Noname manuscript No.
(will be inserted by the editor)

Verification of B$^+$ Trees by Integration of Shape Analysis and Interactive Theorem Proving ⋆

Gidon Ernst, Gerhard Schellhorn, Wolfgang Reif

University of Augsburg, Germany,
e-mail: {ernst,schellhorn,reif}
@informatik.uni-augsburg.de

The date of receipt and acceptance will be inserted by the editor

Abstract Interactive proofs of correctness of pointer-manipulating programs tend to be difficult. We propose an approach that integrates shape analysis and interactive theorem proving, namely TVLA and KIV. The approach uses shape analysis to automatically discharge proof obligations for various data structure properties, such as "acyclicity". To this purpose we define a mapping between typed algebraic heaps and TVLA. We verify the main operations of B$^+$ trees by decomposing the problem into three layers: The top-level is an interactive proof of the main recursive procedures. The actual modifications of the data structure are verified with shape analysis. TVLA itself relies on problem specific constraints and lemmas, that were proven in KIV as a foundation for an overall correct analysis.

Key words Theorem Proving, Shape Analysis, B$^+$ trees, Pointer Structures

1 Introduction

Interactive theorem provers are powerful tools for formal verification. However, using them to reason about pointer structures in the presence of destructive updates can be quite difficult. In contrast, tools based on shape analysis, such as TVLA [21,3], are specifically designed to perform well in these situations, but cannot deal with precise arithmetic and induction, for example.

The Goal of this work is to integrate the two verification approaches to achieve a higher degree of proof automation. We perform a conceptual integration by translating manually between the two worlds. We evaluate how the rather different high-level specification style of algebraic specifications can be mapped to the shape graphs and logic of TVLA.

Our approach uses algebraic specifications and symbolic execution as a convenient framework for verification of the relevant algorithms. Shape analysis is used as a kind of decision procedure that discharges some of the proof goals automatically. However, the proposed integration differs from common approaches with similar goals (that use for example SAT-solvers [2,7]): shape analysis is parameterized with constraints that are specific to the problem domain. These constraints are used as unvalidated assumptions to guide the automatic proof. To ensure a correct analysis they have to be verified interactively. We use KIV [18], an interactive verifier based on structured many-sorted specification.

We exemplify the approach by verifying B$^+$ trees [1]. These are ordered, balanced trees that are commonly used to implement indices in databases or file systems. They have several invariants regarding tree shape, sorting, balance, and node sizes.

By combining KIV and TVLA, we have verified that our implementation of the insertion and deletion operations on B$^+$ trees maintains the invariants and that they are a refinement of their counterparts on algebraic sets. We ensure correctness of the shape analysis specifications and – where possible – abstract from B$^+$ trees as the concrete data structure, so that many generic constraints can be reused for other pointer structures. The proofs done in KIV as well as the TVLA input files are available online at [8].

The paper extends our previous work [9] as follows: We describe the encoding of an explicit heap domain and of the type system in TVLA. The formal mapping is extended to program statements, and we show that the given proof obligations for a sound integration are indeed sufficient. Furthermore, we describe a practical approach to establish preconditions at the program entry point.

This work is organized as follows: Section 2 introduces B$^+$ trees, our verification approach and an algebraic specification of pointer structures. Section 3 describes the shape analysis implemented by TVLA. Section 4 describes the technical details of the integration. Section 5 formalizes B$^+$ tree invariants and explains how these can be tracked with shape analysis. Sec. 6 summarizes our experiences. Related work is compared in Sec. 7, and Sec. 8 draws conclusions.

2 B$^+$ Trees and Approach

B$^+$ trees are ordered, balanced n-ary trees. They are used to implement large sets of keys (or key-value maps). They maintain several invariants to guarantee logarithmic-time operations. The main operations on B$^+$ trees are lookup, insertion, and deletion. In a B$^+$

⋆ The final publication is available at Springer via http://dx.doi.org/10.1007/s10270-013-0320-1

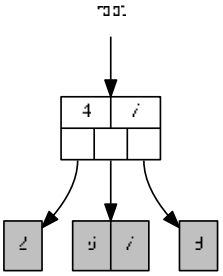


Fig. 1: B^+ tree of rank $N = 1$ with arrays

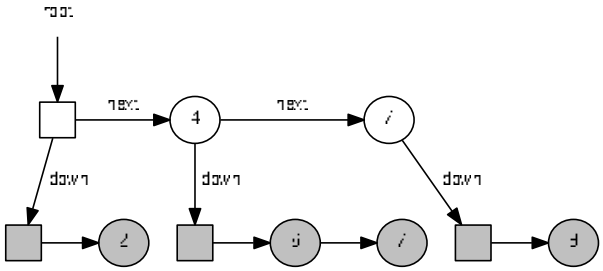


Fig. 2: B^+ tree of rank $N = 1$ with linked lists

tree of *rank* N, a node is either a *branch* that stores $N \leq k \leq 2N$ keys and $k+1$ downward pointers, or it is a *leaf* that stores between N and $2N$ keys (for sets) or key-value mappings. There is an exception to this rule for the root, which must contain at least one key instead of N. A total order on keys is required. The actual *content* of the B^+ tree only consists of the information in the leaves, the keys in the branch nodes organize efficient access (in contrast to B-trees that store content in internal nodes, too). A B^+ tree is *balanced* if all leaves are on the same level, and *sorted* if in each node the keys are sorted in increasing order, while subtrees only contain keys between adjacent keys in the parent.

Figure 1 shows a B^+ tree as it is commonly implemented with arrays. It has one internal node with two keys and three leaves (marked in grey) and represents the set $\{2, 6, 7, 9\}$.

We use linked lists instead of arrays for the representation of nodes. An encoding of arrays in TVLA has been developed [12,13], but unfortunately the version of TVLA version supporting this encoding is not available.

Figure 2 shows the same B^+ tree in this model. Graphical nodes displayed as boxes serve as representatives of entire B^+ tree nodes and accommodate for the first down-pointer. They are subsequently called *heads*. The round nodes store keys and are subsequently called *entries*. The edge labels next and down indicate the names of the corresponding selectors of the respective objects.

```
insert(k;rn) {
  if rn = null
  then create_root(k; rn)
  else
    insert_node(k,rn)
    if overfull(rn)
    then split_root(;rn)
}

delete(k;rn) {
  if rn ≠ null
  then
    if   is_leaf(rn)
       ∧ rn.next.next = null
       ∧ rn.next.key  = k
    then rn := null
    else delete_node(k,rn)
}
```

Fig. 3: Top-level B^+ tree routines

2.1 Algorithms

Both the insertion and deletion algorithm on B^+ trees essentially follow the same strategy: recursively traverse the tree down to a leaf node that is responsible for holding the given key k and perform the desired modification locally on that leaf. This may cause an underflow or overflow with respect to the node size invariant, which is restored by restructuring the tree. For example, an overfull leaf is split into halves, and an additional down-pointer and key are added to the parent. Therefore, restructuring may cascade upwards along the path the recursive descent has taken. This may lead to growth and shrinking of the tree, when a new root node is allocated or an empty root node is deleted.

The top-level routines insert and delete are shown in Fig. 3. They receive the current key as a parameter k and the root node in rn. The semicolon separates input from reference parameters, and rn is passed by reference as the root of the tree may change.

The traversal is implemented by mutually recursive subroutines as shown in Fig. 4 – insert_node descends at a node head (displayed as boxes in Fig. 2), while insert_bentry performs similar actions at branch entries (displayed as circles).

The actual modifications of the data structure occur inside the functions insert_leaf, which stores a key in a leaf, and split_node(rn ; rt) and split_bentry(rbe ; rt) that split the overfull down-child of rn resp. rbe at its median elements.

Figure 5 shows the implementation of split_node and its effect on the data structure, rn is the overfull node, its parent is rp and r1 denotes the entry just before the median r2. The newly allocated node is returned in rt. The parameter rt is not actually used in the implementation but in the specification.

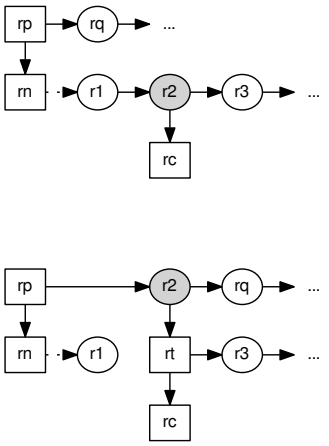
```
insert_node(k,rn) {
  if is_leaf(rn)
  then insert_leaf(k,rn)
  else if k ≤ rn.next.key
    then
      insert_node(k,rn.down)
      if overfull(rn.down)
      then split_node(rn;rt)
    else
      insert_bentry(k,rn.next)
}

insert_bentry(k,rbe) {
  if   rbe.next = null
     ∨ k ≤ rbe.next.key
  then
    insert_node(k,rbe.down)
    if overfull(rbe.down)
    then split_bentry(rbe;rt)
  else
    insert_bentry(k,rbe.next)
}
```

Fig. 4: Mutually recursive insertion routines

```
split_node(rp; rt) {
  rn := rp.down
  median(rn;r1)
  r2 := r1.next
  r3 := r2.next
  r1.next := null
  if is_leaf(rn)
  then
    r2 := new BEntry
    rt := new Leaf
  else
    rc := r2.down
    rt := new Branch
    rt.down := rc
  r2.down := rt
  rt.next := r3
  rq := rp.next
  rp.next := r2
  r2.next := rq
}
```

(a) Code



(b) Modification of structures

Fig. 5: Restructuring routine split_node

The deletion algorithm as shown in Fig. 6 is more complicated than insertion because balance may be restored either by merging nodes or by transferring keys between adjacent nodes from left to right or vice versa.

2.2 Verification Approach

The verification of B^+ trees must establish two properties of the implementation: Insertion and deletion retain

```
delete_node(k,rn) {
  if is_leaf(rn)
  then delete_leaf(k,rn)
  else if k ≤ rn.next.key
    then
      delete_node(k,rn.down)
      if underful(rn.down)
      then if size(rn.next.down) > N + 1
        then transfer_node_left(rn)
        else merge_node(rn)
    else
      delete_bentry(k,rn,rn.next)
      if underful(rn.next.down)
      then if size(rn.down) > N + 1
        then transfer_node_right(rn)
        else merge_node(rn)
}

delete_bentry(k,rbe) {
  if   rbe.next = null
     ∨ k ≤ rbe.next.key
  then
    delete_node(k,rbe.down)
    if   underful(rbe.down)
       ∧ rbe.next ≠ null
    then if size(rbe.next.down) > N + 1
      then transfer_bentry_left(rbe)
      else merge_bentry(rbe)
  else
    delete_bentry(k,rbe.next)
    if   underful(rbe.next.down)
       ∧ rbe.next ≠ null
    then if size(rbe.down) > N + 1
      then transfer_bentry_right(rbe)
      else merge_bentry(rbe)
}
```

Fig. 6: Mutually recursive deletion routines

the B^+ tree invariants, and the operations modify the represented set of keys correctly.

Invariants are collected in a predicate $\mathsf{btree}(r)$ (formalized in Sec. 5) that specifies r as the root of a proper B^+ tree. The set of elements that a B^+ tree with root r represents is denoted by $\mathsf{elts}(r)$ in the following.

Correctness assertions for a program α are stated as *contracts* with a precondition φ and a postcondition ψ. We use the weakest precondition operator [5] to formalize contracts in this paper, i.e., $\varphi \to \mathbf{wp}(\alpha, \psi)$ states that if φ holds then α terminates and ψ holds after executing α. Similarly, the weakest *liberal* precondition is used to encode partial correctness assertions $\varphi \to \mathbf{wlp}(\alpha, \psi)$, which correspond the Hoare triples $\{\varphi\}\ \alpha\ \{\psi\}$.
Formally, we prove for insertion

$$\mathsf{btree}(r) \wedge e = \mathsf{elts}(r) \qquad (1)$$
$$\to \mathbf{wp}(\mathsf{insert}(k;\, r),\ \mathsf{btree}(r) \wedge \mathsf{elts}(r) = e \cup \{k\})$$

and for deletion

$$\mathsf{btree}(r) \wedge e = \mathsf{elts}(r) \quad (2)$$
$$\rightarrow \mathbf{wp}(\mathsf{delete}(k;\, r),\ \mathsf{btree}(r) \wedge \mathsf{elts}(r) = e \setminus \{k\})$$

where variable e denotes the elements of the initial tree. Programs insert and delete are called with the key k to insert or delete. Note again that the root r of the tree is passed by reference.

The correctness criteria for the verification of B^+ trees can be decomposed into three layers along the structure of the algorithms:

- The top-level consists of interactive proofs of the recursive insertion and deletion algorithms.
- The intermediate layer consists of correctness assertions for subroutines that perform actual modifications to the data structure, such as split_node, split_bentry and insert_leaf. These are verified by shape analysis and used as assumptions of the top-level proofs.
- The basis for the verification is an algebraic specification of B^+ trees as pointer structures. It also serves for consistency proofs of the constraints and theorems required for shape analysis.

The interactive proofs in KIV [18] are performed by symbolic execution of the program source code. KIV implements the wp-calculus in the form of Dynamic Logic [14], but any prover that supports Hoare calculus would be sufficient.

Calls to subroutines which are verified with shape analysis are dispatched via their contracts so the interactive verification does not have to deal with the implementation of these subroutines at all. These contracts form the interface between the top and intermediate layer. TVLA shows partial correctness assertions only, thus we additionally show in KIV that all subroutines terminate, i.e., $\varphi \rightarrow \mathbf{wp}(\alpha, \mathsf{true})$.

Subroutines can be classified into restructuring (such as split_node) and content modifications (insert_leaf and delete_leaf). There are several restructuring routines (split, merge, transfer a key to left and right sibling) for leaf-level and internal operations that also differ in whether they affect a node head. The contracts of subroutines of the same class are very similar; concrete examples are given in (3), (4), and (5),[1] where r denotes the root of the tree. In the precondition, $\mathsf{okpath}(r', k)$ specifies that key k belongs into the subtree of r', i.e., k is correctly ordered with respect to the parents of r'. Predicate okpath is defined in Sec. 5.4.

$$\mathsf{btree}(r) \wedge \mathsf{reachable}(r, r') \quad (3)$$
$$\wedge\, e = \mathsf{elts}(r) \wedge \mathsf{okpath}(r', k)$$
$$\rightarrow\ \mathbf{wp}(\mathsf{insert\_leaf}(k, r'),\ \mathsf{btree}(r) \wedge \mathsf{elts}(r) = e \cup \{k\})$$

[1] This contract ignores the node sizes, see (33) for the full contract.

$$\mathsf{btree}(r) \wedge \mathsf{reachable}(r, r') \quad (4)$$
$$\wedge\, e = \mathsf{elts}(r) \wedge \mathsf{okpath}(r', k)$$
$$\rightarrow\ \mathbf{wp}(\mathsf{delete\_leaf}(k, r'),\ \mathsf{btree}(r) \wedge \mathsf{elts}(r) = e \setminus \{k\})$$

$$\mathsf{btree}(r) \wedge \mathsf{reachable}(r, rp) \wedge e = \mathsf{elts}(r) \quad (5)$$
$$\rightarrow\ \mathbf{wp}(\mathsf{split\_node}(rp; rt),\ \mathsf{btree}(r) \wedge \mathsf{elts}(r) = e)$$

The main proof for the insert algorithm is concerned with the mutually recursive procedures insert_node and insert_bentry (see Fig. 4). We combine these into one proof obligation, so that recursive calls from one function to the other are covered by the induction hypothesis. The induction is carried out over the number of nodes in (sub)trees. The critical proof step is to establish $\mathsf{okpath}(r'.\mathsf{next}, k)$ respectively $\mathsf{okpath}(r'.\mathsf{down}, k)$ when a selector is followed, given that $\mathsf{okpath}(r', k)$ holds for the current node r' and key k. This fact follows from the key comparisons in the algorithm.

An alternative to this decomposition scheme is to verify the top-level with TVLA as well, for example with the technique presented in [20], which automatically computes abstractions of subroutines. However, the interactive proof also shows termination and the effort for the recursion is reasonably low.

2.3 Algebraic Formalization of Pointer Structures

Pointer structures consist of *objects* that live inside a *heap* and are accessed indirectly via typed *references*. The heap H is a partial, polymorphic function

$$H : \mathsf{ref}[T] \rightharpoonup T$$

that maps ("dereferences") allocated references with $r \in \mathsf{dom}(H)$ and $r : \mathsf{ref}[T]$ to objects $o = H(r)$ of corresponding type T. $H\{r \mapsto o\}$ denotes the heap that results from H by updating the object stored under r to o, possibly allocating r. With this scheme, heap access is statically type-checked within the logic's type system.

We model objects as instances of free data types. For B^+ trees we obtain three sorts: Node for branch and leaf heads and BEntry, LEntry for their respective entries. Let refn abbreviate ref[Node] and let rn denote variables of type refn in the following (similar conventions for refbe, rbe and refle, rle).

data Node	=	Branch(next: refbe; down: refn)
	|	Leaf(next: refle)
data BEntry	=	BEntry (key: Key; next: refbe; down: refn)
data LEntry	=	LEntry (key: Key; next: refle)

Node, for example, is a type freely generated by the constructors Branch and Leaf. Overloaded selector functions next and down retrieve the respective constructor arguments and are applied in postfix notation similar to Java fields, e.g., if $o = \mathsf{Leaf}(r)$ then $o.\mathsf{next} = r$.

KIV uses an explicit heap variable H in programs that is passed as a reference parameter to all subroutines (left implicit in Figs. 3 to 6). Selector assignments are thus updates to H, for example, for r : BEntry the statement $r.\mathsf{next} := r'$ is executed as

$$H := H\{r \mapsto \mathsf{BEntry}(H(r).\mathsf{key}, r', H(r).\mathsf{down})\}$$

To bridge the gap to the untyped logic of TVLA, we define supersorts/sum-types ref, object, and the enumeration of selectors sel as follows

$$\begin{aligned} \mathsf{ref} &= \mathsf{refn} + \mathsf{refbe} + \mathsf{refle} \\ \mathsf{object} &= \mathsf{Node} + \mathsf{BEntry} + \mathsf{LEntry} \\ \mathsf{sel} &= \mathsf{next} \mid \mathsf{down} \end{aligned}$$

We assume a constant null : ref that is never allocated in a heap. A predicate wt : object × sel such that $\mathsf{wt}(o, s)$ iff $o.s$ is well typed allows us to define heap properties without referring to concrete types of the case study, for example paths and treeness (see Sec. 5.1).

Algebraically specified heaps can contain *dangling pointers* (references pointing outside the heap). A consistent heap requires that whenever an object is stored in the heap, all of its reference selectors are either null or point inside the heap again. In the remainder of this text, we assume that all heaps are consistent.

$$\begin{aligned} &\mathsf{consistent}(H) \leftrightarrow \\ &\quad \forall r \in \mathsf{dom}(H), s : \mathsf{sel}.\ \mathsf{wt}(H(r), s) \\ &\qquad \rightarrow (H(r).s = \mathsf{null} \vee H(r).s \in \mathsf{dom}(H)) \end{aligned} \tag{6}$$

3 Parametric Shape Analysis by 3-Valued Logic

Parametric shape analysis [21] is an instance of abstract interpretation designed for the analysis of pointer-manipulating programs. The actual computations are performed over an approximation of concrete states. A fully automatic analysis is achieved by keeping the abstract state space finite, so that it can be explored exhaustively. The analysis is conservative, i.e., proofs sometimes fail even if the program is correct, but not the other way round. Parametric shape analysis is a generic framework. The user controls abstraction, the encoding of data structure properties and the program statement semantics. The approach is implemented in the TVLA (Three-Valued Logic Analyzer) tool.

Parametric shape analysis is based on untyped first-order logic with transitive closure but without function symbols (which are encoded as predicate symbols). The concrete semantics of formulas $\mathcal{U}, z \models \varphi$ is given by unsorted algebras $\mathcal{U}$ (with a carrier set U of non-null references) and valuations z in the standard way. These algebras (also called logical structures in the following) are used to encode program states.

Part of the logical signature over which algebras $\mathcal{U}$ are defined is constructed from the program signature and data types. A program variable x becomes a unary predicate symbol $\mathsf{x}(r)$ that is constrained to hold for at most one node r with $\mathsf{x} = r$, if $r \neq \mathsf{null}$.

Selectors become binary predicate symbols, for example $H(r_1).\mathsf{next} = r_2$ is encoded as $\mathsf{next}(r_1, r_2) = 1$, if $r_2 \neq \mathsf{null}$. These predicates are constrained to be partial functions. The predicate symbols arising from the program signature are called *core predicates*.

3.1 Abstraction and Execution

The key idea of the abstraction in parametric shape analysis is to represent sets of concrete structures by finitely many bounded abstract algebras. The domain of truth values is extended to contain a third value $\frac{1}{2}$ (sometimes called "indefinite truth"). Intuitively, if a formula evaluates to $\frac{1}{2}$ in an abstract structure then this formula evaluates to true in some of the represented concrete structures and to false in some others.

Note that $\frac{1}{2}$ is only used in abstract structures. Nevertheless, proving a theorem requires that it evaluates to definite truth over all concrete structures $\mathcal{U}$ represented by the three-valued model.

The abstraction partitions objects allocated in the heap into finitely many equivalence classes. As an example, Fig. 7 shows the graphical representation of several concrete linked lists. The corresponding *shape graph* in Fig. 8 represents *all* linked lists with at least two nodes. There is one singleton equivalence class a for the head node. All other memory cells are grouped into the doubly circled *summary node* $\mathsf{b} = \{\ \mathsf{b1}, \ldots\ \}$.

The partitioning is due to the fact that program variable x points to a. In general, the equivalence classes are defined by unary *abstraction predicates*: two objects are in the same class if they are indistinguishable by abstraction predicates. Predicates arising from program variables are abstraction predicates by default, so that objects pointed to by a program variable are distinguished and can be kept in separate singleton classes.

Complementary, binary predicates capture relations between objects and are lifted to summary nodes by truth blurring. For example, the dotted arrow from a to b in Fig. 8 indicates that a.next may point to *some* node in class b but not necessarily all of them, or to none at all. This fact is encoded as $\mathsf{next(a,b)} = \frac{1}{2}$.

Similarly, the dotted arrow from b to itself indicates that some nodes may be linked by next edges. Note that information about the internal structure of the summary node is lost, in particular, concrete structures represented by this shape graph may be disconnected or cyclic.

Programs are represented by finite transition systems, each state corresponding to a specific value of the program counter. For each state, shape analysis computes the set of shape graphs that approximate all heap structures that are possible at that program point. This

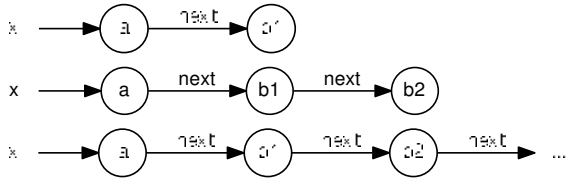


Fig. 7: Concrete linked lists

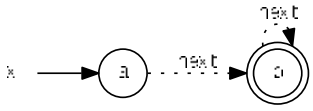


Fig. 8: Abstraction of linked lists

can be done by a fixpoint computation, since the number of states as well as the number of possible shape graphs is finite (there are only finitely many abstraction predicates to distinguish objects).

To compute the fixpoint, for each transition *stm* a *precondition* pre_{stm} (`%p` in TVLA) and a set of *update formulas* $\varphi^{\mathsf{p}}_{stm}$ must be defined. The precondition encodes tests of conditionals or loops, the update formulas must specify the effects of assignments for all core predicates p. A transition is executed by evaluating each update formula in the old state yielding the predicate's definition in the new state.

As an example, y := x is represented by the update formulas $\varphi^{\mathsf{y}}_{\mathsf{y}:=\mathsf{x}}(r)$ for y and $\varphi^{\mathsf{p}}_{\mathsf{y}:=\mathsf{x}}(\underline{r})$ for all other predicates p, defined as

$$\varphi^{\mathsf{y}}_{\mathsf{y}:=\mathsf{x}}(r) \equiv \mathsf{x}(r)$$
$$\varphi^{\mathsf{p}}_{\mathsf{y}:=\mathsf{x}}(\underline{r}) \equiv \mathsf{p}(\underline{r})$$

where $\underline{r} = r_1, \ldots, r_n$ is a vector of variables. We adopt the convention not to mention identity update formulas.

Sagiv et al. [21] define preconditions and update formulas for standard statements such as assignments, selector access and case distinctions.

The update formula for y for selector access y := x.sel is defined as

$$\varphi^{\mathsf{y}}_{\mathsf{y}:=\mathsf{x.sel}}(r) \equiv \exists r_1.\ \mathsf{x}(r_1) \wedge \mathsf{sel}(r_1, r) \qquad (7)$$

Selector assignment statements x.sel := y assume that x.sel = null, so that edges are either added or removed but not both in one step. Therefore when translating KIV programs to TVLA, these assignments have to be rewritten into x.sel := null; x.sel := y. Selector assignments can be understood to add the edge from x to y to the predicate sel, keeping the previous edges. The corresponding update formula for sel is defined as

$$\varphi^{\mathsf{sel}}_{\mathsf{x.sel}:=\mathsf{y}}(r_1, r_2) \equiv \mathsf{sel}(r_1, r_2) \vee (\mathsf{x}(r_1) \wedge \mathsf{y}(r_2)) \qquad (8)$$

The computation is performed entirely on the abstract level. However, it guarantees sound approximation, that is, whenever a transition is taken from a state

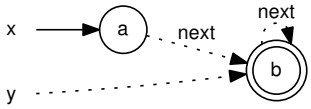


Fig. 9: Imprecise execution of y := x.next

representing a concrete structure $\mathcal{U}$, equation (9) holds for all predicates p, (complete) valuations z, and resulting concrete structures $\mathcal{U}'$. This makes it possible to reason about executions entirely in a two-valued setting, as exploited in Sec. 4.2.

$$\mathcal{U}', z \models \mathsf{p}(\underline{r}) \quad \text{iff} \quad \mathcal{U}, z \models \varphi^{\mathsf{p}}_{stm}(\underline{r}) \qquad (9)$$

During the run of a program, the abstraction is dynamically adjusted with every statement. Suppose, the statement y := x.next should be executed in a state represented by Fig. 8. Naively applying (7) yields $\mathsf{y}(\mathsf{a}) = 0$ and $\mathsf{y}(\mathsf{b}) = \frac{1}{2}$ in the new state, the resulting shape graph is shown in Fig. 9. This is a significant loss of precision, because y = x.next is not necessarily true in the new structure – we have to get hold of the object x.next in a separate node *before* the assignment.

Parametric shape analysis allows us to partially reverse the abstraction by splitting summary nodes with respect to a given condition (`%f` in TVLA). This operation is called *materialization*. Materialization only forces the analysis to consider different cases separately. The set of represented concrete structures remains the same. Continuing the example, memory cells are discerned whether they are a next successor of a:

- Nodes r in equivalence class b1 with $\mathsf{next}(\mathsf{a}, r) = 1$
- Nodes r in equivalence class b2 with $\mathsf{next}(\mathsf{a}, r) = 0$

Three cases arise as shown in Figs. 10 to 12, because either half of the split can be empty – though not both at the same time as a summary node always represents at least one concrete cell.

The analysis may furthermore conclude that b1 must be a singleton since next is a functional predicate: a can only have one next successor. To eliminate the back-arrow from b2 to b1 and to ensure connectedness, further information is necessary as described in the next section.

After the split, the assignment y := x.next can be executed for each of the resulting cases without loss of information.

The conditions that cause such splits are called *focus formulas*. Standard statements define focus formulas to retain that program variables point to singleton nodes e.g., for the assignment y := x.next these are $\mathsf{x}(r)$ and $\exists r_1.\ \mathsf{x}(r_1) \wedge \mathsf{next}(r_1, r)$.

Technically, TVLA ensures that all focus formulas $\phi(\underline{r})$ of a transition evaluate to definite truth (either 0 or 1) for all combinations of nodes $\underline{r}$ in the resulting shape graphs. Additional focus formulas can be given to cause extra splits of summary nodes. We will need these in

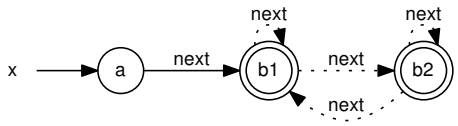


Fig. 10: Case 1 of materializing Fig. 8

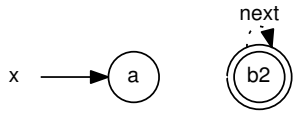


Fig. 11: Case 2

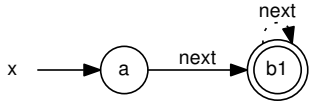


Fig. 12: Case 3

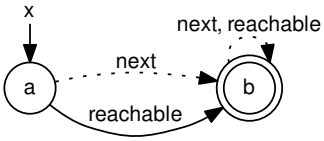


Fig. 13: Connectedness via reachable

Sec. 4.4 and Sec. 5.3. Limitations of the implementation of materialization in TVLA are discussed in Sec. 4.4.

3.2 Preserving Information

Switching to a finite domain cannot preserve all information available in the infinite domain. To preserve more information, two strategies are possible, the *instrumentation* and the *guard* strategy. The first explicitly defines additional *instrumentation predicates*. Predicates step and reachable, defined by

$$\mathsf{step}(r_1, r_2) \leftrightarrow \bigvee_{s \in \mathsf{sel}} s(r_1, r_2)$$

$$\text{and} \quad \mathsf{reachable}(r_1, r_2) \leftrightarrow \mathsf{step}^*(r_1, r_2)$$

are such instrumentation predicates (`%i` in TVLA). Here, step is the disjunction over the set of selectors sel and step^* is the reflexive transitive closure of step. The value of instrumentation predicates is explicitly stored in shape graphs as additional information, as shown in Fig. 13. The solid arrow indicates that *all* nodes in b are reachable from a. Note that just evaluating the definition of reachable in the structure shown in Fig. 8 yields less precise information.

By default, executing a transition reevaluates the definition of instrumentation predicates in the new state. However, this may lose the definite information that an instrumentation predicate stores. For the structure

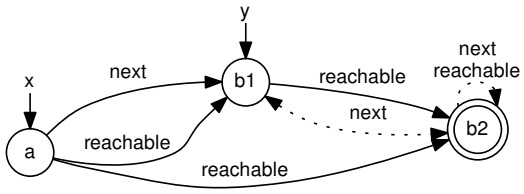


Fig. 14: Materializing reachable

shown in Fig. 14 (resulting from the analogue of case 1 of the assignment y := x.next executed in Fig. 13) we have $\mathsf{next}^*(\mathsf{b1}, \mathsf{b2}) = \frac{1}{2}$. To prevent information loss, parametric shape analysis allows explicit update formulas for instrumentation predicates. For the assignment y := x.next, reachability is unaffected[2], as expressed by

$$\varphi^{\mathsf{reachable}}_{\mathsf{y} := \mathsf{x.next}}(r_1, r_2) \equiv \mathsf{reachable}(r_1, r_2)$$

Assuming that $\mathsf{x.sel} = \mathsf{null}$ and $\mathsf{y} \neq \mathsf{null}$ the update formula for reachable and an assignment x.sel := y is defined as follows

$$\varphi^{\mathsf{reachable}}_{\mathsf{x.sel} := \mathsf{y}}(r_1, r_2) \equiv$$
$$\mathsf{reachable}(r_1, r_2) \ \vee \ (\mathsf{reachable}(r_1, \mathsf{x}) \wedge \mathsf{reachable}(\mathsf{y}, r_2))$$

Update formulas that do not comply with the definitions of instrumentation predicates lead to an unsound analysis. Proof obligation (12), defined in the next section, shows how to prove soundness of such update formulas.

The guard strategy uses global invariants *INV* that hold at all times during the execution of an algorithm. Formally, these are defined by *consistency rules* (`%r` in TVLA). For example, the following rule expresses that an algorithm never creates cyclic structures.

$$\mathsf{reachable}(r_1, r_2) \rightarrow \neg \mathsf{next}(r_2, r_1) \tag{10}$$

INV is the conjunction of all universally closed consistency rules.

Note that Sagiv et al. use a new connective $\triangleright$ for consistency rules with a particular three-valued semantics: whenever φ evaluates to a definite value in $\varphi \triangleright \psi$, the structure can be made more precise using ψ. In particular, (10) eliminates the dotted next-arrow from b2 to b1 in Fig. 14. Since $\triangleright$ is equivalent to $\rightarrow$ in a two-valued setting, we do not make this distinction later on.

The analysis must report an error whenever invariants are violated by assignments. This is done attaching guards ψ_{stm} (`%message` in TVLA) to transitions. The analysis aborts with an error if a guard evaluates to 0 or $\frac{1}{2}$. Guards are also used to detect unsafe operations, such as null-dereferences.

Guards must be strong enough to ensure that *INV* is preserved by transitions, which is expressed by proof obligation (15) (see next section).

[2] TVLA is in fact able to infer this by a dependency analysis of predicate definitions with respect to updates.

TVLA provides several abbreviations for typical constraints, such as `unique` to constrain unary predicates to single nodes, and `function` to constrain binary predicates to functions. We use these whenever appropriate.

To sum up, a transition *stm* is labeled with

1. A precondition pre_{stm} that enables execution of the transition,
2. A guard ψ_{stm} that is used to detect violation of global invariants *INV*,
3. Update formulas $\varphi^{\mathsf{p}}_{stm}$ to specify how predicate p is changed by *stm*,
4. Focus formulas to perform partial concretization (case splits).

In the following sections, we often disregard that program variables are encoded as unary predicates and thus omit various existential quantifiers. However, we use the convention that program variables are written in sans-serif typeface. The formula $\mathsf{reachable}(\mathsf{x}, r)$ must thus be read as an abbreviation for $\exists r_0.\ \mathsf{x}(r_0) \wedge \mathsf{reachable}(r_0, r)$ in TVLA with one free logical variable r.

4 Integration and Proof Obligations

This section shows the mapping between typed algebraic specification and untyped structures. We explain how we track the heap domain and how references are allocated. This serves as the basis for the formal definition of the translation between KIV and TVLA, which is in essence a standard construction of a homomorphism. The translation leads to proof obligations for the correctness of update formulas and guards.

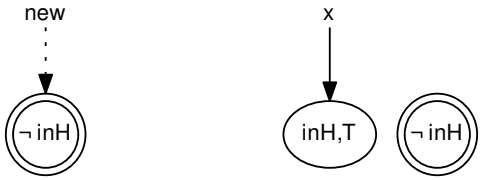
Two further (practical) issues are addressed as well, namely the encoding of the type system and the generation of initial structures from symbolic conditions.

4.1 Heap Domain and Allocation

Parametric shape analysis provides two idioms for dynamic memory management that differ in how the carrier set U of structures $\mathcal{U}$ is viewed. In both cases, U is a set of references.

The first idiom maintains only the *allocated* references in U. There is a designated operation that extends the current structure by new nodes: $U' = U \uplus \{r\}$. These structures implicitly forbid dangling pointers. They are consistent in the sense of (6).

The second idiom (which is used in this work) defines U as the whole carrier set of references. This means that U contains an unallocated part as well, similar to a free-list. The unallocated part is represented as a single summary node in abstract shape graphs. The domain of the heap is captured with a predicate inH that is updated by memory allocation and release. The correspondence to the algebraic heap model is obvious: $\mathsf{inH}(r)$ iff $r \in \mathsf{dom}(H)$.



(a) alloc (b) init

Fig. 15: Allocation

The second idiom allows us to relate the domains of the initial heap of a program run to the final heap. For example, one can specify that a program allocates at most one reference. To this purpose, a second predicate inH_0 stores the initial domain. Allocation updates inH, but leaves inH_0 unchanged.

Some formulas must now take the heap domain into account, in particular, step must be defined as

$$\mathsf{step}(r_1, r_2) \quad \leftrightarrow \quad \mathsf{inH}(r_1) \wedge \mathsf{inH}(r_2) \wedge \bigvee_{s \in \mathsf{sel}} s(r_1, r_2)$$

A reference of type T is allocated by statements $\mathsf{x} := \mathbf{new}\ \mathsf{T}$ in two steps[3] (*alloc* and *init*), as shown in Fig. 15. A special predicate new is pointed to one of the unallocated nodes. This node is then materialized, assigned to x and initialized. The labels on the nodes indicate whether inH holds for them. Predicate T labels the newly allocated node with the correct type (see Sec. 4.3).

The technical details are somewhat intricate: concretely, *alloc* is implemented by the update formula

$$\varphi^{\mathsf{new}}_{alloc}(r) \equiv \left(\neg\ \mathsf{inH}(r) \wedge \frac{1}{2} \right)$$

The formula $\frac{1}{2}$ ensures that new is indefinite afterwards, i.e., we retain the possibility to *differentiate* between nodes r with $\neg\mathsf{inH}(r)$ - remember that we want to pick exactly one. However, note that the result shown in Fig. 15 is actually weaker than desired, as new may be 0 in some concrete structure (see pre_{init} below).

For *init*, we specify the focus formula $\mathsf{new}(r)$, and at least the following update formulas. Further initialization may be required, e.g., for reflexive predicates.

$$\begin{aligned}
\varphi^{\mathsf{x}}_{init}(r) &\equiv \mathsf{new}(r) \\
\varphi^{\mathsf{new}}_{init}(r) &\equiv 0 \\
\varphi^{\mathsf{T}}_{init}(r) &\equiv \mathsf{T}(r) \vee \mathsf{new}(r) \\
\varphi^{\mathsf{inH}}_{init}(r) &\equiv \mathsf{inH}(r) \vee \mathsf{new}(r)
\end{aligned}$$

[3] Several steps are necessary, because TVLA allows materializations only before updates, however, here we assign to new first and then materialize.

If desired, infinite memory can be specified by requiring that both parts of the split are always nonempty:

$$pre_{init} \equiv (\exists r.\ \mathsf{new}(r)) \ \wedge \ (\exists r.\ \neg\ \mathsf{new}(r) \wedge \neg\ \mathsf{inH}(r))$$

The first conjunct ensures that this allocation succeeded (new points to some node after focus). The second ensures that free memory will be available in the future, i.e., the summary node previously contained strictly *more* than one concrete node.

4.2 Formal Translation and Proof Obligations

In KIV the semantics of a specification $Spec = (\Sigma, Ax)$ with many-sorted signature $\Sigma = (S, F, P)$ and axioms Ax is the class of all algebras

$$\mathcal{A} = ((A_s)_{s\in S}, (f^{\mathcal{A}})_{f\in F}, (p^{\mathcal{A}})_{p\in P}))$$

with carrier sets A_s for every sort $s \in S$, functions $f^{\mathcal{A}}$ for $f \in F$ and predicates $p^{\mathcal{A}}$ for $p \in P$ that satisfies the axioms (see e.g. [11] Chapter 10 for an introduction to many-sorted logic).

A valuation $v = w + z$ maps program variables (w part) and logical variables (z part) to appropriate elements of the carrier sets. In particular $w(H)$ for the heap H is a finite partial function $\mathcal{H} \in A_{\mathsf{heap}}$ from references to objects.[4]

The corresponding signature used in TVLA contains predicate symbols s for every selector function $.s$, a unary predicate x for every program variable and predicates q for heap dependent predicates from KIV (like btree) dropping the heap parameter. All other arguments of these predicates are of reference type.

A pair of an algebra $\mathcal{A}$ and valuation w for program variables can be translated to an untyped model $\mathcal{U} := \rho(\mathcal{A}, w)$ of TVLA by a homomorphism ρ. The carrier set U of $\mathcal{U}$ is defined by the carrier set of the reference sort

$$U := A_{\mathsf{ref}} \setminus \{\mathsf{null}^{\mathcal{A}}\} = \Big(\bigcup_{s\in S} A_{\mathsf{ref}[s]}\Big) \setminus \{\mathsf{null}^{\mathcal{A}}\}.$$

and, as previously stated, the domain of the heap is captured in the predicate inH with the interpretation

$$\mathsf{inH}^{\mathcal{U}}(a) \quad \text{iff} \quad a \in \mathsf{dom}^{\mathcal{A}}(w(H))$$

for all $a \in U$. The interpretation of predicates x (for program variables) is derived from the valuation w as well

$$x^{\mathcal{U}}(a) \quad \text{iff} \quad a = w(x)$$

Note that $x^{\mathcal{U}}(a)$ is false if $w(x) = \mathsf{null}^{\mathcal{A}}$ as desired by the TVLA encoding, because $a \in U$ are considered only.

For the sake of simplifying the translation, we introduce additional predicates in KIV for each selector. We could also encode this step directly into the translation. The predicates are defined as

$$s(r_1, r_2, H) \leftrightarrow (r_1 \in \mathsf{dom}(H) \wedge H(r_1).s = r_2 \wedge r_2 \neq \mathsf{null})$$

Selector predicates s and heap-dependent predicates q are then uniformly interpreted as

$$s^{\mathcal{U}}(\underline{a}) \quad \text{iff} \quad s^{\mathcal{A}}(\underline{a}, w(H))$$
$$q^{\mathcal{U}}(\underline{a}) \quad \text{iff} \quad q^{\mathcal{A}}(\underline{a}, w(H))$$

The semantic translation ρ of algebras corresponds to a syntactic translation τ of TVLA formulas to KIV formulas.

Atomic formulas are translated inversely to ρ:

$$\tau(\mathsf{inH}(r)) := r \in \mathsf{dom}(H)$$
$$\tau(x(r)) := (x = r)$$
$$\tau(s(\underline{r})) := s(\underline{r}, H)$$
$$\tau(q(\underline{r})) := q(\underline{r}, H)$$

Formulas with connectives $\bullet \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ and quantifiers $\mathcal{Q} \in \{\forall, \exists\}$ are translated recursively:

$$\tau(\neg\varphi) := \neg\tau(\varphi)$$
$$\tau(\varphi \bullet \psi) := \tau(\varphi) \bullet \tau(\psi)$$
$$\tau(\mathcal{Q}\ \underline{r}.\varphi) := \mathcal{Q}\ \underline{r \neq \mathsf{null}}.\tau(\varphi)$$

Theorem 1 (Formula Translation) *The semantics of a TVLA formula φ is preserved by the translation:*

$$\rho(\mathcal{A}, w), z \models \varphi \quad \textit{iff} \quad \mathcal{A}, w + z \models \tau(\varphi) \tag{11}$$

Proof By structural induction on the formula φ. □

Therefore to prove that a formula φ is valid in TVLA we prove $\tau(\varphi)$ in KIV.

The semantics of statements *stm* in KIV is a binary relation $[\![stm]\!](w + z, w' + z)$ that maps valuations of program variables w to modified valuations w'.

Then, $\mathcal{A}, w + z \models \mathbf{wp}(stm, \varphi)$ implies that *stm* is guaranteed to terminate when started in $w + z$, and for all w' such that $[\![stm]\!](w+z, w'+z)$ holds: $\mathcal{A}, w'+z \models \varphi$.

The corresponding transition in TVLA maps structures $\mathcal{U} = \rho(\mathcal{A}, w)$ to modified structures $\mathcal{U}'$. As the carrier set U is static, $\mathcal{U}'$ is characterized uniquely by the new values of all predicates which in turn are specified in terms of the update formulas.

Definition 1 (PO for Update Formulas) *For predicate p, the proof obligation for soundness of an update formula φ^p_{stm} is:*

$$\begin{aligned} &H_0 = H \wedge \underline{x}_0 = \underline{x} \\ &\wedge\ \tau(INV) \wedge \tau(pre_{stm}) \wedge \tau(\psi_{stm}) \\ &\quad \rightarrow \mathbf{wp}(stm,\ \tilde{\tau}(\varphi^p_{stm}(\underline{r}))\ \leftrightarrow\ \tau(p(\underline{r}))) \end{aligned} \tag{12}$$

[4] Formally, in KIV, the sort heap is defined as a non-free data-type with finite partial functions as models. There is a mixfix apply function .[.] : heap × ref → object, and an element predicate . ∈ . ⊆ ref × heap. For simplicity, we write $H(r)$ instead of $H[r]$.

where $H_0, \underline{x}_0$ are fresh logical variables that store the initial state and

$$\tilde{\tau}(\varphi) := \tau(\varphi)^{H_0,\underline{x}_0}_{H,\underline{x}}$$

substitutes these variables into the translated formula, i.e., $\tilde{\tau}$ translates to a formula that refers to the initial state before the statement stm.

Proof obligation (12) ensures that an update formula evaluated in the old heap H_0 and old values of program variables $\underline{x}_0$ must equal the instrumentation predicate evaluated in the heap H after *stm* has been executed. Note that the proof obligation may assume the global invariants *INV* established by the guard strategy as well as the guard ψ_{stm} itself and the transition's precondition pre_{stm} to yield stronger update formulas.

Theorem 2 (Statement Translation)
If $[\![stm]\!](w+z, w'+z)$, $\mathcal{U} = \rho(\mathcal{A}, w)$ is transformed into $\mathcal{U}'$ by stm in TVLA, and (12) holds for all predicates p, then $\mathcal{U}' = \rho(\mathcal{A}, w')$.

Proof We show that (12) implies for all w, w', z

$$\rho(\mathcal{A}, w'), z \models p(\underline{r}) \quad \text{iff} \quad \rho(\mathcal{A}, w), z \models \varphi^p_{stm}(\underline{r}) \qquad (13)$$

The theorem then follows directly by (9).

Proof obligation (12) shows that for all valuations of program variables w, w' and valuations of logical variables z with $w(H) = z(H_0)$, $w(\underline{x}) = z(\underline{x}_0)$, and $[\![stm]\!](w+z, w'+z)$:

$$\mathcal{A}, w'+z \models \tilde{\tau}(\varphi^p_{stm}(\underline{r})) \quad \text{iff} \quad \mathcal{A}, w'+z \models \tau(p(\underline{r})) \quad (14)$$

Thus, choosing arbitrary w, w', z and setting $z_0 := z\{H_0, \underline{x}_0 \mapsto w(H), w(\underline{x})\}$:

$$\begin{array}{lll}
 & \rho(\mathcal{A}, w'), z \models p(\underline{r}) & \\
\text{iff} & \rho(\mathcal{A}, w'), z_0 \models p(\underline{r}) & \text{by (C1)} \\
\text{iff} & \mathcal{A}, w'+z_0 \models \tau(p(\underline{r})) & \text{by (11)} \\
\text{iff} & \mathcal{A}, w'+z_0 \models \tilde{\tau}(\varphi^p_{stm}(\underline{r})) & \text{by (14)} \\
\text{iff} & \mathcal{A}, w+z_0 \models \tilde{\tau}(\varphi^p_{stm}(\underline{r})) & \text{by (C2)} \\
\text{iff} & \mathcal{A}, w+z_0 \models \tau(\varphi^p_{stm}(\underline{r})) & \text{by (S1)} \\
\text{iff} & \mathcal{A}, w+z \models \tau(\varphi^p_{stm}(\underline{r})) & \text{by (C3)} \\
\text{iff} & \rho(A, w), z \models \varphi^p_{stm}(\underline{r}) & \text{by (11)}
\end{array}$$

C1,C2,C3 are applications of the coincidence lemma to change the valuation of non-free variables. Here, $H, \underline{x} \notin \text{free}(p(\underline{r}))$, $H, \underline{x} \notin \text{free}(\tilde{\tau}(\varphi^p_{stm}(\underline{r})))$, and $H_0, \underline{x}_0 \notin \text{free}(\tau(\varphi^p_{stm}(\underline{r})))$.

S1 is an application of the substitution lemma that allows to use a different variable with the same valuation. Here, $w(H) = z(H_0)$, and $w(\underline{x}) = z(\underline{x}_0)$. □

Definition 2 (PO for Guards) *The proof obligation for a guard ψ_{stm} is*

$$\tau(INV) \wedge \tau(pre_{stm}) \wedge \tau(\psi_{stm}) \rightarrow \mathbf{wp}(stm, \tau(INV)) \qquad (15)$$

Theorem 3 (Invariants)
If $[\![stm]\!](w+z, w'+z)$, $\mathcal{U}$ is transformed into $\mathcal{U}'$ by stm in TVLA, (15) holds, $\mathcal{U} = \rho(\mathcal{A}, w)$, $\mathcal{U}' = \rho(\mathcal{A}, w')$ and $\mathcal{U} \models INV$, then $\mathcal{U}' \models INV$.

Proof By (11) and the semantics of **wp**. □

A corollary of Theorem 3 is that *INV* can be extended by KIV axioms, and by formulas proven valid in KIV (for all v: $\mathcal{A}, v \models \tau(\varphi)$).

4.3 Shape Abstraction for Multiple Types

The logic of parametric shape analysis is untyped. It serves well for traditional examples with a single node type, such as linked lists. The B^+ tree specification given in Sec. 2, however, consists of multiple types. We identify them with unary predicates symbols T_i for each reference type and predicate symbols C_i for each object constructor in the algebraic model: Node, Leaf, Branch, LEntry and BEntry. Note that only one predicate is required if some $\mathsf{T}_i = \mathsf{C}_j$.

In our first attempts, we used these type predicates as abstraction predicates, leading to structures similar to Fig. 16, where objects of different types are always kept apart. Note that the structure precisely determines the selectors applicable on each object. However, in this approach, each subtree consists of four summary nodes. These became too costly to maintain when the number of separate subtrees increases, as required by the proofs for balance and sorting.

Therefore, we allow nodes of different types to be combined, as shown in Fig. 17. As the type system is not represented inside structures anymore, we must explicitly recover type information when the structure is materialized in TVLA. For example, if root is a Branch, then after the execution of x := root.next TVLA must be able to conclude that x points to a BEntry. In general, the rule for a selector $\mathsf{s} : \mathsf{T}$ defined by constructor C is

$$\mathsf{C}(r_1) \wedge \mathsf{s}(r_1, r_2) \rightarrow \mathsf{T}(r_2)$$

Furthermore, an object's type is uniquely defined. For two types $\mathsf{T}_1 \neq \mathsf{T}_2$ this is expressed by

$$\mathsf{T}_1(r) \rightarrow \neg \mathsf{T}_2(r)$$

A constructor C determines its type T (if we assume unique constructor names):

$$\mathsf{C}(r) \rightarrow \mathsf{T}(r)$$

Finally, a program variable x: T may only store references of the declared type

$$\mathsf{x}(r) \rightarrow \mathsf{T}(r)$$

The constraints imitate the global invariant of well-typedness. Their preservation is already established in KIV by type-checking, so no guards are required.

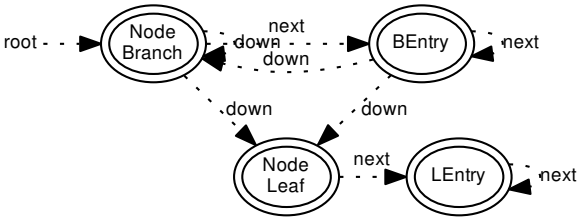


Fig. 16: Separate summary nodes for each type

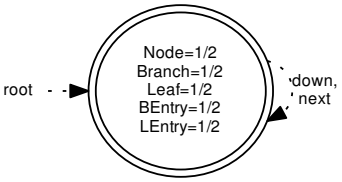


Fig. 17: Combined summary nodes

The semantic mapping is extended to these type predicates as well:

$$\mathsf{T}^{\mathcal{U}}(a) \quad \text{iff} \quad a \in A_{\mathsf{ref[T]}}$$
$$\mathsf{C}^{\mathcal{U}}(a) \quad \text{iff} \quad w(H)(a) \text{ is of the summand } \mathsf{C}$$

4.4 Contracts

So far we have described translation of models, formulas, and statements. This section completes the mapping by a description how programs are proven correct with respect to a given contract using shape analysis. The contracts we are interested in the case study have the form

$$\tau(INV) \wedge \tau(\varphi) \rightarrow \mathbf{wlp}(\alpha, \tau(\psi))$$

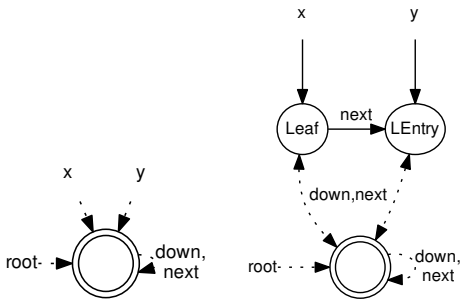
and specify partial correctness with respect to precondition φ and postcondition ψ for program α. The weakest *liberal* precondition is used, because TVLA shows partial correctness only.[5]

TVLA expects the initial state of the analysis to be given explicitly as a set of shape graphs. It is the user's responsibility to ensure that this set represents at least all concrete structures that fulfil the precondition which is usually given as a formula, i.e., the shape graphs should represent the set of concrete structures $\{\mathcal{U} \mid \mathcal{U} \models \varphi\}$.

A manual approach is feasible only for very limited examples: initializing some core or instrumentation predicate to the wrong values, e.g. 1 instead of $\frac{1}{2}$, leads to an inconsistent initial state.

An alternative is to start from very general structures, such as the one shown in Fig. 17 and filter out the uninteresting ones by assume φ operations. The advantage is that general initial structures are relatively easy to construct by hand and can be shared between all contracts.

[5] *Progress monitors* as found in [16] can be used to show termination and are an interesting aspect for future work.



(a) before assume (b) after assume

Fig. 18: Establishing precondition x.next = y

The assume operation can be implemented in principle in TVLA by a transition with the focus formula φ and precondition $pre \equiv \varphi$. The idea is to separate the "good" structures from the "bad" ones by case distinction and then continue only for those that fulfill φ. However, the materialization algorithm implemented by TVLA only supports focus formulas with specific patterns. A precise approximation of preconditions is in fact only computable for some classes of formulas [23].

Fortunately, it is possible to establish all subroutine preconditions of the B$^+$ tree case-study in this way. Most of them are rather simple, for example

$$\mathsf{x} \neq \mathsf{null} \wedge \mathsf{x.next} = \mathsf{y} \tag{16}$$

where x : Leaf and y : LEntry. Starting from Fig. 17, (16) is established by first pointing x and y to an arbitrary node (of the correct type), analogous to the first step of allocation and then assuming the formula. The result is shown in Fig. 18. Note that it is possible and more efficient to assume conjuncts of a precondition individually one after another.

The analysis proves for a postcondition ψ and all final structures $\mathcal{U}'$ that $\mathcal{U}' \models \psi$ holds. This corresponds to an operation assert ψ with guard ψ (which must not evaluate to 0 or $\frac{1}{2}$).

5 Formalization and Verification of B$^+$ Tree Invariants

In this section, we formalize B$^+$ tree invariants. We start with the intuitive definitions as used in KIV and adapt them to shape analysis by using instrumentation predicates, consistency rules, guards and update formulas. We focus on critical aspects, so this section is not exhaustive – additional instrumentation predicates and constraints are often required to achieve a precise analysis result.

The B^+ tree invariants are collected in the predicate btree, the set of keys elts(r) that a B^+ tree with root r represents is axiomatized as shown below. The predicate root restricts the heap to contain only the tree pointed to by r. Predicates in this section have an implicit heap parameter H and $H(r)$.sel is abbreviated as r.sel.

$$\begin{aligned}&\mathsf{btree}(r,[r_1,\ldots,r_m]) \;\leftrightarrow \\ &\quad \mathsf{root}(r)\wedge\mathsf{tree}(r)\\ &\wedge\forall r'.\quad \mathsf{reachable}(r,r')\rightarrow \mathsf{balanced}(r')\wedge\mathsf{sorted}(r')\\ &\qquad \wedge r'\notin\{r_1,\ldots,r_m\}\rightarrow\mathsf{oksize}(r')\end{aligned} \tag{17}$$

where

$$\begin{aligned}&\mathsf{root}(r)\leftrightarrow\forall r'.\ \mathsf{reachable}(r,r')\\ &k\in\mathsf{elts}(r)\leftrightarrow\exists r'.\ \mathsf{reachable}(r,r')\wedge\mathsf{LEntry}(r')\wedge r'.\mathsf{key}=k\end{aligned}$$

Predicate btree has an optional list of nodes $r_1,\ldots,r_m$ whose size may be out of bounds, which is used in contracts of restructuring subroutines.

Quantifiers range over allocated references, and by convention, free variables are universally quantified. The following subsections specify predicates tree (has tree shape), balanced and sorted (tree is balanced and sorted), elts and oksize, and shows the difficulties of encoding them in TVLA.

5.1 Tree Shape

We characterize trees as follows: a node is the root of a tree if there is at most one path from the root to every node in that tree. A path starts with some reference r_1 and follows a sequence of applicable selectors xs : list[sel] to another reference r_2 ([ ] denotes the empty list, + list concatenation).

$$\begin{aligned}\mathsf{tree}(r)\leftrightarrow r\neq\mathsf{null}\wedge\forall x_1,x_2,r_1,r_2.\\ \mathsf{path}(r,x_1,r_1)\wedge\mathsf{path}(r,x_2,r_2)\\ \rightarrow(x_1=x_2\leftrightarrow r_1=r_2)\end{aligned}$$

$$\begin{aligned}\mathsf{path}(r_1,[\,],r_2)&\leftrightarrow r_1\neq\mathsf{null}\wedge r_1=r_2\\ \mathsf{path}(r_1,s+xs,r_2)&\leftrightarrow r_1\neq\mathsf{null}\wedge\\ &\qquad\mathsf{path}(r_1.s,xs,r_2)\wedge\mathsf{wt}(H(r_1),s)\end{aligned}$$

These definitions serve as an intuitive formalization and are used in the algebraic specification for various consistency proofs. For shape analysis though, an alternative characterization is required that does not use recursive definitions or an explicit representation of paths. We employ the guard strategy, as the algorithms preserve tree shape in all intermediate structures. It is sufficient to prohibit cyclic and converging paths in general, similar to [16]. Converging paths are excluded by consistency rules (18) and (19), cycles are excluded by (20), forming the global invariant for tree shape. Guard (21) is used for assignments x.sel := y: the first conjunct prevents the introduction of a new cycle $\mathsf{x}\cdot\mathsf{y}\cdots\mathsf{x}$ and the second conjunct ensures that y does not already have a parent.

$$r_1.\mathsf{next}=r_2.\mathsf{down}\rightarrow r_1.\mathsf{next}=\mathsf{null} \tag{18}$$

for $s\in\{\mathsf{next},\mathsf{down}\}$:

$$r_1.s=r_2.s\wedge r_1.s\neq\mathsf{null}\rightarrow r_1=r_2 \tag{19}$$

$$r_1.s=r_2\rightarrow\neg\ \mathsf{reachable}(r_2,r_1) \tag{20}$$

$$\neg\ \mathsf{reachable}(\mathsf{y},\mathsf{x})\wedge\neg\ \exists r.\ \mathsf{reachable}(r,\mathsf{y})\wedge r\neq\mathsf{y} \tag{21}$$

We have proven that these constraints are equivalent to $\forall r.\ \mathsf{tree}(r)$ under the assumption that there is some r with root(r). The latter is an instrumentation predicate that shape analysis can prove easily to be true in the final states of the subroutines. root(r) cannot be used as an invariant, since routines like split_node have intermediate states where the tree is split into several parts.

5.2 Balance

We characterize balance as follows: A B^+ tree is balanced if each node fulfills the constraint that its down successor is the root of a subtree of height one less than the subtree of its next successor. The height of a node is determined by the maximum number of down selectors on a path to a leaf starting at that node.

$$\mathsf{height}(r)=\begin{cases}0 & \text{if } r=\mathsf{null}\\ \max[\mathsf{height}(r.\mathsf{next}), & \text{otherwise}\\ \quad\mathsf{height}(r.\mathsf{down})+1] & \end{cases} \tag{22}$$

$$\begin{aligned}\mathsf{balanced}(r)\leftrightarrow\ & r.\mathsf{next}\neq\mathsf{null}\wedge r.\mathsf{down}\neq\mathsf{null}\\ &\rightarrow\mathsf{height}(r.\mathsf{next})=\mathsf{height}(r.\mathsf{down})+1\end{aligned} \tag{23}$$

This as well as the following definitions assume that tree(r) holds for all relevant references r. Note that for arbitrary heaps with cyclic structures these definitions would be inconsistent.

These definitions are hard to reproduce in shape analysis as they are based on arithmetic. Therefore we use different definitions in TVLA based on two binary predicates eqh ("equal height") and olh ("one-less height") defined by (24) and (25) that do *local* comparisons of heights. (26) is a definition of balanced in terms of these predicates that can be proven to be equivalent to (23).

$$\mathsf{eqh}(r_1,r_2)\leftrightarrow\mathsf{height}(r_1)=\mathsf{height}(r_2) \tag{24}$$

$$\mathsf{olh}(r_1,r_2)\leftrightarrow\mathsf{height}(r_1)+1=\mathsf{height}(r_2) \tag{25}$$

$$\begin{aligned}\mathsf{balanced}(r)\leftrightarrow\quad & (r.\mathsf{next}\neq\mathsf{null}\rightarrow\mathsf{eqh}(r.\mathsf{next},r))\\ &\wedge\ (r.\mathsf{down}\neq\mathsf{null}\rightarrow\mathsf{olh}(r.\mathsf{down},r))\end{aligned} \tag{26}$$

As the height function is not available during shape analysis, eqh and olh must be specified as core predicates. Several constraints compensate the missing definitions and propagate height comparisons (transitively) between nodes, such as $\mathsf{eqh}(r_1, r_2) \rightarrow \neg\mathsf{olh}(r_1, r_2)$ and $\mathsf{olh}(r_1, r_3) \wedge \mathsf{olh}(r_2, r_3) \rightarrow \mathsf{eqh}(r_1, r_2)$.

Specified as core predicates, eqh and olh do not automatically reflect changes to the height of nodes arising from modifications, so they must be updated explicitly. The critical statements are null assignments to selectors.

We demonstrate the strategy for x.next := null. There are two cases:

If x.down = null the height of x is reduced to one, possibly changing the heights of its ancestors, too. We model this by forgetting the height relations of the affected nodes.

If x.down is non-null and x is balanced, then the height of x remains unchanged. This implies that the height of all other nodes r, in particular its ancestors, is unaffected too. This is expressed by lemma (27), which implies that relative comparisons eqh are also unaffected (second case of (28)).

$$\begin{aligned}&\mathsf{x.down} \neq \mathsf{null} \wedge h = \mathsf{height}(r) \wedge \mathsf{tree}(r) \wedge \mathsf{balanced}(r)\\ &\quad \rightarrow \mathbf{wp}\,(\mathsf{x.next} := \mathsf{null},\ h = \mathsf{height}(r))\end{aligned} \tag{27}$$

Formula (28) shows the update of eqh. We ensure that x is actually balanced with an appropriate guard.

$$\varphi^{\mathsf{eqh}}_{\mathsf{x.next}\,:=\,\mathsf{null}}(r_1, r_2) \equiv \begin{cases} \frac{1}{2} & \text{if } \mathsf{reachable}(r_1, \mathsf{x}) \\ & \quad \vee\, \mathsf{reachable}(r_2, \mathsf{x}) \\ & \text{and } \mathsf{x.down} = \mathsf{null} \\ \mathsf{eqh}(r_1, r_2) & \text{otherwise} \end{cases} \tag{28}$$

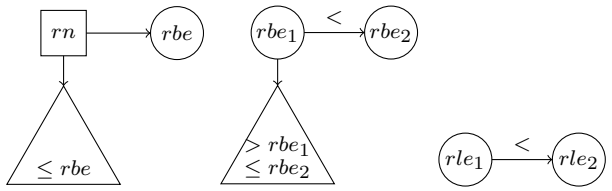
Updates for the first case immediately destroy balance information at ancestors. To avoid this problem, the statements are rearranged to ensure that no ancestor is affected at all by first detaching the node in question from its parent. For example, the underlined statement r1.next := null in Fig. 5 must be executed before the assignment to r2.down and is thus placed somewhere at the start of the function.

Height information is recovered when the first child is attached to a node: the height of a node with exactly one non-null selector is determined by its (single) child, as expressed by the following constraints:

$$\begin{aligned}&r.\mathsf{down} = \mathsf{null} \wedge r.\mathsf{next} \neq \mathsf{null} \rightarrow \mathsf{eqh}(r.\mathsf{next}, r)\\ &r.\mathsf{down} \neq \mathsf{null} \wedge r.\mathsf{next} = \mathsf{null} \rightarrow \mathsf{olh}(r.\mathsf{down}, r)\end{aligned}$$

5.3 Sorting

A B$^+$ tree is sorted if all of its nodes obey the constraints graphically given in Fig. 19. Sorting is maintained by a combination of the instrumentation strategy and guard



(a) Branches (b) Branch Entries (c) Leaf Entries

Fig. 19: Sorting Constraints. Branch-nodes are shown as boxes, entries as circles.

strategy. We abstract from the keys by directly comparing nodes, using abbreviations

$$r_1 \prec_{\mathsf{k}} r_2 \ \leftrightarrow r_1.\mathsf{key} \prec r_2.\mathsf{key} \quad \text{for } \prec \in \{<, \leq\}$$

The constraints shown in Fig. 19 are interpreted as follows: for every node rn all down-successors r (in the triangular subtree) must have smaller keys than $rbe = rn.\mathsf{next}$. For branch entries rbe_1, these nodes r must additionally have greater keys than rbe_1 itself.

This relationship is captured in an instrumentation predicate ok, defined as

$$\mathsf{ok}(r_1, r_2) \leftrightarrow \begin{cases} r_2 \leq_{\mathsf{next}} r_1 & \text{if } \mathsf{Branch}(r_1) \\ r_1 <_{\mathsf{k}} r_2 \wedge r_2 \leq_{\mathsf{next}} r_1 & \text{if } \mathsf{BEntry}(r_1) \end{cases}$$

where $\leq_{\mathsf{next}}$ is another instrumentation predicate that stores the relation of down-successors to the next entry. Predicate $\leq_{\mathsf{next}}$ hides the access to the next entry and is defined as

$$r_2 \leq_{\mathsf{next}} r_1 \ \leftrightarrow\ (r_1.\mathsf{next} \neq \mathsf{null} \rightarrow r_2 \leq_{\mathsf{k}} r_1.\mathsf{next})$$

The invariant that the B$^+$ tree is sorted is preserved in all intermediate states and formalized by the consistency rule

$$\neg\, \mathsf{Leaf}(r_1) \wedge \mathsf{reachable}(r_1.\mathsf{down}, r_2) \rightarrow \mathsf{ok}(r_1, r_2) \tag{29}$$

For statements x.next := y the guard for (29) is (30), the guards for other selector assignments are similar.

$$\begin{aligned}&\mathsf{x} <_{\mathsf{k}} \mathsf{y}\\ &\wedge\, (\forall r.\ \mathsf{reachable}(\mathsf{x.down}, r) \ \rightarrow r \leq_{\mathsf{k}} \mathsf{y})\\ &\wedge\, (\forall r_1, r_2.\ \mathsf{reachable}(r_1.\mathsf{down}, \mathsf{x}) \wedge \mathsf{reachable}(\mathsf{y}, r_2)\\ &\qquad\qquad \rightarrow r_2 \leq_{\mathsf{next}} r_1)\end{aligned} \tag{30}$$

The first conjunct ensures that keys in the linked list of entries remain ordered. The second conjunct checks elements in the down subtree of x to conform to y. The third conjunct checks that nodes in the attached subtree conform to all ancestors r with a down-pointer towards x, where $\mathsf{reachable}(r.\mathsf{down}, \mathsf{x})$ determines these ancestors The guard for x.down := y is similar.

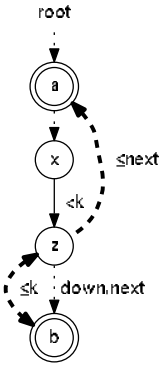


Fig. 20: starting structure

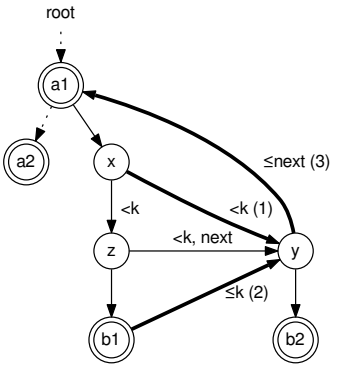


Fig. 21: after materializations

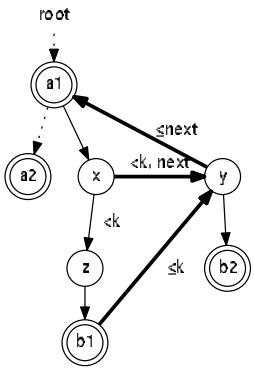


Fig. 22: after modification

The hard problem in TVLA is to ensure that the guard definitely holds when such statements are executed. Fig. 20 to 22 show the execution of the typical sequence y := z.next; z.next := null; x.next := y, starting in the state shown in Fig. 20. The critical relations are depicted as thick arrows in Fig. 21, each corresponds to one of the conjuncts. When these relations evaluate to definite values, the guard holds, as shown in Fig. 22.

The first conjunct $\mathsf{x} <_\mathsf{k} \mathsf{y}$ follows by transitivity over z and is established in Fig. 21. To derive the other two conjuncts, we materialize nodes r such that $\mathsf{reachable}(\mathsf{z.next}, r)$ when executing y := z.next, and we materialize nodes r such that $\mathsf{reachable}(r.\mathsf{down}, \mathsf{x})$ when executing z.next := null. These have the effect of splitting b and a into b1,b2 and a1,a2 respectively.

b1 now represents the subtree that must be checked in the second conjunct and a1 gives exactly the ancestors that are covered by the third conjunct. The necessary relations are then derived from the sorting invariant *before* the statement z.next := null is executed, explicitly stored in the structure and thus available when the guard is evaluated.

Note that in order to prevent nodes that are materialized from being merged back, we have to employ several derived abstraction predicates, e.g., $\mathsf{reachable\text{-}from\text{-}x}(r) \leftrightarrow \mathsf{reachable}(\mathsf{x}, r)$ for program variables x. Deriving unary (reachability) predicates from binary ones with respect to program variables is a common idiom in TVLA.

5.4 Elements and Keys

In [19], the set elts of elements a pointer structure represents is tracked by explicitly labeling objects whose key is in the set in the initial state with an additional (core) predicate. The final state is then related to this predicate.

This formalization has the drawback that leaf entries must be kept distinct from other objects (compare Sec. 4.3), so that $rle.\mathsf{key} \in \mathsf{elts}(r)$ always yields definite values. This would essentially double the number of nodes in structures.

Instead, we mark a leaf entry when its key is assigned with a predicate keychanged, which is initialized false for all nodes. For insert_leaf we establish that it allocates at most one leaf entry, and changes no key of an existing node. This is expressed as postcondition (31), where H_0 and H are the initial and the final heap. The modifications of elts can be derived from this condition in KIV. A similar postcondition is proved for delete_leaf.

$$\begin{aligned} \exists\, rle.\quad & H[rle].\mathsf{key} = k \\ & \wedge\, \mathsf{dom}(H) = \mathsf{dom}(H_0) \cup \{rle\} \\ \wedge\ \forall\, rle_1.\neg & \mathsf{keychanged}(rle_1, H_0, H) \end{aligned} \tag{31}$$

where

$$\begin{aligned} & \mathsf{keychanged}(r, H_0, H) \leftrightarrow \\ & \quad r \in \mathsf{dom}(H_0) \rightarrow H_0[r].\mathsf{key} = H[r].\mathsf{key} \end{aligned} \tag{32}$$

Since contracts (3) for insert_leaf and delete_leaf refer to a key k that is not attached to a node, we need to introduce objects of type Key to the TVLA specification. We must also be able to relate k to the keys of other nodes. This can be done by overloading predicates $\leq_\mathsf{k}$, $<_\mathsf{k}$, and ok to range over keys as well.

Predicate okpath specifies that the recursive descent has taken the correct path such that k can be inserted

in the subtree under r without violating the sorting constraints. It is defined as

$$\begin{aligned}&\mathsf{okpath}(r', k) \leftrightarrow \\ &\quad \forall r.\ \mathsf{reachable}(r.\mathsf{down}, r') \rightarrow \mathsf{ok}(r, k)\end{aligned}$$

and mirrors the sorting constraints given in Sec. 5.3.

To assign a key stored in variable k to an LEntry by the statement x.key := k we copy the sorting relations from the key to the node itself and mark x as changed. The update formulas are defined as

$$\varphi^{\leq \mathsf{k}}_{\mathsf{x.key}\ :=\ \mathsf{k}}(r_1, r_2) \equiv \begin{cases} \mathsf{k} \ \leq_{\mathsf{k}} r_2 & \text{if } r_1 = \mathsf{x} \\ r_1 \leq_{\mathsf{k}} \mathsf{k} & \text{if } r_2 = \mathsf{x} \\ r_1 \leq_{\mathsf{k}} r_2 & \text{otherwise} \end{cases}$$

$$\varphi^{\mathsf{keychanged}}_{\mathsf{x.key}\ :=\ \mathsf{k}}(r) \equiv \mathsf{keychanged}(r) \vee \mathsf{x} = r$$

and similarly for $<_{\mathsf{k}}$. The way keychanged works is similar to balanced – its definition (32) is not available in TVLA. After executing x.key := k, we know that $\mathsf{ok}(r, \mathsf{x}) \leftrightarrow \mathsf{ok}(r, k)$ for all other nodes r, which establishes (29).

5.5 Node Sizes

The size of a node rn is determined by the number of its entries r, i.e., those reachable by following next selectors only. These entries are collected in a set, extensionally defined as $r \in \mathsf{nset}(rn) \leftrightarrow \mathsf{next}^*(rn, r)$. Let N denote the rank of the B^+ tree, then

$$\begin{aligned}&\mathsf{oksize}(r) \leftrightarrow \\ &\quad (\mathsf{Node}(r) \rightarrow \\ &\qquad (\textbf{if } \mathsf{root}(r) \textbf{ then } 1 \textbf{ else } N) \leq |\mathsf{nset}(r)| \leq 2N)\end{aligned}$$

Node sizes are verified by a strategy similar to [13]. There, the sets of concrete individuals represented by summary nodes are tracked, as well as the cardinalities of these sets. [13] is an extension to TVLA that seems capable of directly verifying the node size invariant. However, the prototype implementation is not available, so we imitate the strategy. As an example, for $\mathsf{split\_node}(rp; rt)$, we prove the contract with TVLA

$$\begin{aligned}&rn = rp.\mathsf{down} \wedge \mathsf{btree}(r, [rn]) \qquad (33)\\ &\wedge \mathsf{reachable}(r, rp) \wedge e = \mathsf{elts}(r)\\ &\wedge \mathsf{median}(rn, r_1.\mathsf{next})\\ &\rightarrow\ \mathbf{wp}(\mathsf{split\_node}(rp; rt),\\ &\qquad \mathsf{btree}(r, [rp, rn, rt]) \wedge e = \mathsf{elts}(r)\\ &\qquad \wedge \mathsf{nset}(rp) = \mathsf{nset}_0(rp) \cup \{r_1.\mathsf{next}\}\\ &\qquad \wedge \mathsf{nset}_0(rn) = \mathsf{nset}(rn) \cup \{r_1.\mathsf{next}\} \cup \mathsf{nset}(rt))\end{aligned}$$

where $\mathsf{nset}_0(r)$ denotes the set of entries of r in the initial state. nset-membership is encoded as binary predicates in TVLA. From (33) we prove in KIV that if $|\mathsf{nset}_0(rn)| = 2N + 1$ then both rn and rt have now size N and satisfy oksize, implying $\mathsf{btree}(r, [rp])$.

6 Results and Experiences

To make TVLA usable as a decision procedure we had to solve two problems: the first was to bridge the gap between explicit, typed algebraic heaps specified as partial functions and the implicit view of heaps encoded as the domain of predicates defined in untyped logic. The solution caused some overhead in KIV, to support switching between the generic specification and its instance for B^+ trees. It is however a generic solution that allows us to verify the constraints shape analysis uses for generic predicates such as tree shape or acyclicity once and for all. The second problem was to determine (12) and (15) as the right proof obligations for the instrumentation and the guard strategy.

The overall effort of the case study was around six person-months. The first month was necessary to get familiar with TVLA's user interface, which is very low level. A simple script (available on the web [8]) that removes superfluous information from the output and colorizes the shape graphs was invaluable. Another script was used to generate TVLA transition systems from code.

The main task then was to translate the natural definitions of the B^+ tree invariants into suitable TVLA constraints. It roughly took three person-months to iteratively figure out the right instrumentation predicates, update formulas and consistency rules given in Sec. 5 for the B^+ tree invariants by analyzing failed TVLA proofs.

We noticed that global invariants are often easier to deal with than instrumentation predicates. On one hand, checking guards is typically more efficient than the definition of *INV* as an additional instrumentation predicate. On the other hand, it is also easier to verify guards in KIV, since update formulas for invariants tend to be rather complex, while guards can often be simplified by making them stronger than strictly necessary to prove (15).

The remaining two months were spent on setting up the KIV specifications (including the generic theory), proving correctness of update formulas/guards and the interactive proofs of the main recursion.

The main proofs for the recursive programs were easy with the lemmas established by shape analysis. The proofs use symbolic execution and induction. They are graphically displayed with a structure that follows the structure of the program (see web presentation [8]).

The predicate logic proofs for update formulas, guards and consistency rules are based on KIV's sequent calculus. Automation is achieved by rewrite rules. KIV supports conditional rewriting modulo associativity and commutativity of arbitrary operators. The rules are compiled into functional code, which runs very efficiently even for a large number of rules: the case study here uses around 2000 rules, around 1500 of these were inherited from KIV's standard library of data types. The

heap theory contributes 150 rules, while the remaining 350 are specific for B^+ trees.

Counting the interactive proof steps for the different properties shows that the formalization of node sizes is the most complex with 480 interactions, followed by tree shape (300), sorting (280) and balance (160). Typical interactions are choosing an induction variable, quantifier instantiation, or applying a lemma by clicking on a relevant formula. Occasionally, manual case splits are necessary, too.

The most expensive consistency proofs are for update formulas like (27) and guards like (30). Some of them still required some dozen interactions. This agrees with our expectations that interactive reasoning about pointer manipulations is difficult. However, we have found that these proofs are required, many of the more complex constraints we used in TVLA were initially wrong.

TVLA proofs for most of the 23 subroutines required run times below one minute on a 2.8 GHz CPU equipped with 8 Gb of main memory running 64 bit Linux. Consumption of main memory is high, usually between 500 Mb and 1 Gb, supposedly caused by the high number of predicates (around 30 binary and over 160 unary predicates). A few subroutines, such as rotations in the middle of the tree, took up to 5 minutes.

From our experience, attempting an analysis of the whole insert and delete algorithms with the final specification with TVLA seems feasible. Initial attempts, however, indicate that running TVLA on the composed code requires further optimizations. In particular, the strategy for sorting creates too many structures when traversing the full tree. We also think that it is not practical to develop the specification using TVLA on the full program, since the number of shape graphs grows rapidly with the length of the program, up to several thousands. These would have to be analyzed to find out where exactly the analysis goes wrong. For the subroutines the number was much lower, typically around one hundred.

7 Related Work

Verification of B^+ trees is a hard problem. We are aware of several efforts to verify them. Two pen-and-paper proofs have been done by Fielding [10] and Sexton et al. [22]. The first one uses two refinements with an intermediate level of nested sets. The implementation is given as Pascal code. The other one uses separation logic. Algorithms are given by transitions of an abstract machine specifically designed for the problem.

The only complete mechanized verification we are aware of is by Malecha et al. [17]. It uses a separation logic framework for the Coq theorem prover and a similar formalization as [22]. Although the authors state that a significant degree of automation was achieved by customized proof tactics, the effort is still high: approx. 5000 lines of proof script were needed. Their verification, however, considers some additional operations (e.g., efficient range queries) we have not verified.

Preliminary work in TVLA by Herter [15] verifies some properties of B^+ trees, but is restricted to a statically bounded node size. A shape analysis specification of binary trees has been developed for example by Loginov et al. [16].

Yorsh et al. [23] present an algorithm $\widehat{assume}(\varphi, S)$ that computes the least overapproximation of a subset of structures S that fulfills φ. The underlying idea is very similar to our approach given in Sec. 4.4, but $\widehat{assume}(\varphi, S)$ is only complete with respect to a first-order decision procedure.

Other automatic tools that are based on abstract interpretation/shape analysis are for example Xisa [4] and jStar [6] (in combination with Separation Logic). They might be interesting to investigate as alternatives to TVLA in the future.

8 Conclusion

We have verified an implementation of the main algorithms for B^+ trees using a combination of interactive theorem proving and automated shape analysis.

Our results indicate that the combination of both techniques is a significant improvement compared to using one approach alone. Automation using Shape Analysis has been significantly better than if we would have used KIV exclusively. Soundness of the shape analysis results would have been rather doubtful without proving the more complex constraints with an interactive theorem prover.

The case study has also shown how to bridge the gap between an abstract, typed algebraic approach used by almost all interactive theorem provers and the untyped approach of TVLA in general. Based on these results it is clear now how to implement an automated translation of KIV programs, predicates and constraints to TVLA (which remains work to do).

We must however concede that shape analysis is not as easily usable as a decision procedure would be. There is still a lot of specific knowledge of the internal working of TVLA required to define the right instrumentation predicates (for example $\leq_{\mathsf{next}}$), and (even more) to analyze failed proof attempts from TVLA. Getting meaningful counterexamples from failed proof attempts to analyze whether a proof failed since the goal was wrong or due to overapproximation is still one of the most time-consuming tasks, and a topic for further work.

Acknowledgements We thank Alexander Knapp, Axel Habermaier, and the anonymous reviewers for their valuable feedback.

References

1. R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Acta Informatica*, volume 1, pages 173–189. Springer, 1972.
2. J.C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Proc. of the 1st Int. Conf. on Interactive Theorem Proving*, ITP, pages 131–146. Springer, 2010.
3. I. Bogudlov, T. Lev-Ami, T. Reps, and M. Sagiv. Revamping TVLA: making parametric shape analysis competitive. In *Proc. of the 19th Int. Conf. on Computer Aided Verification*, CAV, pages 221–225. Springer, 2007.
4. B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *Proc. of the 35th Int. Symp. on Principles of Programming Languages*, POPL, pages 247–260. ACM, 2008.
5. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.
6. D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In *Proc. of the 23rd Int. Conf. on Object-oriented programming systems languages and applications*, OOPSLA, pages 213–226. ACM, 2008.
7. A. Dunets, G. Schellhorn, and W. Reif. Automated flaw detection in algebraic specifications. In *Journal of Automated Reasoning*. Springer, 2010.
8. G. Ernst. KIV and TVLA proofs for B^+ trees. `http://www.informatik.uni-augsburg.de/swt/projects/btree.html`, 2011.
9. G. Ernst, G. Schellhorn, and W. Reif. Verification of B^+ trees: an experiment combining shape analysis and interactive theorem proving. In *Proc. of the 9th Int. Conf. on Software Engineering and Formal Methods*, SEFM, pages 188–203. Springer, 2011.
10. E. Fielding. The specification of abstract mappings and their implementation as B^+ trees. Technical report, Oxford University, PRG-18, 1980.
11. J. H. Gallier. *Logic for computer science: foundations of automatic theorem proving*. Harper & Row Publishers, Inc., New York, NY, 1985.
12. D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *Proc. of the 32th Int. Symp. on Principles of Programming Languages*, POPL, pages 338–350. ACM, 2005.
13. S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *Proc. of the 36th Int. Symp. on Principles of Programming Languages*, POPL, pages 239–251. ACM, 2009.
14. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
15. J. Herter. Towards shape analysis of B-trees. Master's thesis, Universität Saarbrücken, 2008.
16. A. Loginov, T. Reps, and M. Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *Prof. of Static Analysis Symposium*, SAS, pages 261–279. Springer, 2006.
17. G. Malecha, G. Morrisett, A. Shinnar, and R.Wisnesky. Toward a verified relational database management system. In *Proc. of the 37th Int. Symp. on Principles of Programming Languages*, POPL, pages 237–248. ACM, 2010.
18. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume II, chapter 1, pages 13 – 39. Kluwer, Dordrecht, 1998.
19. J. Reineke. Shape analysis of sets. In *Workshop "Trustworthy Software"*. IBFI, 2006.
20. N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *Proc. of Static Analysis Symposium*, SAS, pages 284–302. Springer, 2005.
21. M. Sagiv, Reps, and Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Trans. Program. Lang. Syst.*, volume 24, pages 217–298. ACM, 2002.
22. A. Sexton and H. Thielecke. Reasoning about B^+ trees with operational semantics and separation logic. In *Electron. Notes Theor. Comput. Sci.*, volume 218, pages 355–369. Elsevier Science Publishers B. V., 2008.
23. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Proc. of the 10th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, TACAS, pages 530–545. Springer, 2004.