Automatic Functional Correctness Proofs for Functional Search Trees

Tobias Nipkow(✉)

Technische Universität München, Munich, Germany
nipkow@in.tum.de
http://www.in.tum.de/~nipkow

Abstract. In a new approach, functional correctness specifications of *insert*/*update* and *delete* operations on search trees are expressed on the level of lists by means of an inorder traversal function that projects trees to lists. With the help of a small lemma library, functional correctness and preservation of the search tree property are proved automatically (in Isabelle/HOL) for a range of data structures: unbalanced binary trees, AVL trees, red-black trees, 2-3 and 2-3-4 trees, 1-2 brother trees, AA trees and splay trees.

1 Introduction

Most books and articles on search tree data structures do not discuss functional correctness, which is taken to be obvious, but concentrate on non-obvious structural invariants like balancedness. This paper confirms that this is the right attitude by providing a framework for proving the functional correctness of eight different search tree data structures automatically (in Isabelle/HOL [19,21]).

What is proved automatically? Functional correctness of *insert*, *delete* and *isin* together with the preservation of the search tree invariant, i.e. sortedness, by *insert* and *delete*. Structural invariants like balancedness are proved manually, depend on the specific data structure, and are not discussed here.

Which data structures are covered? Unbalanced binary trees, AVL trees, red-black trees, 2-3 and 2-3-4 trees, 1-2 brother trees, AA trees and splay trees.[1] As far as we know, these are the first formal proofs for 2-3 and 2-3-4 trees, 1-2 brother trees and AA trees, and the first automatic proofs for most of the eight data structures.

What does automatic mean? It means that all the required theorems are proved by induction followed by a single invocation of Isabelle's `auto` proof method, parameterized with a fixed set of basic lemmas plus further lemmas about auxiliary functions. The lemmas to be proved about *insert*, *delete* and *isin* are fixed; lemmas about auxiliary functions need to be invented but (mostly) follow a simple pattern.

T. Nipkow—Supported by DFG Koselleck grant NI 491/16-1.

[1] See http://isabelle.in.tum.de/library/HOL/HOL-Data_Structures/ or the source directory `src/HOL/Data_Structures/` in the Isabelle distribution.

J.C. Blanchette and S. Merz (Eds.): ITP 2016, LNCS 9807, pp. 307–322, 2016.
DOI: 10.1007/978-3-319-43144-4_19

The paper is structured as follows. Section 3 presents two approaches to the specification and verification of set implementations: the standard approach and our new approach. Section 4 details the verification framework behind the new approach. In Sect. 5 eight different search tree implementations and their correctness proofs are discussed. In a final section it is shown how the framework can be generalized from sets to maps.

Related work is discussed in the body of the paper. With one exception, the proofs in previous work are not automatic. We refrain from stating this each time and we do not describe how far from automatic they are, although this varies significantly (from a few to more than a hundred lines).

2 Lists and Trees

Lists (type *'a list*) are constructed from the empty list [] via the infix cons-operator "·". The notation $[x,y,z]$ is short for $x \cdot y \cdot z \cdot []$. The infix @ concatenates two lists.

Binary trees are defined as the data type *'a tree* with two constructors: the empty tree or leaf $\langle\rangle$ and the node $\langle l, a, r\rangle$ with subtrees $l, r :: 'a\ tree$ and contents $a :: 'a$.

There is also a type *'a set* of sets with their usual operations.

3 Set Implementations

We require that an implementation of sets provides some type $'a\ t$ (where $'a$ is the element type) and the operations

$empty :: 'a\ t$
$insert :: 'a \Rightarrow 'a\ t \Rightarrow 'a\ t$
$delete :: 'a \Rightarrow 'a\ t \Rightarrow 'a\ t$
$isin :: 'a\ t \Rightarrow 'a \Rightarrow bool$

In the rest of the paper we ignore *empty* because it is trivial.

In order to specify these operations we assume that there is also an *abstraction function* $set :: 'a\ t \Rightarrow 'a\ set$ and a data type invariant $invar :: 'a\ t \Rightarrow bool$. These are not part of the interface and need not be executable but have to be provided in order to prove an implementation correct w.r.t. the specification in Fig. 1. Specifications phrased in terms of abstraction functions that are required to be homomorphisms go back to Hoare [11] and became an integral part of the model-oriented specification language VDM [13]. In the first-order context of universal algebra it was shown that there are always fully abstract models such that any concrete implementation can be shown correct with a homomorphism [16]. In Isabelle, implementations that satisfy such specifications can automatically be plugged in for the abstract type by regarding the abstraction function as a constructor [9]. For example, turning the equation $isin\ s\ x = (x \in set\ s)$ around tells us how to evaluate $x \in set\ s$ with the help of *isin*.

From now on we assume that the element type 'a is linearly ordered.

$$
\begin{array}{l}
\mathit{invar}\ s \Longrightarrow \mathit{set}\ (\mathit{insert}\ x\ s) = \{x\} \cup \mathit{set}\ s \\
\mathit{invar}\ s \Longrightarrow \mathit{set}\ (\mathit{delete}\ x\ s) = \mathit{set}\ s - \{x\} \\
\mathit{invar}\ s \Longrightarrow \mathit{isin}\ s\ x = (x \in \mathit{set}\ s) \\
\mathit{invar}\ s \Longrightarrow \mathit{invar}\ (\mathit{insert}\ x\ s) \\
\mathit{invar}\ s \Longrightarrow \mathit{invar}\ (\mathit{delete}\ x\ s)
\end{array}
$$

Fig. 1. Specification of set implementations

3.1 The Standard Approach

The most compact form of the standard approach to the verification of search tree implementations of sets consists of the following items:

- An abstraction function *set* that extract the set of elements in a tree.
- A recursively defined (binary) search tree invariant
  $\mathit{bst}\ \langle l,a,r\rangle = (\mathit{bst}\ l \wedge \mathit{bst}\ r \wedge (\forall x \in \mathit{set}\ l.\ x < a) \wedge (\forall x \in \mathit{set}\ r.\ a < x)$
- Proof of the correctness conditions in Fig. 1 where *invar* is *bst*, possibly conjoined with additional structural invariants.

There are many variations of the above setup, some of which address two complications that arise when automating the proofs, the quantifiers and the non-free data type of sets:

- In the definition of *bst*, quantifiers are replaced by auxiliary functions that check if all elements in a tree are less/greater than a given element.
- Instead of extensional equality of sets, e.g. $\mathit{set}\ (\mathit{insert}\ x\ s) = \{x\} \cup \mathit{set}\ s$, pointwise equality is proved, e.g. $\mathit{isin}\ (\mathit{insert}\ x\ s)\ y = (x = y \vee \mathit{isin}\ s\ y)$.
- The predicate *bst* is defined inductively rather than recursively.

We subsume all of these variations under the "standard approach". Unless stated otherwise, all related work follows the standard approach.

3.2 The *inorder* Approach

Why is it perfectly obvious that the following equation (where R/B construct red/black nodes) preserves sortedness and the set of elements of a tree?

$$\mathit{balance}\ (R\ (R\ t_1\ a\ t_2)\ b\ t_3)\ c\ t_4 = R\ (B\ t_1\ a\ t_2)\ b\ (B\ t_3\ c\ t_4)$$

Because the sequence of subtrees and elements is the same on both sides! We merely need to make the machine see this as well as we do.

The key idea of our approach is to base it on the inorder traversal of trees. That is, we use lists as an intermediate data type between sets and trees. To this end we need four auxiliary functions on lists:

- $\uparrow :: {'a}\ \mathit{list} \Rightarrow \mathit{bool}$
  $\uparrow$ means that the list is sorted in ascending order w.r.t. $<$.

$$invar\ t \Longrightarrow \lfloor insert\ x\ t \rfloor = ins_{list}\ x\ \lfloor t \rfloor \quad (1)$$
$$invar\ t \Longrightarrow \lfloor delete\ x\ t \rfloor = del_{list}\ x\ \lfloor t \rfloor \quad (2)$$
$$invar\ t \Longrightarrow isin\ t\ x = (x \in elems\ \lfloor t \rfloor) \quad (3)$$
$$invar\ t \Longrightarrow inv\ (insert\ x\ t) \quad (4)$$
$$invar\ t \Longrightarrow inv\ (delete\ x\ t) \quad (5)$$

Fig. 2. Specification of set implementations over ordered types

$$\uparrow [] = True$$
$$\uparrow [x] = True$$
$$\uparrow (x \cdot y \cdot zs) = (x < y \wedge \uparrow (y \cdot zs))$$

– $ins_{list} :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$
ins_{list} inserts an element at the correct position into a sorted list if the element is not present in the list yet.

$$ins_{list}\ x\ [] = [x]$$
$$ins_{list}\ x\ (a \cdot xs) =$$
$$(\mathsf{if}\ x < a\ \mathsf{then}\ x \cdot a \cdot xs\ \mathsf{else\ if}\ x = a\ \mathsf{then}\ a \cdot xs\ \mathsf{else}\ a \cdot ins_{list}\ x\ xs)$$

– $del_{list} :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$
del_{list} deletes the first occurrence of an element from a list.

$$del_{list}\ x\ [] = []$$
$$del_{list}\ x\ (a \cdot xs) = (\mathsf{if}\ x = a\ \mathsf{then}\ xs\ \mathsf{else}\ a \cdot del_{list}\ x\ xs)$$

– $elems :: 'a\ list \Rightarrow 'a\ set$
$elems$ turns a list into the set of its elements.

$$elems\ [] = \emptyset$$
$$elems\ (x \cdot xs) = \{x\} \cup elems\ xs$$

A new specification of sets (over a linearly ordered type $'a$) is shown in Fig. 2. The crucial ingredient is an additional specification function

$$inorder :: 'a\ t \Rightarrow 'a\ list$$

As the name *inorder* suggests, you should now think of $'a\ t$ as a type of trees. We abbreviate $inorder\ t$ by $\lfloor t \rfloor$.

The fact that some t is a search tree, i.e. sorted, can now be expressed as $\uparrow \lfloor t \rfloor$. This invariant will be dealt with automatically. Of course search trees frequently have additional structural invariants. These can be supplied via yet another specification function $inv :: 'a\ t \Rightarrow bool$. Both kinds of invariants are combined into *invar*:

$$invar\ t = (inv\ t \wedge \uparrow \lfloor t \rfloor)$$

The first three propositions in Fig. 2 demand the functional correctness of *insert*, *delete* and *isin* w.r.t. ins_{list}, del_{list} and *elems*. The next two propositions demand that *inv* is invariant. If we interpret function *set* as $elems \circ inorder$ it is easy to show that Fig. 2 implies Fig. 1 with the help of the following simple inductive lemmas:

$$elems\ (ins_{list}\ x\ xs) = \{x\} \cup elems\ xs$$
$$\uparrow xs \Longrightarrow distinct\ xs$$
$$distinct\ xs \Longrightarrow elems\ (del_{list}\ x\ xs) = elems\ xs - \{x\}$$
$$\uparrow xs \Longrightarrow \uparrow (ins_{list}\ x\ xs)$$
$$\uparrow xs \Longrightarrow \uparrow (del_{list}\ x\ xs)$$

In summary: the functional correctness of an implementation of *insert*, *delete* and *isin* on some data structure can be verified by proving the properties in Fig. 2 for some suitable definition of *inorder* and *inv*. In the following section we introduce a library of lemmas about $\uparrow$, ins_{list} and del_{list} that allows us to automate the proofs of (1)–(3).

Note that although we equate (1)–(3) with "functional correctness", it is more: (1)–(3) also imply that sortedness is an invariant.

4 The Verification Framework

We do not claim to provide a framework that can prove any implementation of sets by search trees automatically correct. Instead we provide lemmas that work in practice (they automate the correctness proofs for a list of benchmark implementations presented in Sect. 5) and are well motivated by general considerations concerning the shape of formulas that arise in the verification.

As a motivating example we consider ordinary unbalanced binary trees $'a\ tree$. The textbook definitions of *insert*, *delete* and *isin* are omitted. Let us examine how to prove

$$\uparrow \lfloor t \rfloor \Longrightarrow \lfloor insert\ x\ t \rfloor = ins_{list}\ x\ \lfloor t \rfloor$$

The proof is by induction on t and we consider the case $t = \langle l, a, r \rangle$ such that $x < a$. Ideally the proof looks like this:

$$\lfloor insert\ x\ t \rfloor = \lfloor insert\ x\ l \rfloor\ @\ a \cdot \lfloor r \rfloor = ins_{list}\ x\ \lfloor l \rfloor\ @\ a \cdot \lfloor r \rfloor$$
$$= ins_{list}\ x\ (\lfloor l \rfloor\ @\ a \cdot \lfloor r \rfloor) = ins_{list}\ x\ t$$

The first and last step are by definition, the second step by induction hypothesis, but the third step requires two lemmas:

$$\uparrow (xs\ @\ y \cdot ys) = (\uparrow (xs\ @\ [y]) \wedge \uparrow (y \cdot ys))$$
$$\uparrow (xs\ @\ [a]) \wedge x < a \Longrightarrow ins_{list}\ x\ (xs\ @\ a \cdot ys) = ins_{list}\ x\ xs\ @\ a \cdot ys$$

The first lemma rewrites the assumption $\uparrow \lfloor t \rfloor$ to $\uparrow (\lfloor l \rfloor\ @\ [a]) \wedge \uparrow (a \cdot \lfloor r \rfloor)$, thus allowing the second lemma to rewrite the term $ins_{list}\ x\ (\lfloor l \rfloor\ @\ a \cdot \lfloor r \rfloor)$ to $ins_{list}\ x\ \lfloor l \rfloor\ @\ a \cdot \lfloor r \rfloor$.

It may seem that the two lemmas just shown are rather arbitrary, but we will see that in the context of trees, where each node is a tuple $\langle s_0, a_1, s_1, \ldots, s_n \rangle$ of subtrees s_i alternating with elements a_i, there is an underlying principle. In the properties in Fig. 2 the following three terms are crucial: $\uparrow \lfloor t \rfloor$, $ins_{list}\ x\ \lfloor t \rfloor$ and $del_{list}\ x\ \lfloor t \rfloor$. Assuming that the properties are proved by induction, t will be some (possibly complicated) tree constructor term. Evaluating $\lfloor t \rfloor$ will thus lead to a list of the following form where sublists and individual elements alternate:

$$\uparrow (xs \mathbin{@} y \cdot ys) = (\uparrow (xs \mathbin{@} [y]) \wedge \uparrow (y \cdot ys)) \tag{6}$$
$$\uparrow (x \cdot xs \mathbin{@} y \cdot ys) = (\uparrow (x \cdot xs) \wedge x < y \wedge \uparrow (xs \mathbin{@} [y]) \wedge \uparrow (y \cdot ys)) \tag{7}$$
$$\uparrow (x \cdot xs) \Longrightarrow \uparrow xs \tag{8}$$
$$\uparrow (xs \mathbin{@} [y]) \Longrightarrow \uparrow xs \tag{9}$$
$$\uparrow (xs \mathbin{@} [a]) \Longrightarrow ins_{list}\ x\ (xs \mathbin{@} a \cdot ys) = \quad (\mathit{if}\ x < a\ \mathit{then}\ ins_{list}\ x\ xs \mathbin{@} a \cdot ys\ \mathit{else}\ xs \mathbin{@} ins_{list}\ x\ (a \cdot ys)) \tag{10}$$
$$\uparrow (xs \mathbin{@} a \cdot ys) \Longrightarrow del_{list}\ x\ (xs \mathbin{@} a \cdot ys) = \quad (\mathit{if}\ x < a\ \mathit{then}\ del_{list}\ x\ xs \mathbin{@} a \cdot ys\ \mathit{else}\ xs \mathbin{@} del_{list}\ x\ (a \cdot ys)) \tag{11}$$
$$elems\ (xs \mathbin{@} ys) = elems\ xs \cup elems\ ys \tag{12}$$
$$\uparrow (y \cdot xs) \wedge x \leq y \Longrightarrow x \notin elems\ xs \tag{13}$$
$$\uparrow (xs \mathbin{@} [y]) \wedge y \leq x \Longrightarrow x \notin elems\ xs \tag{14}$$

Fig. 3. Lemmas for $\uparrow$, ins_{list}, del_{list} and *elems*

$$\lfloor t_1 \rfloor \mathbin{@} a_1 \cdot \lfloor t_2 \rfloor \mathbin{@} a_2 \cdot \ldots \cdot \lfloor t_n \rfloor$$

Now we discuss a set of lemmas (see Fig. 3) that allow us to simplify the application of $\uparrow$, ins_{list} and del_{list} to such terms.

Terms of the form $\uparrow(xs_1 \mathbin{@} a_1 \cdot xs_2 \mathbin{@} a_2 \cdot \ldots \cdot xs_n)$ are decomposed into the following *basic* formulas

$\uparrow (xs \mathbin{@} [a])$ (simulating $\forall x{\in}set\ xs.\ x < a$)
$\uparrow (a \cdot xs)$ (simulating $\forall x{\in}set\ xs.\ a < x$)
$a < b$

by the rewrite rules (6)–(7). Lemmas (8)–(9) enable deductions from basic formulas.

Terms of the form $ins_{list}\ x\ (xs_1 \mathbin{@} a_1 \cdot xs_2 \mathbin{@} a_2 \cdot \ldots \cdot xs_n)$ are rewritten with equation (10) (and the defining equations for ins_{list}) to push ins_{list} inwards. Terms of the form $del_{list}\ x\ (xs_1 \mathbin{@} a_1 \cdot xs_2 \mathbin{@} a_2 \cdot \ldots \cdot xs_n)$ are rewritten with Eq. (11) (and the defining equations for del_{list}) to push del_{list} inwards.

Finally we need lemmas (12)–(14) about *elems* on sorted lists.

The lemmas in Fig. 3 form the complete set of basic lemmas on which the automatic proofs of almost all search trees in the paper rest; only splay trees need additional lemmas.

4.1 Proof Automation by Rewriting

The automatic proofs rely on conditional, contextual term rewriting with the following bells and whistles (which Isabelle's simplifier provides):

- Conjunctions in the context are split up into their conjuncts.
- Conditionals and case-expressions can be split automatically.
- A decision procedure for linear orders that can decide if some literal (a possibly negated atom $a < b$ or $a \leq b$) follows from a set of literals in the context.

– Implications (8)–(9) lead to nontermination when used as conditional rewrite rules. It must be possible to direct the simplifier to solve the preconditions of those rules by assumptions in the context rather than a recursive simplifier invocation. In Isabelle there is a constant $ASSUMPTION = (\lambda x.\ x)$ that can be wrapped around a precondition of a rewrite rule and prevents recursive applications of the simplifier to that precondition.

5 An Arboretum

In the rest of this section we focus on (1) and (2) when discussing the proofs of the properties in Fig. 2. This is because requirement (3) can always (except for splay trees) be proved automatically without further lemmas and (4) and (5) are specific to the individual data structures and not part of functional correctness.

Because there is not enough space to present all definitions and proofs, Table 1 gives an overview in terms of lines of code and numbers of functions needed for each data structure. Because *isin* is (almost) the same for all of them (except splay trees), it is excluded. The table shows that there is at most one lemma per function, except for splay trees.

Table 1. Code and proof statistics for *insert* + *delete* (l.o. = lines of)

	Unbal	AVL	Red-Black	2-3	2-3-4	Brother	AA	Splay
l.o. code	17	45	61	88	143	66	55	46
functions	3	8	11	12	16	10	8	4
lemmas	3	6	11	10	14	10	6	5

The majority of lemmas about auxiliary functions follow a simple pattern. Typical examples are balancing functions, e.g. $\lfloor bal\ t \rfloor = \lfloor t \rfloor$, or smart constructors, e.g. $\lfloor node\ l\ a\ r \rfloor = \lfloor l \rfloor\ @\ a \cdot \lfloor r \rfloor$. We call these *trivial lemmas*. More complicated lemmas are discussed explicitly in the text; we call them *non-trivial*.

All our implementations compare elements with a comparison operator *cmp* that returns an element of the **datatype** $cmp = LT \mid EQ \mid GT$.

5.1 Unbalanced Trees

Function *insert* is trivial and (1) is proved directly. Function *delete* is more interesting because it is defined with the help of an auxiliary function:

$$
\begin{array}{l}
delete\ x\ \langle\rangle = \langle\rangle \\
delete\ x\ \langle l,\ a,\ r\rangle = \\
(\mathsf{case}\ cmp\ x\ a\ \mathsf{of}\ LT \Rightarrow \langle delete\ x\ l,\ a,\ r\rangle \\
\mid EQ \Rightarrow \mathsf{if}\ r = \langle\rangle\ \mathsf{then}\ l\ \mathsf{else}\ \mathsf{let}\ (x,\ y) = del\_min\ r\ \mathsf{in}\ \langle l,\ x,\ y\rangle \\
\mid GT \Rightarrow \langle l,\ a,\ delete\ x\ r\rangle)
\end{array}
$$

$$\begin{array}{l} \mathit{del\_min}\ \langle l,\ a,\ r\rangle = \\ (\mathit{if}\ l = \langle\rangle\ \mathit{then}\ (a,\ r)\ \mathit{else\ let}\ (x,\ l') = \mathit{del\_min}\ l\ \mathit{in}\ (x,\ \langle l',\ a,\ r\rangle)) \end{array}$$

The proof of (2) requires the following lemma about *del_min* that the user has to formulate himself; the proof is again automatic.

$$\mathit{del\_min}\ t = (x,\ t') \wedge t \neq \langle\rangle \Longrightarrow x \cdot \lfloor t' \rceil = \lfloor t \rceil$$

This is one of the more "difficult" lemmas to invent.

5.2 AVL Trees

Our starting point was an existing formalization [20] which follows the standard approach. Functional correctness of AVL trees can be proved without assuming any structural (height) invariants. The only non-trivial lemma we require is

$$\mathit{del\_max}\ t = (t',\ a) \wedge t \neq \langle\rangle \Longrightarrow \lfloor t' \rceil\ @\ [a] = \lfloor t \rceil$$

Related Work. Filliâtre and Letouzey [6] report on a verification of AVL trees in Coq. They follow the standard approach, except that the executable functions are extracted from constructive proofs. An updated version of their proofs in the Coq distribution gives the functions explicitly. Ralston [26] reports a proof with ACL2. The verification by Clochard [4] in Why3 is interesting because he also abstracts trees to their inorder traversal and reports that the proofs for AVL trees are automatic.

5.3 Red-Black Trees

Red-black trees were invented by Bayer [3]. Guibas and Sedgewick [8] introduced the red/black color convention. Red-black trees can be seen as an encoding of 2-3-4 trees as binary trees.

Our starting point was an existing formalization in the Isabelle distribution (in `HOL/Library/RBT_Impl.thy`, by Reiter and Krauss) which in turn is based on the code by Okasaki [22] (for *insert*) and Stefan Kahrs [14] (for *delete* see the URL given in the article). The original verification has a certain similarity to ours because it also involves an inorder listing of the tree (function *entries*), but a number of the proofs are distinctly long and manual. In contrast, the only non-trivial lemmas we require are the following ones that need to be proved simultaneously about three auxiliary functions:

$$\begin{array}{l} \uparrow \lfloor t \rceil \Longrightarrow \lfloor \mathit{del}\ x\ t \rceil = \mathit{del}_{\mathit{list}}\ x\ \lfloor t \rceil \\ \uparrow \lfloor l \rceil \Longrightarrow \lfloor \mathit{delL}\ x\ l\ a\ r \rceil = \mathit{del}_{\mathit{list}}\ x\ \lfloor l \rceil\ @\ a \cdot \lfloor r \rceil \\ \uparrow \lfloor r \rceil \Longrightarrow \lfloor \mathit{delR}\ x\ l\ a\ r \rceil = \lfloor l \rceil\ @\ a \cdot \mathit{del}_{\mathit{list}}\ x\ \lfloor r \rceil \end{array}$$

Of course the proof is automatic, as usual.

Functional correctness of red-black trees can be proved without assuming any structural (red-black) invariants.

Related Work. Filliâtre and Letouzey [6] and Appel [2] verified red-black trees in Coq.

5.4 2-3 Trees

In a 2-3 tree (invented by Hopcroft in 1970 [5]), every non-leaf node has either two or three children: $\langle l, a, r\rangle$ or $\langle l, a, m, b, r\rangle$ where l, m, r are trees and a, b are elements. One can view $\langle l, a, m, b, r\rangle$ as a more compact representation of $\langle l, a, \langle m, b, r\rangle\rangle$ (see AA trees). Their structural invariant is that they are balanced, i.e. all leaves occur at the same depth.

Our code is based on the lecture notes by Turbak [29], who presents the key transformations in a graphical format. We present the more complex *delete* function in Fig. 4. Function *del* descends into the tree until the element (or a leaf) is found. Modified subtrees are recombined with smart constructors *nodeij* that combines i subtrees where subtree j has been modified and is wrapped up in either T_d (if the height of the subtree is unchanged) or Up_d (if the height of the subtree has decreased). We only show the functions *nodei*1 because the other *nodeij* are symmetric.

The lemmas required for the correctness proof are similar to what we have seen already, with one new complication: the balancedness invariant *bal* is frequently required as a precondition, e.g. here:

$$del\_min\ t = (x,\ t') \wedge bal\ t \wedge 0 < height\ t \Longrightarrow x \cdot \lfloor tree_d\ t' \rfloor = \lfloor t \rfloor$$

Our automatic framework can cope because *bal* and *height* are defined in a straightforward manner by primitive recursion.

Related Work. The existing formalization of 2-3 trees in the Isabelle distribution (in `HOL/ex/Tree23.thy`, by Huffman and Nipkow) proves invariants but not functional correctness. Hoffmann and O'Donnell [12] give an equational definition of insertion. Reade [27] gives a similar equational definition of insertion and adds deletion; Turbak's version of deletion appears a bit simpler. Reade sketches (because there are too many cases) a pen-and-paper correctness proof and writes: "Mechanical support for such reasoning and the potential for partial automation of similar proofs are topics currently being investigated by the author".

5.5 2-3-4 Trees

2-3-4 trees are an extension of 2-3 trees where nodes may also have 4 children: $\langle t_1, a, t_2, b, t_3, c, t_4\rangle$. Their structural invariant is that they are balanced, i.e. all leaves occur at the same depth. The code for 2-3-4 trees can also be viewed as an extension of that for 2-3 trees with additional cases. There are also new smart constructors *node*4j, e.g. *node*41:

$$node41\ (T_d\ t_1)\ a\ t_2\ b\ t_3\ c\ t_4 = T_d\ \langle t_1,\ a,\ t_2,\ b,\ t_3,\ c,\ t_4\rangle$$
$$node41\ (Up_d\ t_1)\ a\ \langle t_2,\ b,\ t_3\rangle\ c\ t_4\ d\ t_5 = T_d\ \langle\langle t_1,\ a,\ t_2,\ b,\ t_3\rangle,\ c,\ t_4,\ d,\ t_5\rangle$$

```
datatype 'a up_d = T_d ('a tree23) | Up_d ('a tree23)

tree_d (T_d t) = t
tree_d (Up_d t) = t

node21 (T_d t1) a t2 = T_d ⟨t1, a, t2⟩
node21 (Up_d t1) a ⟨t2, b, t3⟩ = Up_d ⟨t1, a, t2, b, t3⟩
node21 (Up_d t1) a ⟨t2, b, t3, c, t4⟩ = T_d ⟨⟨t1, a, t2⟩, b, ⟨t3, c, t4⟩⟩

node31 (T_d t1) a t2 b t3 = T_d ⟨t1, a, t2, b, t3⟩
node31 (Up_d t1) a ⟨t2, b, t3⟩ c t4 = T_d ⟨⟨t1, a, t2, b, t3⟩, c, t4⟩
node31 (Up_d t1) a ⟨t2, b, t3, c, t4⟩ d t5 = T_d ⟨⟨t1, a, t2⟩, b, ⟨t3, c, t4⟩, d, t5⟩

del_min ⟨⟩, a, ⟨⟩⟩ = (a, Up_d ⟨⟩)
del_min ⟨⟨⟩, a, ⟨⟩, b, ⟨⟩⟩ = (a, T_d ⟨⟨⟩, b, ⟨⟩⟩)
del_min ⟨l, a, r⟩ = (let (x, l') = del_min l in (x, node21 l' a r))
del_min ⟨l, a, m, b, r⟩ = (let (x, l') = del_min l in (x, node31 l' a m b r))

del x ⟨⟩ = T_d ⟨⟩
del x ⟨⟨⟩, a, ⟨⟩⟩ = (if x = a then Up_d ⟨⟩ else T_d ⟨⟨⟩, a, ⟨⟩⟩)
del x ⟨⟨⟩, a, ⟨⟩, b, ⟨⟩⟩ =
T_d (if x = a then ⟨⟨⟩, b, ⟨⟩⟩ else if x = b then ⟨⟨⟩, a, ⟨⟩⟩ else ⟨⟨⟩, a, ⟨⟩, b, ⟨⟩⟩)
del x ⟨l, a, r⟩ =
(case cmp x a of LT ⇒ node21 (del x l) a r
 | EQ ⇒ let (a', t) = del_min r in node22 l a' t | GT ⇒ node22 l a (del x r))
del x ⟨l, a, m, b, r⟩ =
(case cmp x a of LT ⇒ node31 (del x l) a m b r
 | EQ ⇒ let (a', m') = del_min m in node32 l a' m' b r
 | GT ⇒ case cmp x b of LT ⇒ node32 l a (del x m) b r
           | EQ ⇒ let (b', r') = del_min r in node33 l a m b' r'
           | GT ⇒ node33 l a m b (del x r))

delete x t = tree_d (del x t)
```

Fig. 4. Deletion in 2-3 trees

```
node41 (Up_d t1) a ⟨t2, b, t3, c, t4⟩ d t5 e t6 =
T_d ⟨⟨t1, a, t2⟩, b, ⟨t3, c, t4⟩, d, t5, e, t6⟩
node41 (Up_d t1) a ⟨t2, b, t3, c, t4, d, t5⟩ e t6 f t7 =
T_d ⟨⟨t1, a, t2⟩, b, ⟨t3, c, t4, d, t5⟩, e, t6, f, t7⟩
```

Related Work. It appears that the only (partially) published functional implementations of 2-3-4 trees is one in Maude [15] where the full code is available online. No formal proofs are reported.

5.6 1-2 Brother Trees

A 1-2 brother tree [23,24] is a binary tree with one further constructor $N1$ from trees to trees for unary nodes. The structural invariant is that the tree

is balanced (all leaves at the same depth) and that every unary node has a binary brother. Unary nodes allow us to balance any tree. There is a bijection between 1-2 brother trees and AVL trees: remove the unary nodes from a 1-2 brother tree and you obtain an AVL tree. Our formalization is based on the article by Hinze [10] where all code and invariants can be found. Hinze captures the invariant by two sets $B\ h$ and $U\ h$, the sets of brother trees of height h that have a binary (or nullary) respectively unary root node. The actual brother trees are captured by B; U is an auxiliary notion. The correctness lemmas (1)–(3) for *insert*, *delete* and *isin* employ the abbreviation $T\ h = B\ h \cup U\ h$:

$$t \in T\ h \wedge \uparrow \lfloor t \rfloor \Longrightarrow \lfloor insert\ a\ t \rfloor = ins_{list}\ a\ \lfloor t \rfloor$$
$$t \in T\ h \wedge \uparrow \lfloor t \rfloor \Longrightarrow \lfloor delete\ x\ t \rfloor = del_{list}\ x\ \lfloor t \rfloor$$
$$t \in T\ h \wedge \uparrow \lfloor t \rfloor \Longrightarrow isin\ t\ x = (x \in elems\ \lfloor t \rfloor)$$

The non-trivial but automatic auxiliary lemmas are

$$t \in T\ h \wedge \uparrow \lfloor t \rfloor \Longrightarrow \lfloor ins\ a\ t \rfloor = ins_{list}\ a\ \lfloor t \rfloor$$
$$t \in T\ h \wedge \uparrow \lfloor t \rfloor \Longrightarrow \lfloor del\ x\ t \rfloor = del_{list}\ x\ \lfloor t \rfloor$$

$$t \in T\ h \Longrightarrow$$
$$(del\_min\ t = None) = (\lfloor t \rfloor = []) \wedge$$
$$(del\_min\ t = Some\ (a,\ t') \longrightarrow \lfloor t \rfloor = a \cdot \lfloor t' \rfloor)$$

5.7 AA Trees

Arne Anderson [1] invented a particularly simple form of balanced trees, named AA trees by Weiss [30]. They encode 2-3 trees as binary trees (with the help of an additional height field, although a single bit would suffice). Their main selling point is simplicity and compactness of the code. Our verification started from the functional version of AA trees published by Ragde [25] without proofs. The proofs for insertion were automatic as usual, but deletion posed problems.

The use of non-linear patterns in the Haskell code for `delete` was easily fixed. Then a failed correctness proof revealed that function `dellrg` goes down the wrong branch in the recursive case. After this bug was corrected the next complication was the fact that the definition of function `adjust` (which is supposed to restore the invariant after deletion) does not cover certain trees that cannot arise. Therefore I needed to introduce the following invariant corresponding to the textual invariants AA1–AA3 in [25]; function *lvl* returns the height field of a node:

$$invar\ \langle\rangle = True$$
$$invar\ \langle h,\ l,\ a,\ r\rangle =$$
$$(invar\ l \wedge invar\ r \wedge h = lvl\ l + 1 \wedge$$
$$(h = lvl\ r + 1 \vee (\exists\, lr\ b\ rr.\ r = \langle h,\ lr,\ b,\ rr\rangle \wedge h = lvl\ rr + 1)))$$

Proving that insertion and deletion preserve the invariant was non-trivial, in particular because there were two more bugs:

– Function `dellrg` fails to call `adjust` to restore the invariant. This is the correct code (we call `dellrg` *del_max*):

$$del\_max\ \langle lv,\ l,\ a,\ \langle\rangle\rangle = (l,\ a)$$
$$del\_max\ \langle lv,\ l,\ a,\ r\rangle = (\mathsf{let}\ (r',\ b) = del\_max\ r\ \mathsf{in}\ (adjust\ \langle lv,\ l,\ a,\ r'\rangle,\ b))$$

– The auxiliary function `nlvl` is incorrect. The correct version is as follows: $nlvl\ t = (\mathsf{if}\ sngl\ t\ \mathsf{then}\ lvl\ t\ \mathsf{else}\ lvl\ t\ +\ 1)$

For the verification of functional correctness of deletion the domain of the partial `adjust` had to be characterized by a predicate *pre_adjust* (not in [25]). With its help we can formulate and prove the trivial *inorder*-lemma for *adjust*:

$$t \neq \langle\rangle \wedge pre\_adjust\ t \Longrightarrow \lfloor adjust\ t\rfloor = \lfloor t\rfloor$$

The main correctness theorem (2) requires a number of further lemmas:

$$del\_max\ t = (t',\ x) \wedge t \neq \langle\rangle \wedge invar\ t \Longrightarrow \lfloor t'\rfloor\ @\ [x] = \lfloor t\rfloor$$
$$invar\ \langle lv,\ l,\ a,\ r\rangle \wedge post\_del\ l\ l' \Longrightarrow pre\_adjust\ \langle lv,\ l',\ b,\ r\rangle$$
$$invar\ \langle lv,\ l,\ a,\ r\rangle \wedge post\_del\ r\ r' \Longrightarrow pre\_adjust\ \langle lv,\ l,\ a,\ r'\rangle$$
$$invar\ t \wedge (t',\ x) = del\_max\ t \wedge t \neq \langle\rangle \Longrightarrow post\_del\ t\ t'$$
$$invar\ t \Longrightarrow post\_del\ t\ (delete\ x\ t)$$

As usual, the proofs of the *inorder*-lemmas and theorems are automatic. The last four lemmas and the pre- and post-conditions involved are part of the invariant proofs and are merely reused. Hence they are not included in Table 1.

5.8 Splay Trees

Splay trees [28] are self-adjusting binary search trees where query and update operations modify the tree by rotating the accessed element to the root of the tree. The logarithmic amortized complexity of splay trees has been verified before [18]. The functional correctness proofs [17] followed the standard approach. Starting from the same code we automated those proofs.

Splay trees are different from the other trees we cover. All operations are based on a function $splay :: 'a \Rightarrow 'a\ tree \Rightarrow 'a\ tree$ that rotates the given element (or an element close to it) to the root of the tree. For example, this is *isin*:

$$isin\ t\ x = (\mathsf{case}\ splay\ x\ t\ \mathsf{of}\ \langle\rangle \Rightarrow False \mid \langle l,\ a,\ r\rangle \Rightarrow x = a)$$

See elsewhere [17,18] for *insert* and *delete*. Note that *isin* should return the new tree as well to achieve amortized logarithmic complexity. This is awkward in a functional language and gives the data structure an imperative flavour.

The verification is more demanding than before and we present all the required lemmas in Fig. 5. Lemmas (15)–(20) extend our lemma library in Fig. 3 but are only required for splay trees. With the help of these lemmas, the proofs of (1)–(3) are automatic.

$$\uparrow (x \cdot xs) \wedge y \leq x \Longrightarrow \uparrow (y \cdot xs) \quad (15)$$
$$\uparrow (xs \mathbin{@} [x]) \wedge x \leq y \Longrightarrow \uparrow (xs \mathbin{@} [y]) \quad (16)$$
$$\uparrow (x \cdot xs) \Longrightarrow ins_{list}\ x\ xs = x \cdot xs \quad (17)$$
$$\uparrow (xs \mathbin{@} [x]) \Longrightarrow ins_{list}\ x\ xs = xs \mathbin{@} [x] \quad (18)$$
$$\uparrow (x \cdot xs) \Longrightarrow del_{list}\ x\ xs = xs \quad (19)$$
$$\uparrow (xs \mathbin{@} [x]) \Longrightarrow del_{list}\ x\ (xs \mathbin{@} ys) = xs \mathbin{@} del_{list}\ x\ ys \quad (20)$$

$$(splay\ a\ t = \langle\rangle) = (t = \langle\rangle)$$
$$(splay\_max\ t = \langle\rangle) = (t = \langle\rangle)$$
$$splay\ x\ t = \langle l, a, r\rangle \wedge \uparrow \lfloor t \rfloor \Longrightarrow (x \in elems\ \lfloor t \rfloor) = (x = a)$$
$$\lfloor splay\ x\ t \rfloor = \lfloor t \rfloor$$
$$\uparrow \lfloor t \rfloor \wedge splay\ x\ t = \langle l, a, r\rangle \Longrightarrow \uparrow (\lfloor l \rfloor \mathbin{@} x \cdot \lfloor r \rfloor)$$
$$splay\_max\ t = \langle l, a, r\rangle \wedge \uparrow \lfloor t \rfloor \Longrightarrow \lfloor l \rfloor \mathbin{@} [a] = \lfloor t \rfloor \wedge r = \langle\rangle$$

Fig. 5. Lemmas for splay tree verification

6 Maps

6.1 Specifications

Search trees can implement maps as well as sets. Although sets are a special case of maps, we presented sets first because their simplicity facilitates the explanation of the basic concepts. Now we present the modifications required for maps. An implementation of maps must provides a type $('a,'b)\ t$ (where $'a$ are the keys and $'b$ the values) with the operations

$$empty :: ('a,'b)\ t$$
$$update :: 'a \Rightarrow 'b \Rightarrow ('a,'b)\ t \Rightarrow ('a,'b)\ t$$
$$delete :: 'a \Rightarrow ('a,'b)\ t \Rightarrow ('a,'b)\ t$$
$$lookup :: ('a,'b)\ t \Rightarrow 'a \Rightarrow 'b\ option$$

where **datatype** $'a\ option = None \mid Some\ 'a$ is predefined. Function *lookup* also plays the role of the abstraction function. In addition there is a data type invariant $invar :: ('a,'b)\ t \Rightarrow bool$. The specification of maps is shown in Fig. 6 (corresponding to Fig. 1). It uses the function update notation

$$f(a := b) = (\lambda x.\ \mathit{if}\ x = a\ \mathit{then}\ b\ \mathit{else}\ f\ x)$$

$$invar\ m \Longrightarrow lookup\ (update\ a\ b\ m) = (lookup\ m)(a := Some\ b)$$
$$invar\ m \Longrightarrow lookup\ (delete\ a\ m) = (lookup\ m)(a := None)$$
$$invar\ m \Longrightarrow invar\ (update\ a\ b\ m)$$
$$invar\ m \Longrightarrow invar\ (delete\ a\ m)$$

Fig. 6. Specification of map implementations

Now we assume that the keys are linearly ordered. Search trees are abstracted to a list of key-value pairs sorted by their keys. The auxiliary functions $\uparrow$, ins_{list} and del_{list} are replaced by

$\uparrow_1 :: ('a \times 'b)\ list \Rightarrow bool$
$upd_{list} :: 'a \Rightarrow 'b \Rightarrow ('a \times 'b)\ list \Rightarrow ('a \times 'b)\ list$
$del_{list} :: 'a \Rightarrow ('a \times 'b)\ list \Rightarrow ('a \times 'b)\ list$

- $\uparrow_1\ xs = \uparrow (map\ fst\ xs)$ where $fst\ (a, b) = a$.
- $upd_{list}\ a\ b$ updates a sorted (w.r.t. $\uparrow_1$) list by either inserting (a, b) at the correct position (w.r.t. $<$) if no $(a, \_)$ is in the list, or replacing the first $(a, \_)$ by (a, b) otherwise.
- $del_{list}\ a$ deletes the first occurrence of a pair $(a, \_)$ from a list.

$$
\begin{aligned}
&invar\ t \Longrightarrow \lfloor update\ a\ b\ t \rfloor = upd_{list}\ a\ b\ \lfloor t \rfloor \\
&invar\ t \Longrightarrow \lfloor delete\ a\ t \rfloor = del_{list}\ a\ \lfloor t \rfloor \\
&invar\ t \Longrightarrow lookup\ t\ a = map\_of\ \lfloor t \rfloor\ a \\
&invar\ t \Longrightarrow inv\ (update\ a\ b\ t) \\
&invar\ t \Longrightarrow inv\ (delete\ a\ t)
\end{aligned}
$$

Fig. 7. Specification of map implementations over ordered types

Our second specification of maps (over a linearly ordered type $'a$) is shown in Fig. 7 (corresponding to Fig. 2). It is again based on an inorder function:

$inorder :: ('a,'b)\ t \Rightarrow ('a \times 'b)\ list$

Again, we abbreviate $inorder\ t$ by $\lfloor t \rfloor$.

The search tree invariant is now expressed as $\uparrow_1 \lfloor t \rfloor$. Structural invariants can be added via the specification function $inv :: ('a,'b)\ t \Rightarrow bool$ and we define

$invar\ t = (inv\ t \wedge \uparrow_1 \lfloor t \rfloor)$

The first three propositions in Fig. 7 express functional correctness of *update*, *delete* and *lookup* w.r.t. upd_{list}, del_{list} and *map_of*. The latter is a predefined function on key-value lists:

$map\_of\ [] = (\lambda\_.\ None)$
$map\_of\ ((a, b) \cdot ps) = (map\_of\ ps)(a := b)$

The next two propositions demand that *inv* is invariant. It is easy to show that Fig. 7 implies Fig. 6.

6.2 Proof Automation

Figure 8 (corresponding to Fig. 3) shows the set of lemmas used to automate the correctness proofs of implementations of maps. There are no lemmas about $\uparrow_1$ because its definition is simply unfolded and the lemmas (6)–(9) about $\uparrow$ apply.

The litmus tests for the lemma collection are the correctness proofs for the map-variants of all the search trees discussed in Sect. 5. The code of the map-variants is structurally the same as their set-counterparts. The same is true for the lemmas required in the verification. In the end, the proofs of the map-variants are just as automatic as the ones of their set-counterparts.

$$\uparrow_1 (ps \mathbin{@} [(a, b)]) \Longrightarrow upd_{list}\ x\ y\ (ps \mathbin{@} (a, b) \cdot qs) =$$
$$(\mathit{if}\ x < a\ \mathit{then}\ upd_{list}\ x\ y\ ps \mathbin{@} (a, b) \cdot qs\ \mathit{else}\ ps \mathbin{@} upd_{list}\ x\ y\ ((a, b) \cdot qs))$$

$$\uparrow_1 (ps \mathbin{@} (a, b) \cdot qs) \Longrightarrow del_{list}\ x\ (ps \mathbin{@} (a, b) \cdot qs) =$$
$$(\mathit{if}\ x < a\ \mathit{then}\ del_{list}\ x\ ps \mathbin{@} (a, b) \cdot qs\ \mathit{else}\ ps \mathbin{@} del_{list}\ x\ ((a, b) \cdot qs))$$

$$map\_of\ (ps \mathbin{@} qs)\ x =$$
$$(\mathit{case}\ map\_of\ ps\ x\ \mathit{of}\ None \Rightarrow map\_of\ qs\ x \mid Some\ y \Rightarrow Some\ y)$$
$$\uparrow (a \cdot map\ fst\ ps) \wedge x < a \Longrightarrow map\_of\ ps\ x = None$$
$$\uparrow (map\ fst\ ps \mathbin{@} [a]) \wedge a \leq x \Longrightarrow map\_of\ ps\ x = None$$

Fig. 8. Lemmas for upd_{list}, del_{list} and *map_of*

7 Conclusion

Our proof method works well because all the trees we considered follow the same ordering principle: inorder traversal yields a sorted list. Two referees suspected that for Trie-like trees [7] it would not work so well. I formalized binary trees where nodes are addressed by bit lists indicating the path to the node. A direct correctness proof is easy. The methods of this paper can also be applied (the list of addresses of the nodes in a tree, in prefix order, is lexicographically ordered) but the proof is more complicated and less automatic. Our approach seems overkill and awkward for such search trees.

Acknowledgement. Daniel Stüwe found and corrected the two invariant-related bugs in AA trees and proved preservation of the invariant under deletion for AA trees and 1-2 Brother trees.

References

1. Andersson, A.: Balanced search trees made simple. In: Dehne, F., Sack, J.-R., Santoro, N. (eds.) WADS 1993. LNCS, vol. 709, pp. 60–71. Springer, Heidelberg (1993)
2. Appel, A.: Efficient verified red-black trees (2011)
3. Bayer, R.: Symmetric binary B-trees: Data structure and maintenance algorithms. Acta Inform. **1**, 290–306 (1972)
4. Clochard, M.: Automatically verified implementation of data structures based on AVL trees. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 167–180. Springer, Heidelberg (2014)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1990)
6. Filliâtre, J.-C., Letouzey, P.: Functors for proofs and programs. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 370–384. Springer, Heidelberg (2004)
7. Fredkin, E.: Trie memory. Commun. ACM **3**(9), 490–499 (1960)
8. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: 19th Annual Symposium on Foundations of Computer Science, pp. 8–21. IEEE Computer Society (1978)

9. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 100–115. Springer, Heidelberg (2013)
10. Hinze, R.: Purely functional 1-2 brother trees. J. Funct. Program. **19**(6), 633–644 (2009)
11. Hoare, C.: Proof of correctness of data representations. Acta Inform. **1**, 271–281 (1972)
12. Hoffmann, C.M., O'Donnell, M.J.: Programming with equations. ACM Trans. Program. Lang. Syst. **4**(1), 83–112 (1982)
13. Jones, C.B.: Software Development. A Rigourous Approach. Prentice Hall, London (1980)
14. Kahrs, S.: Red black trees with types. J. Funct. Program. **11**(4), 425–432 (2001)
15. Martí-Oliet, N., Palomino, M., Verdejo, A.: A tutorial on specifying data structures in Maude. Electr. Notes Theor. Comput. Sci. **137**(1), 105–132 (2005)
16. Nipkow, T.: Are homomorphisms sufficient for behavioural implementations of deterministic and nondeterministic data types? In: Brandenburg, F.J., Wirsing, M., Vidal-Naquet, G. (eds.) STACS 1987. LNCS, vol. 247, pp. 260–271. Springer, Heidelberg (1987)
17. Nipkow, T.: Splay tree. Archive of Formal Proofs, Formal proof development, August 2014. http://isa-afp.org/entries/Splay_Tree.shtml
18. Nipkow, T.: Amortized complexity verified. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 310–324. Springer, Heidelberg (2015)
19. Nipkow, T., Klein, G.: Concrete Semantics with Isabelle/HOL. Springer (2014). http://concrete-semantics.org
20. Nipkow, T., Kunčar, O., Pusch, C.: AVL trees. Archive of Formal Proofs, Formal proof development, March 2004. http://isa-afp.org/entries/AVL-Trees.shtml
21. Nipkow, T., Paulson, L.C., Wenzel, M. (eds.): Isabelle/HOL. A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
22. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1998)
23. Ottmann, T., Six, H.W.: Eine neue Klasse von ausgeglichenen Binärbäumen. Angewandte Informatik **18**(9), 395–400 (1976)
24. Ottmann, T., Wood, D.: 1-2 brother trees or AVL trees revisited. Comput. J. **23**(3), 248–255 (1980)
25. Ragde, P.: Simple balanced binary search trees. In: Caldwell, J., Hölzenspies, P., Achten, P. (eds.) Trends in Functional Programming in Education, EPTCS, vol. 170, pp. 78–87 (2014)
26. Ralston, R.: ACL2-certified AVL trees. In: Proceedings of 8th International Workshop ACL2 Theorem Prover and its Applications, pp. 71–74. ACM (2009)
27. Reade, C.: Balanced trees with removals: an exercise in rewriting and proof. Sci. Comput. Program. **18**(2), 181–204 (1992)
28. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. ACM **32**(3), 652–686 (1985)
29. Turbak, F.: CS230 Handouts – Spring 2007 (2007). http://cs.wellesley.edu/cs230/spring07/handouts.html
30. Weiss, M.A.: Data Structures and Algorithm Analysis, 2nd edn. Benjamin/Cummings, Redwood City (1994)