sets of sets of some primitive type, and so on. We are assuming an existing specification of the natural numbers *nat*.

The empty set is provided as a constant. Other sets can be constructed by inserting and removing elements using `.insert`$(\cdot)$ and `.remove`$(\cdot)$. The `.selectAndRemove` function returns an element and removes it from the set. It can be used to iterate over a set. The `.sizeOf` function returns the cardinality of the set as a natural number. The $\in$ predicate allows to test set membership. $\subseteq$ and $=$ correspond to subset and equality of sets.

Most of the axioms are straightforward. We distinguish equality on sets $=$, equality on elements $=_{el}$, and equality on natural numbers $=_{nat}$. Axiom (1) assures that every possible set can be constructed by applications of $\varnothing$ and `.insert`. In axiom (5) we only have an implication because the `.selectAndRemove` function chooses an element nondeterministically. Axioms (6) and (7) correspond to the extensionality axiom of set theory. Axioms (8)-(13) deal with the cardinality of sets. The axioms are complete in the sense that the meaning of arbitrary formulae over the given alphabet (the functions and predicates of the ADT specification) can be derived.

set =

begin generic specification

$$
\begin{array}{llclcl}
\textbf{parameter} & \text{element} & & & & \\
\textbf{using} & \text{nat} & & & & \\
\textbf{sorts} & \text{set} & & & & \\
\textbf{constants} & \varnothing & : & \text{set} & & \\
\textbf{functions} & \cdot\texttt{.insert}(\cdot) & : & \text{set} \times \text{element} & \rightarrow & \text{set} \\
 & \cdot\texttt{.remove}(\cdot) & : & \text{set} \times \text{element} & \rightarrow & \text{set} \\
 & \cdot\texttt{.selectAndRemove} & : & \text{set} & \rightharpoonup & \text{element} \times \text{set} \\
 & \cdot\texttt{.sizeOf} & : & \text{set} & \rightarrow & \text{nat} \\
\textbf{predicates} & \cdot \in \cdot & : & \text{element} \times \text{set} & & \\
 & \cdot \subseteq \cdot & : & \text{set} \times \text{set} & & \\
 & \cdot = \cdot & : & \text{set} \times \text{set} & & \\
\textbf{variables} & s, s' & : & \text{set} & & \\
 & a, b & : & \text{element} & &
\end{array}
$$

$$
\begin{array}{lr}
\textbf{axioms}\ \text{set}\ \textbf{generated by}\ \varnothing, \texttt{.insert}; & (1) \\
\neg(a \in \varnothing), & (2) \\
a \in s\texttt{.insert}(b) \leftrightarrow a =_{el} b \vee a \in s, & (3) \\
a \in s\texttt{.remove}(b) \leftrightarrow a \neq_{el} b \wedge a \in s, & (4) \\
(a, s') = s\texttt{.selectAndRemove} \rightarrow a \in s \wedge a \notin s' \wedge s'\texttt{.insert}(a) = s, & (5) \\
s \subseteq s' \leftrightarrow a \in s \rightarrow a \in s', & (6) \\
s = s' \leftrightarrow s \subseteq s' \wedge s' \subseteq s, & (7) \\
\varnothing\texttt{.sizeOf} =_{nat} 0, & (8) \\
s\texttt{.insert}(b)\texttt{.sizeOf} =_{nat} s\texttt{.sizeOf} \leftrightarrow b \in s, & (9) \\
s\texttt{.insert}(b)\texttt{.sizeOf} =_{nat} s\texttt{.sizeOf} + 1 \leftrightarrow \neg(b \in s), & (10) \\
s\texttt{.remove}(b)\texttt{.sizeOf} =_{nat} s\texttt{.sizeOf} \leftrightarrow \neg(b \in s), & (11) \\
s\texttt{.remove}(b)\texttt{.sizeOf} =_{nat} s\texttt{.sizeOf} - 1 \leftrightarrow b \in s, & (12) \\
(a, s') = s\texttt{.selectAndRemove} \rightarrow s'\texttt{.sizeOf} =_{nat} s\texttt{.sizeOf} - 1. & (13)
\end{array}
$$

end generic specification

Table 1. ADT Set

3 Shape Analysis of Implementations

In this section we analyze two prototypical C implementations of the ADT Set. One implementation is based on singly-linked lists, the other on binary trees. After briefly introducing parts of the two implementations, we proceed to describe our analyses. The main goal of the analyses is to prove that the implementations comply with the ADT specification given in Chapter 2. The implementations each contain the two methods, `insertElement`, `removeElement` and the function `isElement`. They implement the $\cdot$`.insert(`$\cdot$`)`, $\cdot$`.remove(`$\cdot$`)` functions and the $\cdot \in \cdot$ predicate, respectively. We chose to show the following two axioms, since they capture the most important aspects of the ADT Set:

$$a \in s.\texttt{insert}(b) \leftrightarrow a =_{el} b \vee a \in s, \ (3)$$
$$a \in s.\texttt{remove}(b) \leftrightarrow a \neq_{el} b \wedge a \in s \ \ (4)$$

Our analyses are conducted using TVLA [LAS00] and are based on previous analyses on lists and trees contained in the TVLA 2 distribution.

3.1 List-based Implementation

```
typedef struct List
{
  void* data;
  struct List* next;
} List;

typedef struct Set
{
  List* list;
  int (*compare)(void*, void*);
  int size;
} Set;
```

(a)

```
int isElement(Set* set, void* element)
{
  List* list = set->list;

  while (list != 0)
  {
    if (compare(list->data, element) == 0)
      return 1;

    list = list->next;
  }

  return 0;
}
```

(b)

Fig. 1. C structure declarations for Lists and Sets and C source of membership test

Our first set implementation uses singly-linked lists to store the elements. It also maintains the size of the current set. The structure declarations are visible in Figure 1. When allocating such a set, a compare-function has to be given, that establishes an equivalence relation on the data elements.

Figure 1 also shows the code for testing set membership. The method simply iterates over the list, comparing each item with the element that is tested for set membership.

Figure 2 shows the implementations of the insertion and removal methods. The insertion method iterates over the list until it either finds the element or reaches the final element of the list, indicated by a null-pointer in the next-field. If the element was not found it is appended at the end. Removal works similarly. When the element is found, it is decoupled from the list and the memory is freed.

Data Structure Invariants Our analyses rely on a number of data structure invariants at entrance to the methods. Showing their maintenance is part of the proof. By data structure invariants we mean invariants that are related directly to the concrete data structure employed to implement the ADT Set. In this case properties of singly-linked lists:

- The list is acyclic
- The list does not contain any duplicate elements

We use instrumentation predicates to capture these properties formally using first-order logic.

3.2 Tree-based Implementation

As in the list-based case, a compare-function is needed. This time it has to implement a reflexive total order. This is necessary, to build an ordered tree. Figure 3 shows the structure declarations. Every node in the tree stores one of the set elements and maintains pointers to two children nodes *left* and *right*.

Figure 3 also contains the source of the set membership test. The method simply traverses the tree until it either finds the element or reaches a leaf node. The source of the insertion and removal methods on trees can be found in the appendix, since it is too large to be dealt with here. We restrict ourselves to mentioning the main ideas of the two algorithms. New elements are always inserted as new leaf nodes, by traversing the tree to the correct position. While insertion of elements if fairly easy and quite similar to its list pendant, removal of elements is a non-trivial task. Figure 4 illustrates this. Removing elements that are stored in leaf nodes is simple (left). They can simply be decoupled from their respective parent nodes. If the node has one child, we can connect this child at the place of the node to its former parent node (middle). The most complicated case arises when the particular node has two child nodes (right). In this case, we have to find another node in the tree to replace the element node. This node has to be smaller than all nodes on the right and greater than all nodes on the left. There are two ways to find such an element. Either one can take the right-most element of the left subtree or the left-most element of the right subtree. We chose to always take the right-most element of the left subtree. In addition, there are some special cases of the latter case. For instance, if the root of the left subtree is already the right-most element of the left subtree.

```
void insertElement(Set* set, void* element)
{
  List* list = set->list;
  List* prev = 0;

  while (list != 0)
  {
    if (compare(list->data, element) == 0)
        return;

    prev = list;
    list = list->next;
  }

  List* newList = (List*)malloc(sizeof(List));
  newList->data = element;
  newList->next = 0;
  set->size++;

  if (prev == 0) //list is empty
  {
    set->list = newList;
  }
  else //append item to list
  {
    prev->next = newList;
  }
}
```

(a)

```
void* removeElement(Set* set, void* element)
{
  List* temp;
  List* list = set->list;

  if (list == 0)
    return;

  if (compare(list->data, element) == 0)
  {
    set->size--;
    set->list = list->next;
    free(list);
  }
  else
    while (list->next != 0)
    {
      if (compare(list->next->data, element) == 0)
      {
        void* deletedElement = list->next->data;
        set->size--;
        temp = list->next->next;
        free(list->next);
        list->next = temp;
        return deletedElement;
      }
      list = list->next;
    }
}
```

(b)

Fig. 2. C source of Insertion and Removal methods

```
typedef struct Tree
{
  void* data;
  struct Tree* left;
  struct Tree* right;
} Tree;

typedef struct Set
{
  Tree* tree;
  int (*compare)(void*, void*);
  int size;
} Set;
```

(a)

```
int isElement(Set* set, void* element)
{
  Tree* tree = set->tree;

  while (tree != 0)
  {
    if (compare(tree->data, element) == 0)
      return 1;
    else if (compare(tree->data, element) < 0)
      tree = tree->left;
    else
      tree = tree->right;
  }

  return 0;
}
```

(b)

Fig. 3. C structure declarations for Trees and Sets and C source of isElement test

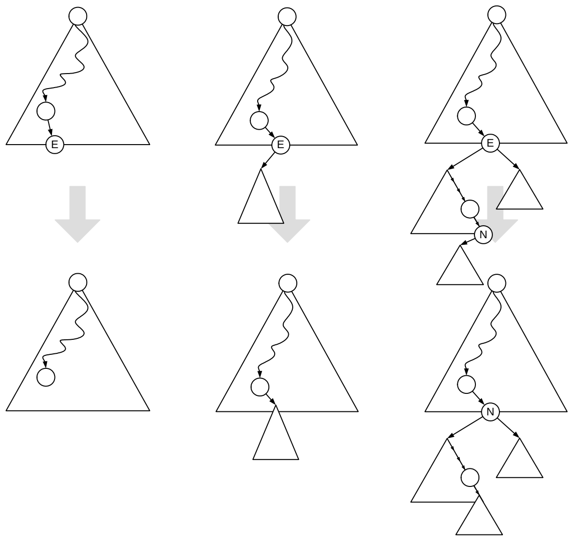


Fig. 4. Removal from Ordered Tree

Data Structure Invariants In order to prove our ADT Set axioms we need to maintain two data structure invariants:

- The structure representing the set is a tree
  Out of many equivalent definitions for “binary treeness”, we chose the following: Whenever an element is reachable from the left child of a node in the structure, then it is not reachable from the right child, and vice versa.
- The tree is ordered
  Every element reachable from the left child is smaller and every element reachable from the right child is greater. This implies that the tree does not contain duplicate elements. It also implies the first data structure invariant. It is still useful to consider the first invariant, because it may help in proving this one.

Again, we used instrumentation predicates to formalize the two invariants using first-order logic. Proving the latter proved to be quite difficult. It is a global property, i.e. it does relate elements in the tree that are not directly connected. We will go into more detail about this in the analysis section.

3.3 Shape Analysis

To prove the ADT Set axioms we perform three analyses for each implementation. The analyses of the insertion methods prove the following:

$$\mathit{isElement}(a, s.\texttt{insertElement}(b)) \leftrightarrow a =_{el} b \vee \mathit{isElement}(a, s)$$

Notice the difference compared with the corresponding axiom (3). The instrumentation predicate *isElement* replaces the $\cdot \in \cdot$ predicate. That is we prove the property of the insertion method in terms of an instrumentation predicate. The same holds for the removal methods and axiom (4). There, we prove:

$$\mathit{isElement}(a, s.\texttt{removeElement}(b)) \leftrightarrow a \neq_{el} b \wedge \mathit{isElement}(a, s)$$

To conclude the proofs we show that the `isElement` functions in both implementations are equivalent to the instrumentation predicate *isElement*:

$$\mathit{isElement}(a, s) \leftrightarrow s.\texttt{isElement}(a)$$

Combining this equivalence with the two preceding proofs yields:

$$s.\texttt{insertElement}(b).\texttt{isElement}(a) \leftrightarrow a =_{el} b \vee s.\texttt{isElement}(a)$$
$$s.\texttt{removeElement}(b).\texttt{isElement}(a) \leftrightarrow a \neq_{el} b \wedge s.\texttt{isElement}(a)$$

These two equivalences correspond directly to axioms (3) and (4).

Shape Analysis of List-based Implementation Our analysis is based on existing analyses on lists and trees. We borrowed the concrete semantics of most of the statements from these. The following table shows how we represent the state by logical predicates.

Predicate	**Intended Meaning**
$x(v)$ for each $x \in \mathit{Var}$	Pointer variable x points to heap cell v.
$n(v_1, v_2)$	The *next* selector of v_1 points to v_2.
$\mathit{deq}(v_1, v_2)$	The *data*-fields of v_1 and v_2 are equal.
$\mathit{isSet}(v)$	v represents a set.
$\mathit{or}[n, x](v)$ for each $x \in \mathit{Var}$	v was reachable from x via *next*-fields.

As depicted, pointer variables are represented by unary predicates. The *next*-field is modeled by a binary predicate. Since we can only model the structure of the heap by these predicates, primitive values have to be dealt with differently. Abstracting from the concrete values of the *data*-fields, we capture the equivalence relation between *data*-fields by the binary predicate *deq*. This corresponds to the compare-function needed in the implementation. To differentiate between set locations and other locations in the heap, the *isSet* predicate is used. To be able to relate elements contained in the list before the execution of one of our procedures with their output structures, we mark elements reachable from x via *next*-fields using the $\mathit{or}[n, x]$ predicate.

While the above core predicates suffice to define the concrete semantics of all the statements, we need additional instrumentation predicates to gain precision.

Predicate	**Defining Formula**	**Intended Meaning**
$is[n](v)$	$\exists v_1, v_2.(v_1 \neq v_2 \wedge n(v_1, v) \wedge n(v_2, v))$	v is shared.
$c[n](v)$	$\exists v_1.(n(v_1, v) \wedge n^*(v_1, v_2))$	v resides on a cycle.
$t[n](v_1, v_2)$	$n^*(v_1, v_2)$	Transitive reflexive closure of *next*.
$r[n, x](v)$ for each $x \in$ *Var*	$\exists v_1.(x(v_1) \wedge t[n][v_1, v))$	v is reachable from x via *next*-fields.
$noeq[deq, n](v)$	$\forall v_1.(((t[n](v_1, v) \vee t[n](v, v_1)) \wedge v_1 \neq v) \rightarrow (\neg deq(v_1, v) \wedge \neg deq(v, v_1)))$	The *data*-field of v is different from the *data*-fields of locations that can reach v and that are reachable from v.
$validSet(v)$	$isSet(v) \wedge noeq[deq, n](v)$	v represents a valid set (no duplicate entries).
$isElement(v_1, v_2)$	$isSet(v_2) \wedge \exists v.(t[n](v_2, v) \wedge deq(v_1, v) \wedge v \neq v_2)$	v_1 is an element of set v_2.

The first four of these instrumentation predicates capture general properties of the shape of the heap. They have been used in previous analyses of list-manipulating programs. $c[n]$ covers the acyclicity data structure invariant mentioned in the implementation section.

The $noeq[deq, n]$ predicate is tailored specifically to the current task. It expresses that no two elements in the list have equal *data*-fields. The definition comprises both directions, i.e. both elements reachable from v and elements from which v is reachable. This actually makes it easier to reestablish the property when manipulating the list. It is a formalization of the second data structure invariant for lists. *validSet* does not help to increase precision. It only increases the readability of the output structures.

To capture our notion of set membership we define the *isElement*-predicate. v_1 is an element of set v_2 if its *data*-field is equal to one of the nodes reachable from v_2. Our analysis shows that the effect of the insertion and removal methods on set membership, expressed by *isElement* conforms to the ADT Set axioms.

Our input structures cover all possible lists representing sets pointed to by *set*. *element* points to the element that shall be inserted into the set. Figure 5 displays these structures. In (a) *set* is empty. In (b) *set* is non-empty and set membership of *element* is unknown, *isElement*'s value is indefinite for the nodes pointed to by *element* and *set*.

Insertion Running the analysis for insertion yields three output structures that are shown in Figure 6. All of the resulting structures fulfill the data structure invariants, i.e. $noeq[deq, n]$ is true for the set and $c[n]$ is false everywhere. Also, *isElement* is true for the nodes pointed to by *element* and *set*. In addition, the $or[n, set]$-predicate indicates that elements which were formerly reachable from *set* are still reachable after the execution of *setInsert*.

Looking at the structures one can identify the different cases that the insertion method has to deal with. Structure (a) corresponds to the empty set as input structure. In structure (b) a new element had to be appended to the list, because the *data*-field of *element* is not equal to any of the original elements of the list (the *deq* predicate is false). In structure (c) *element* was already contained in the list, indicated by the *isElement*-predicate.

Removal When translating the C code into a Control Flow Graph in TVLA, we omitted the deallocation of the element in the list. This is only for illustration purposes.

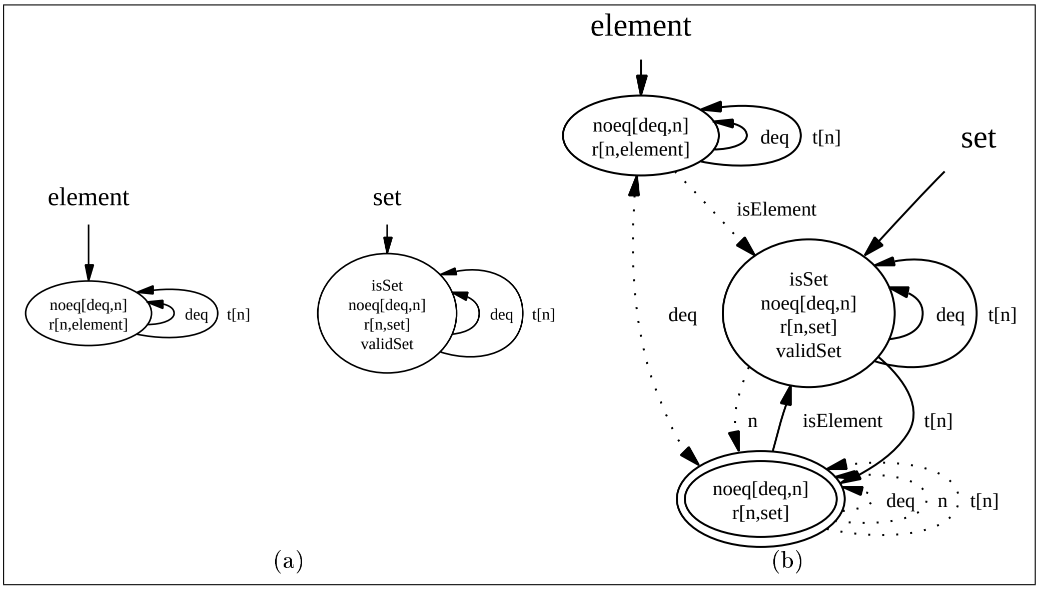


Fig. 5. Input Structures for List-based Insertion and Removal

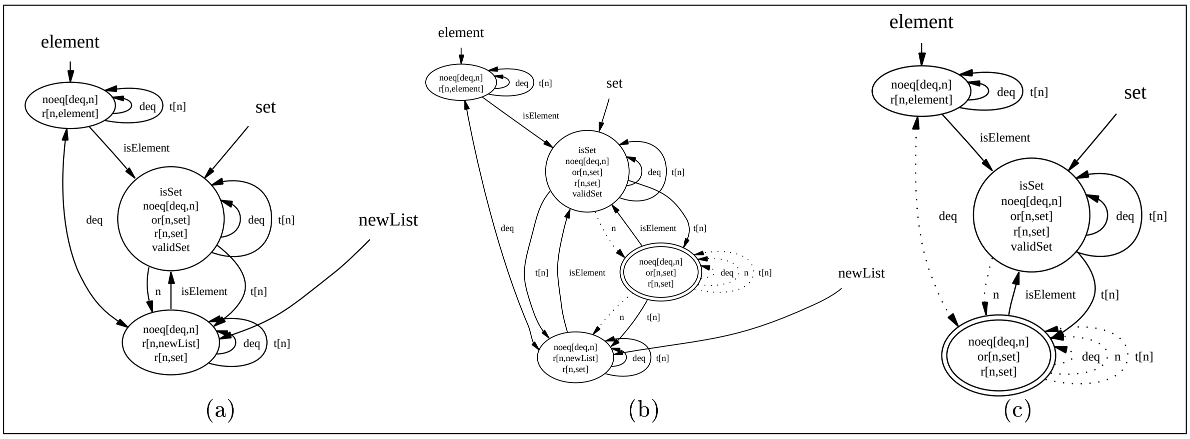


Fig. 6. Output Structures for List-based Insertion

Running *setRemove* results in four output structures displayed in Figure 7. Again, the maintenance of the data structure invariants is proven: $noeq[deq, n]$ is true and $c[n]$ is false everywhere. The element has indeed been removed from the list. This can be observed by the *isElement*-predicate. Other elements of the set are still contained, as indicated by the $or[n, set]$-predicate.

Structures (a) and (c) correspond to the case where *element* was not contained in the set before. The two other structures (a) and (d) reflect the case where *element* was indeed part of the set. The abstraction also distinguishes between empty (c and d) and non-empty sets (a and b).

Membership Test We omit to display the output structures of this analysis, since the routine is not manipulating the heap at all. The analysis checked that our *isElement* function returns true if and only if the *isElement*-predicate holds. This is done by separating the structures into those that reach a point where true is returned and those structures that reach a point where false is returned. By this, we establish a connection between the different analyses. The two other analyses

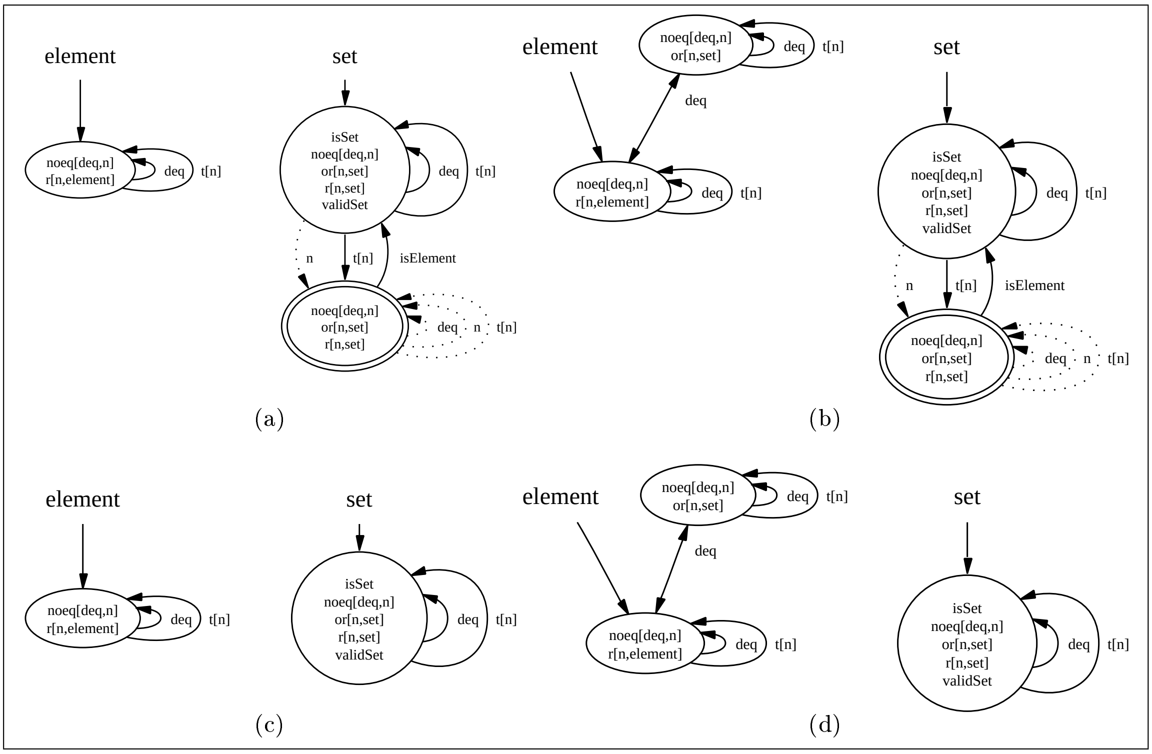


Fig. 7. Output Structures for List-based Removal

on list insertion and removal only proved correctness in terms of the *isElement*-predicate. The current analysis shows that this was just.

Shape Analysis of Tree-based Implementation The domain is represented in a similar way as in the list-based case. Instead of having a *next*-predicate, *left*- and *right*-predicates are used to model the *left*- and *right*-fields in the tree. The *left*-predicate is also used to model the *tree*-field in the set structure to minimize the number of predicates. The *tree*-field only occurs at most once in all of the structures.

Predicate	**Intended Meaning**
$x(v)$ for each $x \in Var$ $sel(v_1, v_2)$ for each $sel \in \{left, right\}$ $dle(v_1, v_2)$	Pointer variable x points to heap cell v. The $left$ ($right$) selector of v_1 points to v_2. v_1->$data \leq v_2$->$data$.
$or[x](v)$ for each $x \in Var$ $isSet(v)$	v was reachable from x via *left*- and *right*-fields. v represents a set.

As noted in the implementation section, an ordering relation is needed here. It is modeled by the *dle*-predicate, which is assumed to be reflexive and transitive during the analysis. $or[x]$ and *isSet* have the same meaning as before.

While the core predicates used to model the domain were very similar to the list-based case, the choice of instrumentation predicates was quite different. We separate them into two parts. One is solely concerned with the structure of the trees. The other also deals with ordering.

Predicate	**Defining Formula**	**Intended Meaning**
$\mathit{down}(v_1, v_2)$	$\mathit{left}(v_1, v_2) \lor \mathit{right}(v_1, v_2)$	The union of the two selector predicates left and right.
$\mathit{downStar}(v_1, v_2)$	$\mathit{down}^*(v_1, v_2)$	Records reachability between tree nodes.
$\mathit{downStar}[\mathit{sel}](v_1, v_2)$ for each $\mathit{sel} \in \{\mathit{left}, \mathit{right}\}$	$\exists v.(\mathit{sel}(v_1, v) \land \mathit{down}^*(v, v_2))$	Remembers the first selector needed to reach v_2 from v_1.
$r[x](v)$ for each $x \in \mathit{Var}$	$\exists v_1.(x(v_1) \land \mathit{downStar}(v_1, v))$	v is transitively reachable from x.
$\mathit{treeNess}$	$\forall v_1, v_2, v.((\mathit{downStar}[\mathit{left}](v, v_1) \land \mathit{downStar}[\mathit{right}](v, v_2)) \Rightarrow (\neg \mathit{downStar}(v_1, v_2) \land \neg \mathit{downStar}(v_2, v_1)))$	The heap consists of trees.

The two $\mathit{downStar}[\mathit{sel}]$-predicates record reachability between tree-nodes, where the first selector on the path is sel. In ordered trees this determines the relation between the elements in the tree. To be able to check whether the ordering is maintained, it is important to keep this relation precise for elements that are manipulated. $\mathit{treeNess}$ records the first data structure invariant mentioned in the implementation section. We decided to make $\mathit{treeNess}$ a global nullary predicate to reduce the size of the domain. There is a drawback to this approach however. It is nearly impossible to reestablish the property once it is violated, because we lose information about parts of the heap that still satisfy the property. A unary $\mathit{treeNess}$ predicate would be able to capture local violations and make it easier to reestablish the property after it was temporarily destroyed. The methods that we checked maintain $\mathit{treeNess}$ in the entire heap permanently allowing to use the nullary predicate.

Predicate	**Defining Formula**	**Intended Meaning**
$\mathit{dle}[x, \mathit{left}](v)$ for each $x \in \mathit{Var}$	$\exists v_1.(x(v_1) \land \mathit{dle}(v, v_1) \land \neg \mathit{dle}(v_1, v))$	The data-field of v is less than the data-field of v_1, where v_1 is pointed to by x.
$\mathit{dle}[x, \mathit{right}](v)$ for each $x \in \mathit{Var}$	$\exists v_1.(x(v_1) \land \neg \mathit{dle}(v, v_1) \land \mathit{dle}(v_1, v))$	The data-field of v is greater than the data-field of v_1, where v_1 is pointed to by x.
$\mathit{inOrder}[\mathit{dle}]$	$\forall v_2, v_4.(\mathit{downStar}[\mathit{left}](v_2, v_4) \Rightarrow (\mathit{dle}(v_4, v_2) \land \neg \mathit{dle}(v_2, v_4))) \land \forall v_2, v_4.(\mathit{downStar}[\mathit{right}](v_2, v_4) \Rightarrow (\neg \mathit{dle}(v_4, v_2) \land \mathit{dle}(v_2, v_4)))$	All the trees in the heap are in order.
$\mathit{isElement}(v_1, v_2)$	$\mathit{isSet}(v_2) \land \exists v_{equal}.(\mathit{downStar}(v_2, v_{equal}) \land \mathit{dle}(v_{equal}, v_1) \land \mathit{dle}(v_1, v_{equal}) \land v_{equal} \neq v_2$	v_1 is an element of set v_2.

The $\mathit{dle}[x, \mathit{sel}]$ captures the relation between the node pointed to by x and other heap nodes. These predicates are used to partition the heap into elements less than the node pointed to by x and those that are greater. Being unary predicates they can be used as abstraction predicates. This could be called a “pseudo-binary abstraction”, since parts of the binary predicate dle are taken to form several unary predicates.

$\mathit{inOrder}[\mathit{dle}]$ formalizes the second data structure invariant for ordered trees. It requires elements in the left subtree of a node to be smaller and elements in the right subtree to be greater than the node itself. Smaller and greater are expressed in terms of dle.

The set membership property $\mathit{isElement}$ is formalized similarly to the list-based case. v_1 is an element of set v_2 if its data-field is equal to one of the nodes reachable from v_2, where equal can be formulated in terms of *dle*.

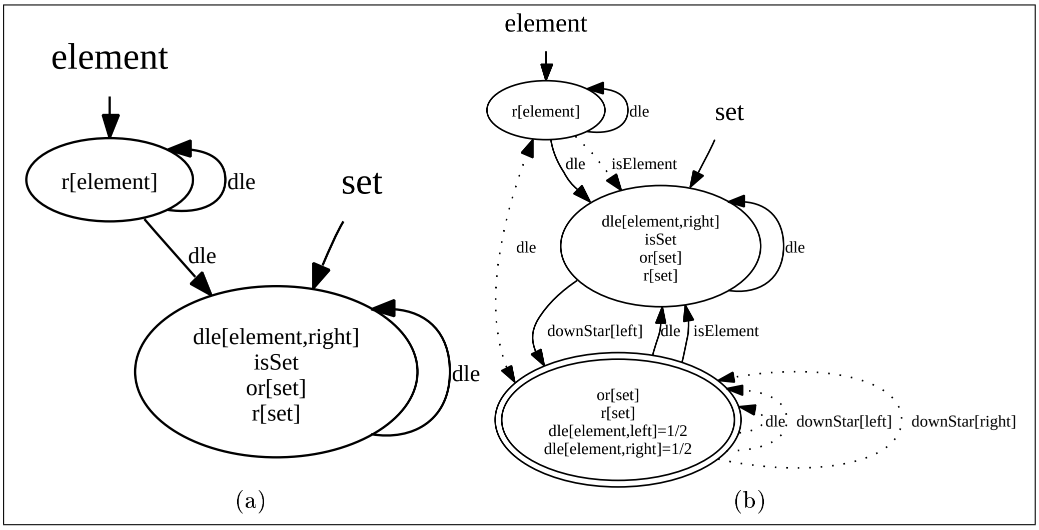
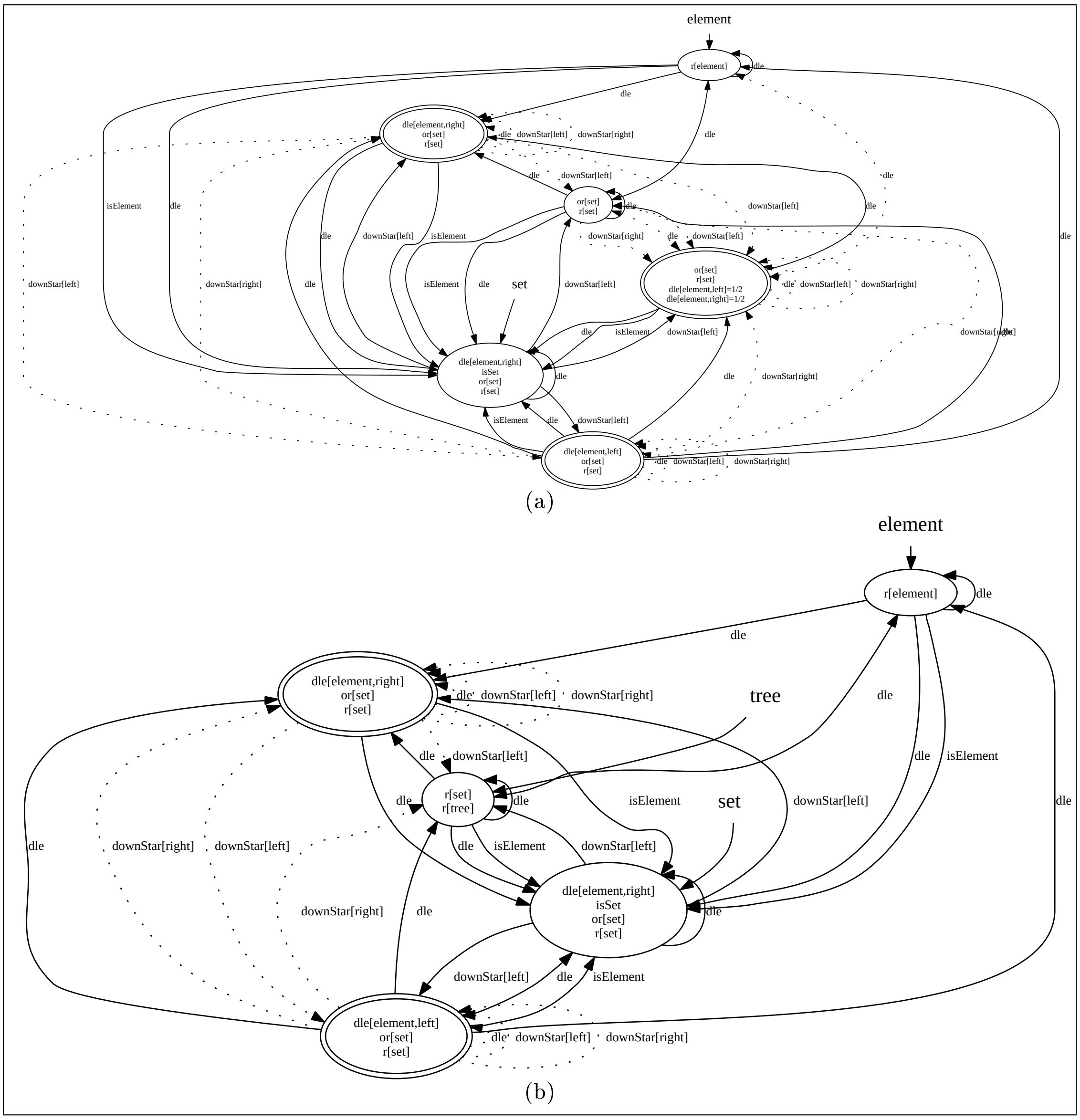


Fig. 8. Input Structures for Tree-based Insertion and Removal

Figure 8 displays the input structures for our analysis of the insertion and removal methods. In the following we omitted several predicates to make the visualizations more readable. The predicates that we left our were $left, right, down, downStar$. Again, we want to cover all possible sets by these abstract structures. In structure (a) *set* is empty and thus *element* is not an element of *set*. Structure (b) represents non-empty sets. *element* might be part of the set, indicated by the dotted *isElement*-predicate and the dotted *dle*-predicate between *element* and the contents of *set*. We also had to assign a value to the *dle*-predicate for *set* which does not have a *data*-field. Its *data*-field is assumed to be greater than all other *data*-fields. Elements that were originally reachable from *set* are marked with $or[set]$ as in the list-based case.

Insertion Running the analysis for set insertion yields 21 structures at exit. Most of them concern special cases where the element had to be inserted in the left- or right-most position of the tree or where the left or the right subtree of the root was empty. All resulting structures fulfilled the data structure invariants and *element* had been inserted into *set*. We picked two structures that represent the most general cases. They can be seen in Figure 9.

Due to the number of binary predicates involved in the analysis the output structures are hard to read. Also, the visualization engine does not know our intuition behind the different predicates, which could help to generate more readable output. In structure (a) the algorithm found a node in the tree that is equal to *element*. The three summary nodes make up the rest of the tree. The summary node to the right represents the subtree of the node that was found. The other two summary nodes partition the parents and neighbors into those that have a smaller *data*-field and those that have a greater *data*-field. For this particular case the partitioning of the set is not important. For structure (b) however it is the key to proving that the ordering is preserved. Here, no node in the tree was found that was equal to *element*. Therefore a new heap node was allocated and inserted into the tree, preserving the ordering. This is were the partition into smaller and larger elements becomes important. Nodes that are greater than the new node can only reach it via a path that starts by going left: $downStar[left]$ is indefinite and $downStar[right]$ is false.



(a)

(b)

Fig. 9. Sample Output Structures for Tree-based Insertion

Nodes with a smaller *data*-field can in turn only reach it via a path that starts with a *right*-edge ($downStar[right] = 1/2$ and $downStar[left] = 0$).

Removal As noticed in the implementation section, tree-based removal was the most complicated routine that we analyzed. Its size and complexity led to very time-consuming analyses that did not allow a trial and error approach when choosing the abstraction predicates. We used the same predicates as in the analysis of the insertion algorithm. They were developed for this method though and proved to work for the simpler insertion routine, too.

Proving that *element* is not a member of *set* after the analysis was simple, once the data structure invariants could be established. The ordering property ensures that every element only occurs once in the tree. Showing that the ordering data structure invariant was maintained was more difficult. The key predicates involved in proving this were $dle[x, sel]$ and $downStar[sel]$. The use of these predicates in the insertion routine already hints at why they are useful for removal. Figure 4 illustrates the different possibilities when removing an element from the tree. As the algorithm keeps track of the relevant nodes (those represented by circles in the figure) in the graph through pointer variables, $dle[x, sel]$ delivers the necessary partition to keep relevant ordering information. In addition $downStar[sel]$ captures the important first selectors on paths between these parts of the tree.

To cope with the long analysis times we decomposed the problem into smaller ones first:

- Finding the element to delete.
- The element has one or no children.
- The element has two children, the most difficult case.

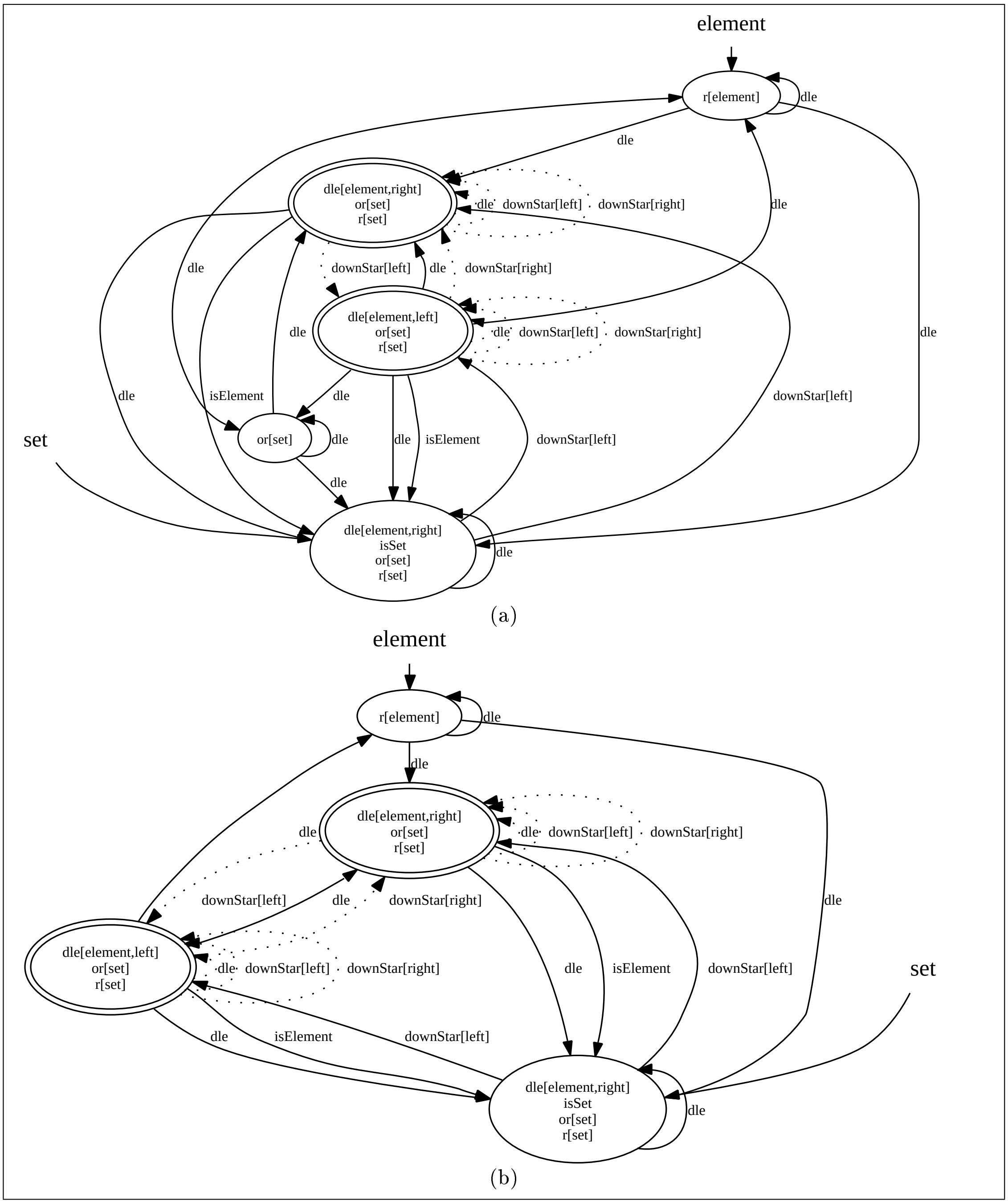
In the end we put everything together.

Again, we decided to present only two representative output structures out of overall eight. They are shown in Figure 10. Both structures satisfy the two data structure invariants modeled by $inOrder[dle]$ and $treeNess$. In structure (a) *element* was contained in *set* and therefore removed from it. For demonstration purposes we did not free the element taken from the tree. One can see that the tree has been partitioned into nodes with a greater *data*-field and nodes with a smaller *data*-field than *element*. The same holds for structure (b). In this case *element* was not contained in *set* at the invocation of the routine. No node was removed from the tree.

Membership Test Again, we omit to display the output structures. It is quite obvious that the analysis succeeds, because the tree traversal analyzed is part of the insertion and removal methods as well, which were analyzed before.

Empirical Results Table 2 presents some data about the four analyses. The analysis of the insertion, removal and membership test methods of our list-based implementation resulted in a similar number of structures and relatively short analysis times. In the tree-based case, however, the difference was considerable. This can probably be explained with the higher number of unary predicates in the removal analysis, which led to more structures per location. The worst-case complexity of the analysis is doubly-exponential in the number of abstraction predicates. Additionally, the control flow graph (see Figure 11) for removal contains more than three times as many locations as the CFG for insertion.

Discussion We managed to show interesting properties of list- and tree-based set implementations. Our analyses assumes data structure invariants specific to the respective implementation to hold at the entrance. The maintenance of these invariants throughout the execution of the routines



(a)

(b)

Fig. 10. Sample Output Structures for Tree-based Removal

Analysis	**#locations in CFG**	**#unary predicates**	**#binary predicates**	**#structures**	**average #structs per location**	**maximal #structs per location**	**time**
Membership, List-based	9	20	5	28	3	6	2.570s
Insertion, List-based	19	29	5	81	4	11	2.720s
Removal, List-based	22	29	5	124	5	11	4.050s
Membership, Tree-based	10	18	11	84	8	19	32.84s
Insertion, Tree-based	25	24	11	536	21	91	69.23s
Removal, Tree-based	76	42	11	27697	364	3132	21767s

Table 2. Empirical Results

is established. Using these invariants our analysis was able to prove that the effect of the insertion and removal methods complies with axioms of the ADT Set. The nature of the shape analysis framework limited our proofs to partial correctness.

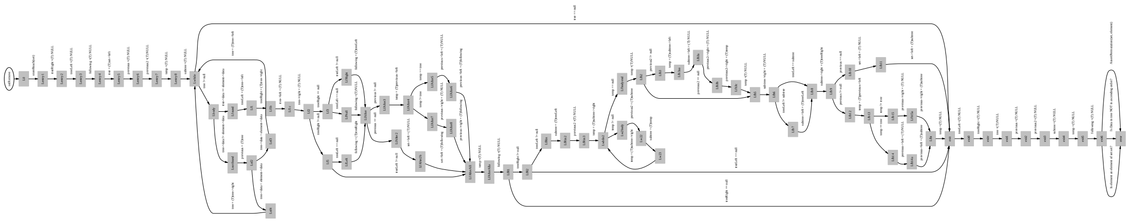


Fig. 11. CFG for Tree Removal

We used the *isElement*-predicate to relate different analyses. While the insertion and removal methods were proved correct in terms of *isElement*, the analysis of the set membership routine showed the equivalence of this routine with *isElement*. This approach loosely corresponds to the abstraction mechanism used in [LKR04]. They use sets to abstract from more complex data structures, which limits them to statically allocated data structures. Our use of *isElement* on the other hand allows to handle dynamically allocated sets.

Choosing the right instrumentation predicates required a thorough understanding of the data structures involved. For trees this meant identifying that reachability alone is not very interesting, but that the first edge on a path from one node to another is important. However, the predicates are not tailored to specific algorithms, but to the underlying data structures. They might prove useful for other algorithms on trees and lists as well.

Abstraction Expressions The need to partition the trees into smaller and larger elements led to the introduction of the $dle[x, sel]$-predicate family. The effect of these unary predicates on the abstraction could also be achieved by using the binary dle-predicate in the abstraction process.

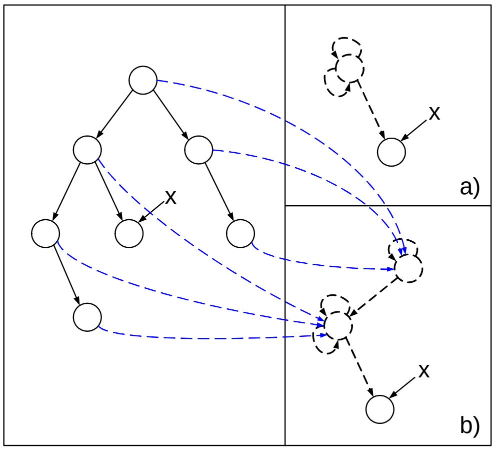


Fig. 12. Abstraction Expressions

Here, individuals should only be joined if they have the same canonical name and if they agree on binary abstraction predicates to other canonical names. This is illustrated in Figure 12. The tree on the left is supposed to be in order. The ordering predicate is not visualized to make it more readable. Canonical Abstraction would collapse all the nodes not pointed to by x (a). The relation between the resulting summary node and the node pointed to by x would be indefinite. Additionally abstracting from *dle* would instead create two summary nodes and keep ordering information definite. Of course, the proposed abstraction can also be achieved using a number of unary abstraction predicates. The number of predicates needed for this is linear in the number of abstraction predicates though, to cover all canonical names.

We propose to specify the abstraction through *Abstraction Expressions*:

Definition 1 (Syntax of Abstraction Expressions). *The set of* Abstraction Expressions *over a set of unary predicates U and a set of binary predicates B is defined inductively as follows:*

- $\{u_1, \ldots, u_n\}$ *is an abstraction expression if* $\{u_1, \ldots, u_n\} \subseteq U$,
- $AE_1 \wedge AE_2$ *is an abstraction expression if* AE_1 *and* AE_2 *are abstraction expressions,*
- $AE \triangleright \{b_1, \ldots, b_n\}$ *is an abstraction expression if* AE *is an abstraction expression and* $\{b_1, \ldots, b_n\} \subseteq B$.

We define the semantics of *Abstraction Expressions* by giving an associated equivalence relation. The equivalence relation determines which nodes are to be merged.

Definition 2 (Semantics of Abstraction Expressions). *The associated equivalence relation* $\sim_{AE}$ *to an* Abstraction Expression AE *is defined inductively as follows:*

- $x \sim_{\{u_1,\ldots,u_n\}} y :\Leftrightarrow \bigwedge_{u \in \{u_1,\ldots,u_n\}} u(x) = u(y)$,
- $x \sim_{AE_1 \wedge AE_2} y :\Leftrightarrow x \sim_{AE_1} y \wedge x \sim_{AE_2} y$,
- $x \sim_{AE \triangleright \{b_1,\ldots,b_n\}} y :\Leftrightarrow x \sim_{AE} y \wedge \bigwedge_{b \in \{b_1,\ldots,b_n\}} \forall z.(\bigsqcup_{\{w \mid w \sim_{AE} z\}} b(x,w) = \bigsqcup_{\{w \mid w \sim_{AE} z\}} b(y,w))$.

The *Abstraction Expression* $\{u_1, \ldots, u_n\}$ is equivalent to *Canonical Abstraction* over $\{u_1, \ldots, u_n\}$. The abstraction depicted in case (b) of Figure 12 can be specified using the *Abstraction Expression* $\{x\} \triangleright \{dle\}$. It will be interesting to see whether there are more applications, where abstraction can be specified more easily using such expressions than by plain *Canonical Abstraction*.

Dead Predicates To speed up the analyses we included additional actions in the control flow graphs of the tree-based programs. These actions nullified certain variables and allowed the engine to collapse structures that were otherwise isomorphic. This was only done for unary predicates representing dead variables, i.e. predicates that further steps of the analysis did not rely on. These predicates could be called dead predicates. A similar effect could have been achieved by marking these predicates as non-abstraction predicates locally. This approach was previously described in Roman Manevich's Master Thesis [Man03]. These dead predicates could be determined by a preceding static analysis. At the time the analyses were conducted it had not been integrated into TVLA yet. We believe that it may dramatically increase the performance of analyses in larger programs that contain many loosely coupled sections. Unfortunately, we cannot give experimental results about the magnitude of the effect. Our analysis for the tree-based removal method did not terminate within days without this optimization. Of course, the optimization could also decrease precision, because more structures are collapsed, possibly losing relevant information. However, in such a case it seems that the wrong abstraction is used, but the analysis succeeds by coincidence.

4 Conclusion

We created a precise shape analysis for programs that are manipulating ordered trees. It is particularly tailored to invariants of the tree data structure. Choosing the right instrumentation predicates required a thorough understanding of the data structures involved. This meant identifying that reachability alone is not very interesting, but that the first edge on a path from one node to another is important. We implemented the analysis in TVLA [LA00,LAS00] and successfully applied it to methods of the tree-based set implementation. The analysis proved that the implementation complies to the axioms (3) and (4) of the ADT Set specification.

$$a \in s.\mathtt{insert}(b) \leftrightarrow a =_{el} b \vee a \in s, \ (3)$$
$$a \in s.\mathtt{remove}(b) \leftrightarrow a \neq_{el} b \wedge a \in s \ \ (4)$$

We used the *isElement*-predicate to relate different analyses. Our analyses of the insertion and removal methods established the two axioms in terms of *isElement*. Another analysis then established the equivalence between *isElement* and the set membership method $\cdot.\mathtt{insert}(\cdot)$. Adapting existing analyses for singly-linked lists allowed us to show the same property for our list-based set implementation.

Inspired by a family of instrumentation predicates used in our tree analysis, we propose a new way of specifying abstractions by so-called "Abstraction Expressions". These expressions allow to not only use unary but also binary predicates in the abstraction specification. "Abstraction Expressions" have the same expressive power as *Canonical Abstraction*. However, we need a smaller number of predicates to express certain abstractions.

5 Future Work

We successfully analyzed a tree-based set implementation. Since the analysis is tailored to the underlying data structure and not to the specific algorithms employed, it might be possible to analyze other algorithms working on trees using the same abstraction.

The tree structure lends itself naturally to recursion. We could possibly combine recent work on interprocedural shape analysis [RS01] with our abstractions to be able to analyze recursive implementations. Modern data structure libraries usually contain more efficient set implementations using balanced trees, like AVL or red-black trees. They maintain even more complicated data structure invariants than the unbalanced tree implementation we analyzed. Algorithms on these structures can usually be implemented more easily using recursion, too. Extending our analysis to cope with the invariants of balanced trees might make such algorithms amenable as well.

Abstraction Expressions seem useful where we want to distinguish individuals if they differ by binary predicates originating from individuals that we distinguish. In our tree-based analysis, we could separate smaller and larger tree elements. In the shape analysis for RESET, we could use the set membership relation to separate individuals in terms of the sets they belong to. An implementation of the concept would allow deeper insight into the usefulness of the approach.

References

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 296–310, New York, NY, USA, 1990. ACM Press.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification I*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.

[EM90] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, New York, NY, USA, 1996. ACM Press.

[LA00] Tal Lev-Ami. TVLA: A framework for kleene based static analysis. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2000.

[LARSW00] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 26–38, New York, NY, USA, 2000. ACM Press.

[LAS00] Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 280–301, London, UK, 2000. Springer-Verlag.

[LEW97] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of abstract data types*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[LKR04] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking using set interfaces and pluggable analyses, 2004.

[Man03] Roman Manevich. Data structures and algorithms for efficient shape analysis. Master's thesis, Tel-Aviv University, School of Computer Science, Tel-Aviv, Israel, January 2003. Available at www.cs.tau.ac.il/rumster/msc_thesis.pdf.

[Mic04] Sun Microsystems. Java 2 platform standard edition 5.0 api specification, 2004. Available at http://java.sun.com/j2se/1.5.0/docs/api/.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA - A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, 1999.

[MS96] David R. Musser and Atul Saini. *STL tutorial and reference guide*, volume - of *Addison-Wesley professional computing ser.* Addison-Wesley, 1996.

[Rei05] Jan Reineke. Shape analysis of sets. Master's thesis, Universität des Saarlandes, Germany, June 2005. Available at http://rw4.cs.uni-sb.de/ reineke/publications/MasterReineke.pdf.

[RS01] Noam Rinetzky and Mooly Sagiv. Interprocedural shape analysis for recursive programs. *Lecture Notes in Computer Science*, 2027:133–149, 2001.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3–valued logic. In *Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3–valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.