Verification of B$^+$ Trees: An Experiment Combining Shape Analysis and Interactive Theorem Proving

Gidon Ernst, Gerhard Schellhorn, and Wolfgang Reif

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

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". We verify the main operations of B$^+$ trees by decomposition of the problem into three layers. At the top level is an interactive proof of the main recursive procedures. The actual modifications of the data structure are verified with shape analysis. To this purpose we define a mapping of typed algebraic heaps to TVLA. TVLA itself relies on various constraints and lemmas, that were proven in KIV as a foundation for an overall correct analysis.

1 Introduction

Interactive theorem provers are powerful tools for formal verification. However, reasoning about pointer structures in the presence of destructive updates can be quite difficult with them. In contrast, fully automatic tools based on shape analysis, such as TVLA [14,2], are specifically designed to perform well in these situations, but can not deal with precise arithmetic and induction, for example.

B$^+$ trees [1] 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.

Verification of B$^+$ trees is a hard problem. We are aware of several efforts to verify them. Two pen-and-paper proofs are [4] and [15]. The first uses two refinements with an intermediate level of nested sets. The implementation is given as Pascal code. The other 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 [10]. It uses the separation logic framework of the Coq theorem prover and a similar formalization as [15]. 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 is [8]. It verifies some properties of B$^+$ trees, but is restricted to a statically bounded rank.

Our approach uses algebraic specifications and wp-calculus 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): 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.

In this work, we perform a conceptual integration by translating manually between the two worlds. We evaluate how the rather different high-level specification style used in algebraic specifications can be mapped to the shape graphs and logic of TVLA.

We use the theorem prover KIV [11], but the approach should be applicable with other interactive theorem provers, too. By combining KIV and TVLA, we have verified that our implementation of the insertion and deletion operations on B$^+$ trees maintains the invariants for tree shape, balance, sorting and node sizes 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 [3].

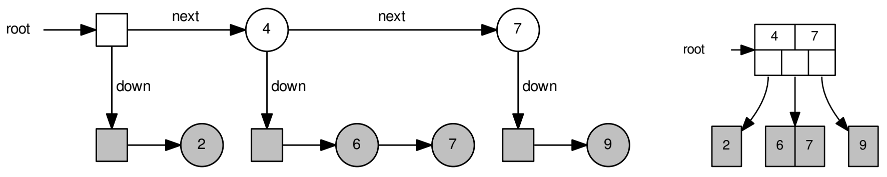
This work is organized as follows: Section 2 introduces B$^+$ trees, our verification approach and an algebraic specification of pointer structures. Section 3 briefly describes the shape analysis implemented by TVLA. Section 4 formalizes B$^+$ tree invariants and explains how these can be tracked with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

We use linked lists instead of arrays for the representation of nodes. An encoding of arrays in TVLA has been developed [5,6], but unfortunately the modified TVLA versions are not available, so we remain with the core concepts in this case study.

Figure 1 shows an example B$^+$ tree in this model, compared to an equivalent array-based one. Graphical nodes displayed as boxes serve as representatives of entire B$^+$ tree nodes (subsequently called *heads*), while the round nodes (subsequently called *entries*) store the keys. The edge labels next and down indicate the names of the corresponding selectors of the respective objects. This B$^+$ tree represents the set $\{2, 6, 7, 9\}$.



(**a**) pointer-based (**b**) array-based

Fig. 1. B^+ tree representations

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, possibly leading to growth and shrinking at the root.

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

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

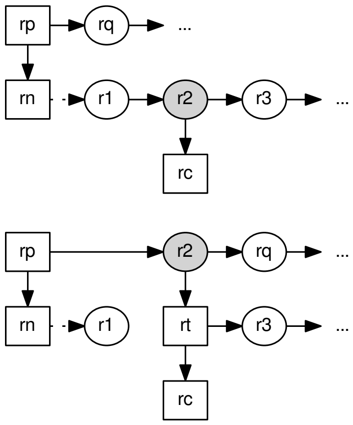
Fig. 2. Mutually recursive insertion routines

Figure 2 lists two mutually recursive subroutines of the insertion algorithm, given in the abstract program syntax used in KIV. They receive the current key as a parameter k and the current node in rn and rbe respectively – insert_node descends at a node head (displayed as boxes in Fig. 1), while insert_bentry performs similar actions at branch entries (displayed as circles). The top-level routine insert, which is not shown here, is very similar to insert_node but additionally has to deal with an empty tree as well as growth and shrinking. The actual modifications of the data structure are hidden 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. The deletion algorithm is similar but more complicated because balance may be restored either by merging nodes or by transferring keys between adjacent nodes.

Figure 3 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 (which is required to specify the contract).

```
split_node(rp ; rt) {
  rn := rp.down
  compute r1
  r2 := r1.next
  r3 := r2.next
  r1.next := null
  rc := r2.down
  rt := new Branch
  r2.down := rt
  rt.down := rc
  rt.next := r3
  rq := rp.next
  rp.next := r2
  r2.next := rq
}
```

(a) Code



(b) Structure before/after modification

Fig. 3. Restructuring routine split_node

2.2 Verification Approach

The verification of B$^+$ trees must establish two properties of the implementation: 1) insertion and deletion retain the B$^+$ tree invariants and 2) the operations correspond to their set-theoretic counterparts. Invariants are collected in a predicate btree(r) (formalized in Sec. 4) 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 elts(r) in the following. Formally,

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

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

must be proved, where $\mathbf{wp}(\alpha, \varphi)$ denotes the weakest precondition of program α with postcondition φ. e denotes the elements of the initial tree. Programs insert and delete are called with the key k to insert or delete. The root r of the tree is passed by reference as it may change.

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. These depend on shape analysis to verify subroutines that perform actual modifications to the data structure, such as split_node, split_bentry and insert_leaf, forming the intermediate layer. 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 are performed by symbolic execution of the program source code. KIV implements the wp-calculus in the form of Dynamic Logic [7], 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.

Subroutines can be classified into restructuring, such as split_node, and content modifications consisting of insert_leaf and delete_leaf. There are 16 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)[1] and (4) where r denotes the root of the tree. In the precondition, $\mathsf{lok}(rl, k)$ ("leaf of key") specifies that key k actually belongs into the leaf rl which is required to establish sorting in the result.

$$\begin{aligned}
&\mathsf{btree}(r) \wedge \mathsf{reachable}(r, rp) \wedge e = \mathsf{elts}(r) \\
&\qquad \rightarrow \ \mathbf{wp}(\mathsf{split\_node}(rp; rt),\ \mathsf{btree}(r) \wedge \mathsf{elts}(r) = e) \qquad (3)\\
&\mathsf{btree}(r) \wedge \mathsf{reachable}(r, rl) \wedge e = \mathsf{elts}(r) \wedge \mathsf{lok}(rl, k) \\
&\qquad \rightarrow \ \mathbf{wp}(\mathsf{insert\_leaf}(k, rl),\ \mathsf{btree}(r) \wedge \mathsf{elts}(r) = e \cup \{k\}) \qquad (4)
\end{aligned}$$

The main proof for the insert algorithm is concerned with the mutually recursive procedures insert_node and insert_bentry (see Fig. 2). 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 $\mathsf{lok}(r.\mathsf{next}, k)$ resp. $\mathsf{lok}(r.\mathsf{down}, k)$ given that $\mathsf{lok}(r, k)$ holds for the current node r and key k – which 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 [13], which automatically computes the contracts 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 $H : \mathsf{ref}[T] \nrightarrow T$ that maps ("dereferences") allocated references $r \in \mathsf{dom}(H)$ with $r : \mathsf{ref}[T]$ to objects $o = H(r)$ of corresponding type T. 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 $o = \mathsf{Leaf}(r)$

[1] This contract ignores the node sizes, see (22) for the full contract.

then $o.\mathsf{next} = r$. Additionally, predicates such as $o.\mathsf{leaf?}$ are provided to test by which constructor an object has been built.

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

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

We assume a constant $\mathsf{null} : \mathsf{ref}$ that is never allocated in a Heap. A predicate $\mathsf{wt} : \mathsf{object} \times \mathsf{sel}$ 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. 4.1).

As opposed to TVLA, 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.

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

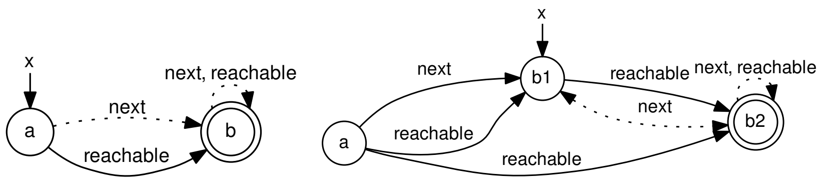
3 Introduction to Parametric Shape Analysis

Parametric shape analysis [14] is an instance of abstract interpretation. 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 logic is three-valued with the domain of truth values $\mathbb{B}_3 = \{0, \frac{1}{2}, 1\}$. The third value $\frac{1}{2}$ denotes unknown truth (sometimes called “indefinite”) and aggregates contradicting truth values in abstract states. An abstraction function maps the infinite concrete domain of possible data structures to finitely many bounded abstract structures.

The key idea is to partition objects allocated in the heap into finitely many equivalence classes, so that all objects in the same class are indistinguishable by a set of user-definable properties of interest. These properties are given by unary *abstraction predicates*. Typically, there are singleton classes for objects pointed to by program variables. Classes with more than one element are called *summary nodes*. A single summary node can represent a whole subtree, for example.

As an example, Fig. 4a shows a *shape graph* for a heap which contains a singly-linked list. Node a represents a single object, while all other memory cells are grouped



(a) original structure **(b)** after splitting b into b1 and b2

Fig. 4. Splitting of a summary node

into the (doubly circled) summary node b. The partitioning is according to the fact that program variable x points to a.

The program signature (program variables and selectors) is encoded as part of the logical signature. A program variable x becomes an unary predicate symbol $\mathsf{x}(r)$ that is constrained to hold for one node only: the one x points to. Selectors become binary predicate symbols, for example $H[r_1].\mathsf{next} = r_2$ is encoded as $\mathsf{next}(r_1, r_2) = 1$. These predicates are constrained to be partial functions. The predicate symbols arising from the program signature are called *core predicates*.

In Fig. 4a the dotted arrow indicates that $\mathsf{next}(\mathsf{a}, \mathsf{b}) = \frac{1}{2}$, since the next selector of node a points to some node in class b, but not to all. All nodes summarized by b are definitely reachable from a, as indicated by the solid arrow.

The program itself is represented by a finite transition system between states, 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 can be done by a fixpoint computation, since the number of states as well as the number of shape graphs is finite.

To compute the fixpoint, for each transition a *precondition* (`%p` in TVLA) and an *update formula* must be defined. The precondition encodes tests of conditionals or loops, the update formula must specify the effects of assignments for all core predicates.

As an example x := y is represented by the update formulas $\varphi^{\mathsf{x}}_{\mathsf{x}:=\mathsf{y}}(r) = \mathsf{y}(r)$ for x and $\varphi^{\mathsf{p}}_{\mathsf{x}:=\mathsf{y}}(r) = \mathsf{p}(r)$ for all other predicates p. A transition is executed by evaluating each update formula in the old state yielding the predicate's values in the new state. The statement x.next := y is represented as $\varphi^{\mathsf{next}}_{\mathsf{x.next}:=\mathsf{y}}(r_1, r_2) = \mathsf{next}(r_1, r_2) \vee \mathsf{x}(r_1) \wedge \mathsf{y}(r_2)$.[2] [14] defines preconditions and update formulas for standard statements such as assignments, selector access and case distinctions. All 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.

During the run of a program, the abstraction is dynamically adjusted with every statement. Suppose, the statement x := x.next should be executed in Fig. 4a and recall that each program variable should point to a singleton node: for x we have to get hold of the object a.next in this case. Parametric shape analysis splits summary nodes as necessary with an operation called *materialization*, as shown in Fig. 4b. Here, b1 = a.next is the

[2] For x : refbe, this corresponds to $H := H[\mathsf{x} \mapsto \mathsf{BEntry}(H(\mathsf{x}).\mathsf{key}, \mathsf{y}, H(\mathsf{x}).\mathsf{down})]$ in KIV.

direct successor of a and b2 represents the remaining elements of b.[3] Technically, the analysis ensures that a given set of *focus formulas* yields definitive values for all objects r. For the assignment $\mathsf{x} := \mathsf{x.next}$ the focus formulas are $\mathsf{x}(r)$ and $\exists r_1.\, \mathsf{x}(r_1) \wedge \mathsf{next}(r_1, r)$. Additional focus formulas can be given to cause extra splits of summary nodes. We will need these in Sec. 4.3.

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 reachable and step, defined by

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

are such instrumentation predicates (step* is the reflexive transitive closure of step). 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

$$\mathsf{reachable}(r_1, r_2) \rightarrow \neg\mathsf{step}(r_2, r_1)$$

The value of instrumentation predicates is explicitly stored in shape graphs. By default, executing a transition reevaluates the defining formula in the new state. However, this may lose the definite information that an instrumentation predicate stores. For the structure resulting from the assignment shown in Fig. 4b we have $\mathsf{reachable}'(\mathsf{b1}, \mathsf{b2}) = \mathsf{next}'^*(\mathsf{b1}, \mathsf{b2}) = \frac{1}{2}$. To prevent this, parametric shape analysis allows explicit update formulas for instrumentation predicates. Assuming that $\mathsf{x.sel} = \mathsf{null}$ and $\mathsf{y} \neq \mathsf{null}$ the update formula for reachable and an assignment $\mathsf{x.sel} := \mathsf{y}$ is

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

This update formula preserves definite reachability of b2 from b1. However, update formulas that do not comply with the definitions of instrumentation predicates lead to an unsound analysis. To ensure soundness, we verify in KIV that given an update formula $\varphi^{\mathsf{p}}_{stm}$ for statement *stm* and predicate p

$$INV \wedge H_0 = H \rightarrow \mathbf{wp}\left(stm, \tau(\varphi^{\mathsf{p}}_{stm})\,(r_1, \ldots, r_n, H_0) \leftrightarrow \mathsf{p}(r_1, \ldots, r_n, H)\right) \tag{5}$$

holds, i.e., the update formula evaluated in the old heap H_0 must equal the instrumentation predicate evaluated in the heap H after *stm* has been executed. Here, $\tau(\varphi)$ denotes the translation of formula φ to KIV's logic (see appendix A). Note that the proof obligation may assume the global invariants established by the guard strategy to yield stronger update formulas. To establish such global invariants, *guards* (`%message` in TVLA) are attached to transitions, that ensure that the invariant is preserved (therefore the name). TVLA stops the analysis whenever it finds that an input shape graph for a transition violates its guard. To ensure soundness, we verify in KIV

$$INV \wedge \tau(\psi) \rightarrow \mathbf{wp}(stm, INV) \tag{6}$$

for each assignment *stm* and its guard formula ψ.

[3] A second shape graph (not shown) is necessary for the special case where b1 has no successors.

When defining a global invariant is possible, then checking guards is typically more efficient than the definition of INV as an additional instrumentation predicate, that has to be tracked to be valid in all intermediate states. 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 (6), see (11) for an example.

4 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 $\mathsf{elts}(r)$ that a B$^+$ tree with root r represents is axiomatized as shown. 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 $H[r].s$ is abbreviated as $r.s$.

$$\begin{aligned} \mathsf{btree}(r, [r_1, \ldots, r_m]) \;&\leftrightarrow\; \mathsf{root}(r) \wedge \mathsf{tree}(r) \\ &\wedge \forall r'.\quad \mathsf{reachable}(r, r') \rightarrow \mathsf{balanced}(r') \wedge \mathsf{sorted}(r') \\ &\qquad\qquad \wedge\, r' \notin \{r_1, \ldots, r_m\} \rightarrow \mathsf{oksize}(r') \\ \text{where}\quad \mathsf{root}(r) \;&\leftrightarrow\; \forall r'.\, \mathsf{reachable}(r, r') \\ k \in \mathsf{elts}(r) \;&\leftrightarrow\; \exists r'.\, \mathsf{reachable}(r, r') \wedge r'.\mathsf{lentry?} \wedge r'.\mathsf{key} = k \end{aligned} \tag{7}$$

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), balance and sorted (tree is balanced and sorted), elts and oksize, and show the difficulties of encoding them in TVLA.

4.1 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).

$$\begin{aligned} \mathsf{tree}(r) \leftrightarrow r \neq \mathsf{null} \wedge \forall x_1, x_2, r_1, r_2. \\ \mathsf{path}(r, x_1, r_1) \wedge \mathsf{path}(r, x_2, r_2) \rightarrow (x_1 = x_2 \leftrightarrow r_1 = r_2) \\ \mathsf{path}(r_1, [\,], r_2) \leftrightarrow r_1 \neq \mathsf{null} \wedge r_1 = r_2 \\ \mathsf{path}(r_1, s + xs, r_2) \leftrightarrow r_1 \neq \mathsf{null} \wedge \mathsf{path}(r_1.s, xs, r_2) \wedge \mathsf{wt}(H(r_1), s) \end{aligned}$$

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 [9]. Converging paths are excluded by consistency rules (8) and (9), cycles are excluded by (10), forming the global invariant for tree shape. Guard (11) is used for assignments x.sel := y.

$$r_1.\mathsf{next} = r_2.\mathsf{down} \rightarrow r_1.\mathsf{next} = \mathsf{null} \tag{8}$$

$$r_1.s = r_2.s \wedge r_1.s \neq \mathsf{null} \rightarrow r_1 = r_2 \qquad \text{for } s \in \{\mathsf{next}, \mathsf{down}\} \tag{9}$$

$$r_1.s = r_2 \rightarrow \neg\mathsf{reachable}(r_2, r_1) \tag{10}$$

$$\neg\mathsf{reachable}(\mathsf{y}, \mathsf{x}) \wedge \neg\exists r.\ (\mathsf{reachable}(r, \mathsf{y}) \wedge r \neq \mathsf{y}) \tag{11}$$

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

4.2 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.

$$\mathsf{height}(r) = \begin{cases} 0 & \text{if } r = \mathsf{null} \\ \max\left[\mathsf{height}(r.\mathsf{next}),\ \mathsf{height}(r.\mathsf{down}) + 1\right] & \text{otherwise} \end{cases} \tag{12}$$

$$\begin{aligned} \mathsf{balanced}(r) \leftrightarrow\ & r.\mathsf{next} \neq \mathsf{null} \wedge r.\mathsf{down} \neq \mathsf{null} \\ & \rightarrow \mathsf{height}(r.\mathsf{next}) = \mathsf{height}(r.\mathsf{down}) + 1 \end{aligned} \tag{13}$$

This as well as the following definitions assume that $\mathsf{tree}(r)$ holds for all relevant references r. Note that for arbitrary heaps with cyclic structures they 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 (14) and (15) that do a *local* comparisons of heights. (16) is a definition of balanced in terms of these predicates that can be proven to be equivalent to (13).

$$\mathsf{eqh}(r_1, r_2) \leftrightarrow \mathsf{height}(r_1) = \mathsf{height}(r_2) \tag{14}$$

$$\mathsf{olh}(r_1, r_2) \leftrightarrow \mathsf{height}(r_1) + 1 = \mathsf{height}(r_2) \tag{15}$$

$$\begin{aligned} \mathsf{balanced}(r) \leftrightarrow\ & (r.\mathsf{next} \neq \mathsf{null} \rightarrow \mathsf{eqh}(r.\mathsf{next}, r)) \\ \wedge\ & (r.\mathsf{down} \neq \mathsf{null} \rightarrow \mathsf{olh}(r.\mathsf{down}, r)) \end{aligned} \tag{16}$$

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 $\mathsf{eqh}(r_1, r_2) \to \neg\mathsf{olh}(r_1, r_2)$ and $\mathsf{olh}(r_1, r_3) \wedge \mathsf{olh}(r_2, r_3) \to \mathsf{eqh}(r_1, r_2)$.

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 $\mathsf{x.next} := \mathsf{null}$. There are two cases: If $\mathsf{x.down} = \mathsf{null}$ 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 $\mathsf{x.down}$ is non-null and x is balanced, then the height of x remains unchanged. This implies that the height of all other nodes, in particular its ancestors, is unaffected too. This is expressed by lemma (17), which implies that relative comparisons eqh are also unaffected (second case of (18)).

$$\begin{aligned} &\mathsf{x.next} \neq \mathsf{null} \;\wedge h = \mathsf{height}(r) \wedge \forall r_0.\ \mathsf{tree}(r_0) \wedge \mathsf{balanced}(r_0) \\ &\quad \to \mathbf{wp}\,(\mathsf{x.down} := \mathsf{null},\ h = \mathsf{height}(r)) \end{aligned} \tag{17}$$

Formula (18) shows the update of eqh. We ensure that x is actually balanced with an appropriate guard.

$$\mathsf{eqh}'(r_1, r_2) \leftrightarrow \begin{cases} \frac{1}{2} & \text{if } \mathsf{reachable}(r_1, \mathsf{x}) \vee \mathsf{reachable}(r_2, \mathsf{x}) \\ & \text{and } \mathsf{x.down}_1 = \mathsf{null} \\ \mathsf{eqh}(r_1, r_2) & \text{otherwise} \end{cases} \tag{18}$$

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 $\mathsf{r1.next} := \mathsf{null}$ in Fig. 3 is therefore placed before any heap modification.

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:

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

4.3 Sorting

A B$^+$ tree is sorted if all of its nodes obey the constraints graphically given in Fig. 5. Sorting is maintained with the guard strategy. Equation (19) formalizes the sorting constraint shown in Fig. 5b for branch entries rbe_1 that is preserved in all intermediate states of the algorithm.

$$\begin{aligned} &\forall r.\ \mathsf{reachable}(rbe_1.\mathsf{down}, r) \to rbe_1 <_{\mathsf{k}} r \wedge r \leq_{\mathsf{next}} rbe_1 \\ &\quad\text{where}\quad r \leq_{\mathsf{next}} rbe_1 \;\leftrightarrow\; rbe_1.\mathsf{next} \neq \mathsf{null} \to r \leq_{\mathsf{k}} rbe_1.\mathsf{next} \\ &\quad\text{and}\quad\quad r_1 \leq_{\mathsf{k}} r_2 \;\leftrightarrow r_1.\mathsf{key} \leq r_2.\mathsf{key} \quad (<_{\mathsf{k}} \text{ is defined similarly}) \end{aligned} \tag{19}$$

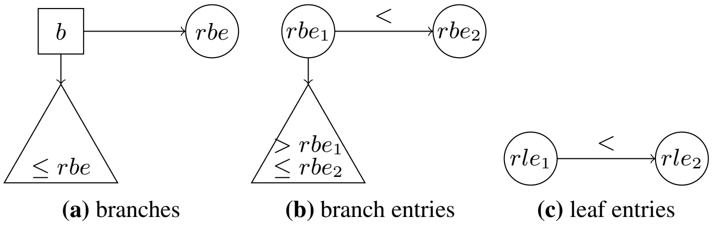


Fig. 5. Sorting Constraints. Branch-nodes are shown as boxes, entries as circles.

The guard for the assignment x.next := y is

$$\mathsf{x} <_{\mathsf{k}} \mathsf{y} \wedge (\forall r.\ \mathsf{reachable}(\mathsf{x.down}, r) \ \rightarrow r \leq_{\mathsf{k}} \mathsf{y}) \ \wedge \qquad (20)$$
$$\forall r_1, r_2.\ \mathsf{reachable}(r_1.\mathsf{down}, \mathsf{x}) \wedge \mathsf{reachable}(\mathsf{y}, r_2) \rightarrow r_2 \leq_{\mathsf{next}} r_1$$

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 reachable(r_1.down, x) determines these ancestors. The guard for x.down := y is similar.

The hard problem in TVLA is to ensure that the guard definitely holds when such statements are executed. Fig. 6 shows the execution of the typical sequence y := z.next; z.next := null; x.next := y starting with Fig. 6a. The critical relations are depicted as thick arrows in Fig. 6b, each corresponds to one of the conjuncts. When these relations evaluate to definite values, the guard holds, as shown in Fig. 6c.

The first conjunct x $<_{\mathsf{k}}$ y follows by transitivity over z and is established in (b). To derive the other two conjuncts, we focus on nodes r such that reachable(z.next, r) when executing y := z.next, and we focus on nodes r such that reachable(r.down, x) when executing z.next := null. These have the effect of splitting b and a respectively. b1 now represents the subtree that must be checked in the second conjunct and a1 gives exactly

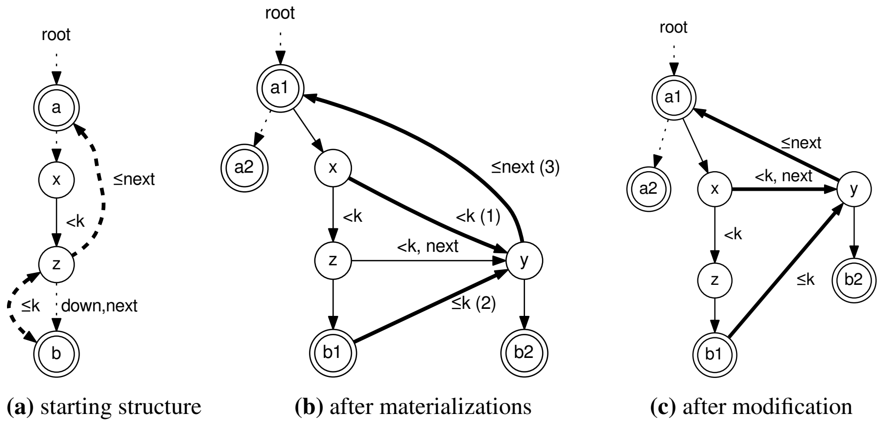


Fig. 6. Tracking sorting through modifications

the ancestors that are covered by the third conjunct. The necessary relations are then derived from the sorting invariant *before* the statement $\mathsf{z.next} := \mathsf{null}$ 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.

4.4 Elements

In [12], 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. For our case this formalization has the drawback that leaf entries must be kept distinct from other objects, so that $rle.\mathsf{key} \in \mathsf{elts}(r)$ always yields definite values. Instead, we mark a leaf entry when its keys changes, or when it is allocated or deallocated. For $\mathsf{insert\_leaf}$ we establish that it allocates at most one leaf entry, and changes no key. This is expressed as postcondition (21), where H 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 $\mathsf{delete\_leaf}$.

$$\begin{aligned}\exists\, rle.\quad & H'[rle].\mathsf{key} = k \land \mathsf{dom}(H') = \mathsf{dom}(H) \cup \{rle\} \\ & \land \forall rle_1 \in \mathsf{dom}(H).\ H[rle_1].\mathsf{key} = H'[rle_1].\mathsf{key}\end{aligned} \tag{21}$$

4.5 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

$$\mathsf{oksize}(r) \leftrightarrow (r.\mathsf{node?} \rightarrow (\mathbf{if}\ \mathsf{root}(r)\ \mathbf{then}\ 1\ \mathbf{else}\ N) \leq |\mathsf{nset}(r)| \leq 2N)$$

Node sizes are verified by a strategy similar to [6]. There, the sets of concrete individuals represented by summary nodes are tracked, as well as the cardinalities of these sets. [6] is an extension to TVLA that seems capable to directly verify 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 following contract with TVLA:

$$\begin{aligned} & rn = rp.\mathsf{down} \land \mathsf{btree}(r, [rn]) \land \mathsf{reachable}(r, rp) \land e = \mathsf{elts}(r) \land \mathsf{median}(rn, r_1.\mathsf{next}) \\ & \quad \rightarrow \mathbf{wp}(\mathsf{split\_node}(rp; rt),\ \ \mathsf{btree}(r, [rp, rn, rt]) \land e = \mathsf{elts}(r) \\ & \qquad\qquad \land \mathsf{nset}(rp) = \mathsf{nset}_0(rp) \cup \{r_1.\mathsf{next}\} \\ & \qquad\qquad \land \mathsf{nset}_0(rn) = \mathsf{nset}(rn) \cup \{r_1.\mathsf{next}\} \cup \mathsf{nset}(rt))\end{aligned} \tag{22}$$

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 (22) 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 $\mathsf{btree}(r, [rp])$.

5 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 (5) and (6) 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 website) 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. 4 for the B^+ tree invariants by analyzing failed TVLA proofs.

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 using the lemmas established by shape analysis. The most expensive consistency proofs are for update formulas like (17) and guards like (20). 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 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.

6 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.

Acknowledgments. We thank Alexander Knapp for valuable feedback.

References

1. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indices. Acta Informatica 1, 173–189 (1972)
2. Bogudlov, I., Lev-Ami, T., Reps, T., Sagiv, M.: Revamping TVLA: Making Parametric Shape Analysis Competitive. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 221–225. Springer, Heidelberg (2007)
3. Ernst, G.: KIV and TVLA proofs for B^+-Trees (2011), `http://www.informatik.uni-augsburg.de/swt/projects/btree.html`
4. Fielding, E.: The specification of abstract mappings and their implementation as B+ trees. Technical report, Oxford University, PRG-18 (1980)
5. Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: Proc. 32nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, POPL, pp. 338–350. ACM, New York (2005)
6. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: Proc. of the 36th ACM SIGPLAN-SIGACT Symp Principles of programming languages, POPL, pp. 239–251. ACM, New York (2009)
7. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
8. Herter, J.: Towards shape analysis of B-trees. Master's thesis, Universität Saarbrücken (2008)
9. Loginov, A., Reps, T., Sagiv, M.: Automated verification of the deutsch-schorr-waite tree-traversal algorithm. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 261–279. Springer, Heidelberg (2006)

10. Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Toward a verified relational database management system. In: Proc. of the 37th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, POPL, pp. 237–248. ACM, New York (2010)
11. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction—A Basis for Applications, pp. 13–39. Kluwer, Dordrecht (1998)
12. Reineke, J.: Shape analysis of sets. In: Workshop "Trustworthy Software". IBFI (2006)
13. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural Shape Analysis for Cutpoint-Free Programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 284–302. Springer, Heidelberg (2005)
14. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24, 217–298 (2002)
15. Sexton, A., Thielecke, H.: Reasoning about B+ trees with operational semantics and separation logic. Electron. Notes Theor. Comput. Sci. 218, 355–369 (2008)

A Translation from KIV to TVLA

This appendix sketches the formal definition of the translation between KIV and TVLA, which is in essence a standard construction for a homomorphism.

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 $\mathcal{A} = ((A_s)_{s \in S}, f^{\mathcal{A}}, p^{\mathcal{A}}))$ 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. A valuation v maps variables to appropriate elements of the carrier sets. In particular $v(H)$ for the heap H is a partial function from references to objects.

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 a valuation v can be translated to an untyped model $\mathcal{U} := \rho(\mathcal{A}, v)$ of TVLA. The carrier set U of $\mathcal{U}$ is defined as the domain of the heap: $U := \{a : a \in dom(v(H))\}$. Selectors are interpreted as

$$s^{\mathcal{U}} := \Big\{(a, b) : a \in dom(v(H)) \wedge b = v(H)(a).s^{\mathcal{A}} \wedge b \in dom(v(H))\Big\}$$

and other predicates q are interpreted as

$$q^{\mathcal{U}}(a_1, \ldots, a_n) \text{ iff } q^{\mathcal{A}}(a_1, \ldots, a_n, v(H))$$

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

$$\tau(s(r_1, r_2)) = (H[r_1].s = r_2) \text{ and } \tau(q(r_1, \ldots, r_n)) = q(r_1, \ldots, r_n, H)$$

It is easy to prove (by induction over the formula) that for any TVLA formula φ

$$\rho(\mathcal{A}, v) \models \varphi \text{ iff } \mathcal{A}, v \models \tau(\varphi)$$

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

Semantically, assignments stm in KIV map a valuation v to a modified valuation v'. Proof obligation (5) guarantees that for an update formula φ^p_{stm}

$$\rho(\mathcal{A}, v') \models p(r_1, \ldots, r_n) \text{ iff } \rho(\mathcal{A}, v) \models \varphi^p_{stm}(r_1, \ldots, r_n)$$

i.e., the semantics of p in the state $\rho(\mathcal{A}, v')$ after the assignment is as predicted by φ^p_{stm}.