OPEN ACCESS
OPUS AUGSBURG

Universität Augsburg
Universitätsbibliothek

Separating separation logic – modular verification of red-black trees

Gerhard Schellhorn, Stefan Bodenmüller, Martin Bitterlich, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Schellhorn, Gerhard, Stefan Bodenmüller, Martin Bitterlich, and Wolfgang Reif. 2023. "Separating separation logic – modular verification of red-black trees." Lecture Notes in Computer Science 13800: 129–47. https://doi.org/10.1007/978-3-031-25803-9_8.

Nutzungsbedingungen / Terms of use: **licgercopyright**

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/

Separating Separation Logic – Modular Verification of Red-Black Trees *

Gerhard Schellhorn, Stefan Bodenmüller, Martin Bitterlich, and Wolfgang Reif

Institute for Software and Systems Engineering
University of Augsburg, Germany
{schellhorn,stefan.bodenmueller,martin.bitterlich,reif}
@informatik.uni-augsburg.de

Abstract. Interactive theorem provers typically use abstract algebraic data structures to focus on algorithmic correctness. Verification of programs in real programming languages also has to deal with pointer structures, aliasing and, in the case of C, memory management. While progress has been made by using Separation Logic, direct verification of code still has to deal with both aspects at once. In this paper, we show a refinement-based approach that separates the two issues by using a suitable modular structure.
We exemplify the approach with a correctness proof for red-black trees, demonstrating that our approach can generate efficient C code that uses parent pointers and avoids recursion. The proof is split into a large part almost identical to high-level algebraic proofs and a separate small part that uses Separation Logic to verify primitive operations on pointer structures.

Keywords: Hierarchical Components, Refinement, Verification, Separation Logic, Efficient C Code, Red-Black Trees

1 Introduction

Interactive theorem provers typically use high-level algebraic data structures like lists, sets, or trees to verify the correctness of algorithms conceptually. Code generated from such algorithms is typically purely functional and often not very efficient. Side effects, aliasing, or memory allocation are absent, except when a heap with allocation and deallocation is explicitly modeled, which is rarely done when studying algorithmic correctness.

However, verification of programs in real programming languages has to deal with the fact that all non-primitive data types are represented as pointer structures, and destructive operations are often used to improve efficiency. The most popular concept to handle these issues is to use Separation Logic, which moves the specification of a heap structure into the semantics of the logic. Provers that

* Partly supported by the Deutsche Forschungsgemeinschaft (DFG), “Verifikation von Flash-Dateisystemen” (grants RE828/13-1 and RE828/13-2).

target the verification of C, Java, or Rust programs like VeriFast [20] or Viper [26] are directly based on it. Many interactive theorem provers now support a library for Separation Logic similar to the one we give in Section 3.2.

However, direct verification of algorithms given e.g. in C still suffers from the complexity of conceptual correctness arguments being intertwined with questions about pointer aliasing and side effects.

This paper contributes an approach that modularizes the verification effort of a library implementation of sets by red-black trees into two independent parts: a bigger one that deals with functional correctness on an algebraic level, and a smaller part that is independent of the first and deals with mapping small operations (like removing a leaf or rotating at a path) on abstract data structures to operations on pointer structures. The approach separates the use of Separation Logic from the proof of conceptual correctness by restricting it to the latter part. It is based on components with sequential programs linked by data refinement, supported natively in our theorem prover KIV.

We have chosen red-black trees as they offer good worst-case guarantees for the operations search, insert and remove. Their verification on an algebraic level is already non-trivial. However, our goal was to verify an efficient version such that the resulting code is on par with standard C code implementations. This mandates that our final implementation uses parent-pointers and avoids recursion to be as efficient as possible. Our implementation is based on the pseudocode given in [9].

Red-black trees are also useful in the Flashix project [5], where we have implemented and verified a realistic file system for flash memory which can be used as a kernel module in Linux. There, red-black are used to balance erase counts of raw flash blocks in the wear leveling algorithm. Since verification there is also based on the concept of components connected by refinement, we could replace an unverified external C library with the verified implementation described here.

This paper is organized as follows. Sec. 2 introduces characteristic features of our theorem prover KIV that comprises both a specification and programming language. Sec. 3 presents the algebraic data types to describe a red-black tree and the explicit heap that is used to reason about pointer-based programs. Sec. 4 explains how a software system can be broken down into hierarchical components that refine an abstract system description to a realistic implementation.

Sec. 5 highlights the implementation split into a common part and elementary operations that can also be performed on a pointer structure. Sec. 6 follows with an overview of some key properties for verification. Sec. 7 presents existing approaches and draws a comparison to them.

2 Background

To develop the necessary formal specifications and prove that our implementation follows them, we use the theorem prover KIV, which provides interactive verification using a sequent calculus with explicit proof trees. The basic logic of the specification language is higher order logic (HOL), recently extended from

monomorphic to polymorphic types. KIV supports an imperative programming language with recursive procedures and nondeterminism. Details on the syntax can be found in [33], Fig. 4 shows a procedure definition. The arguments of a procedure **proc#**(*in*; *ref*; *out*) are grouped into sequences of input, reference, and output parameters. KIV does not support global variables, these must be added explicitly as reference parameters.

Reasoning about sequential programs in KIV is done with a weakest precondition calculus, borrowing notation from Dynamic Logic (DL) [18], including its two standard modalities: the formula $[\alpha]\varphi$ (*box*) denotes that, for every terminating run of α, the final state must satisfy φ, corresponding to the weakest liberal precondition $wlp(\alpha, \varphi)$. The formula $\langle\alpha\rangle\varphi$ (*diamond*) guarantees that there is a terminating execution of α that establishes φ. Finally, the formula $\langle\!|\alpha|\!\rangle\,\varphi$ (*strong diamond*) states that all runs of α terminate with a final state satisfing φ (weakest precondition $wp(\alpha, \varphi)$). Partial and total correctness of a program α with respect to pre-/post-conditions *pre*/*post* is written $pre \rightarrow [\alpha]\ post$ and $pre \rightarrow \langle\!|\alpha|\!\rangle\, post$, respectively. The calculus is more expressive than standard Hoare-like program logics since it allows to combine and nest program formulas. This allows e.g. to establish a relation between two programs, which will be useful in defining proof obligations for refinement, cf. Sec. 4.

The main proof technique for verifying program correctness in KIV is *symbolic execution*. Each symbolic execution step calculates the strongest postconditions of the first program statement from the preconditions. When the symbolic execution of the program is completed, the goal is reduced to predicate logic, where proof automation is achieved via rewrite rules and heuristics, see [33].

3 Structured Specifications of Algebraic Data Types

In KIV, structured algebraic specifications are used to build a hierarchy of data type definitions. Primitive data types may be generated freely or non-freely. Specifications can be augmented by additional functions and combined using standard structuring operations like enrichment, union, and renaming. It is also possible to specify parameterized data types that can be instantiated explicitly.

3.1 Algebraic Red Black Tree Definition

The standard approach for proving the correctness of algorithms using complex data structures is to specify the data structures algebraically. Red-black trees [17,35] can be defined as a polymorphic free data type *rbtree*('*a*), using a constant constructor `SENTINEL` (representing the leaves of the tree) and a non-constant constructor `Node`.

$$\mathit{rbtree}('a) \;:=\; \texttt{SENTINEL} \mid \texttt{Node}(\texttt{.elem} : \; 'a \;;\; \texttt{.color} : \; \mathit{rbcolor} \;; \; \texttt{.left} : \mathit{rbtree}('a) \;;\; \texttt{.right} : \mathit{rbtree}('a))$$

Nodes have a color (either `RED` or `BLACK`, defined by the enumeration type *rbcolor*), a left and a right subtree, and an element of generic type '*a*. These

fields can be accessed via the postfix selector functions `.elem`, `.color`, `.left`, and `.right`. A type variable $'a$ for the type of elements stored in the tree is used in the definition. So in principle, the data type can be used with any element type. However, to express the properties of binary search trees, a generic, totally ordered elements type *tord* (with $<$) is used. The resulting tree type is written *rbtree*(*tord*). The specification can be instantiated later as needed by suitable types, e.g. natural numbers or integers. When such a parameter is instantiated, KIV generates proof obligations to ensure that the instantiated type satisfies the assumed properties (in this case, a total order over the type).

For a free data type specification, KIV generates all necessary axioms, as well as update functions (written e.g. *rbt*`.color:=` *newcol*), including their definitions. Note that selector (and update) functions are not given axioms for all arguments: `SENTINEL.color` is left unspecified. The semantic function in a model is still total, and `SENTINEL.color` may be any value, following the standard loose approach to semantics. However, KIV attaches a *domain* to the function for use in programs. Calling `.color` outside of its domain in a program (here: with `SENTINEL`, where it is "undefined") will raise an exception. Therefore, proving the correct use of the data type in programs includes showing the absence of such exceptions, i.e. one has to prove that all operations are called with arguments within their respective domain.

3.2 Modeling the Heap and Separation Logic

Reasoning about destructive pointer algorithms requires to model the heap, either implicitly as part of the semantics of formulas or explicitly as an algebraic data type. In KIV, the latter approach is realized: heaps are specified as a polymorphic non-free data type $heap('a)$. A heap can be considered a partial function mapping references r (of type $ref('a)$) to objects *obj* of a generic type $'a$, where allocation of references is explicit and the reference type contains a distinguished element `null` that is never allocated (representing the null pointer).

The $heap('a)$ data type is inductively generated by the constant $\emptyset$ representing the empty heap, allocating a new reference r (written h `++` r), or updating an allocated location r with a new object *obj* (written $h[r$ `:=` $obj]$). Again, the object type is not specified further so that the heap specification can be used with any concrete object type (for red-black trees, the type *rbnode* represents individual nodes of the tree, see Sec. 5).

A predicate $r \in h$ checks whether a reference is allocated in a heap, and a function $h[r]$ is used to lookup objects in a heap (this corresponds to dereferencing a pointer). References can also be deallocated by the function h `--` r.

Similar to the selector functions of free data types, the constructor functions as well as lookup and deallocation are partial functions in order to specify valid accesses to the heap: accesses to the heap with the `null` reference are always undefined ($r \neq$ `null`), allocation is allowed with a new reference ($\neg\ r \in h$) only. Lookup, update, and deallocation require an allocated reference $r \in h$.

In KIV, all parameters of procedures are explicit. Hence, when reasoning about pointer-based programs, the heap must be an explicit parameter of the

program as well. To facilitate the verification of such programs, we built a simple library for Separation Logic (SL) [32] in KIV, similar to the libraries of Isabelle [24] and Coq [8]. We give some information, to explain the notation used in the following. SL formulas are encoded using heap predicates $hP : heap('a) \to bool$. A heap predicate describes the structure of a heap h. At its simplest, h is the empty heap `emp`:

$$\texttt{emp}(h) \leftrightarrow h = \emptyset$$

The maplet $r \mapsto obj$ describes a singleton heap containing only one reference r mapping to an object obj. It is defined as a higher-order function of type $(ref('a) \times 'a) \to heap('a) \to bool$:

$$(r \mapsto obj)(h) \leftrightarrow h = (\emptyset \texttt{ ++ } r)[r \texttt{ := } obj] \land r \neq \texttt{null}$$

More complex heaps can be described using the separating conjunction $hP_0 \texttt{ * } hP_1$ asserting that the heap consists of two disjoint parts, one satisfying hP_0 and one satisfying hP_1, respectively. Since it connects two heap predicates, it is defined as a function with type $(heap('a) \to bool) \times (heap('a) \to bool) \to (heap('a) \to bool)$:

$$(hP_0 \texttt{ * } hP_1)(h) \leftrightarrow \exists\, h_0, h_1.\ h_0 \perp h_1 \land h = h_0 \cup h_1 \land hP_0(h_0) \land hP_1(h_1)$$

Besides the basic SL definitions, the KIV library contains various abstractions of commonly used pointer data structures like singly-/doubly-linked lists or binary trees. These abstractions allow to prove the functional correctness (incl. memory safety) of algorithms on pointer structures against their algebraic counterparts. We will demonstrate this approach for a red-black tree implementation.

4 Modular Software Systems

For the development of complex software systems in KIV, we use the concept of hierarchical components combined with the contract approach to data refinement [10]. A component is an abstract data type $(ST, \texttt{Init}, (\texttt{Op}_j)_{j \in J})$ consisting of a set of states ST, a set of initial states $\texttt{Init} \subseteq ST$, and a set of operations $\texttt{Op}_j \subseteq In_j \times ST \times ST \times Out_j$. An operation $\texttt{Op}_j$ takes inputs In_i and outputs Out_j and modifies the state of the component. Operations are specified with contracts using the operational approach of ASMs [6]: for an operation $\texttt{Op}_j$, we give a precondition pre_j and a program α_j in the form of a procedure declaration $\mathbf{op}_j\#(in_j; st; out_j)$ **pre** pre_j $\{\alpha_j\}$. The program α_j is given in KIV's imperative programming language and establishes the postcondition of the operation. Instead of defining initial states directly, we also give a procedure declaration $\mathbf{init}\#(in_{init}; st; out_{init})$ $\{\alpha_{init}\}$.

Components are distinguished between specifications and implementations. The former are used to model the functional requirements of a (sub-)system and are typically kept as simple as possible by heavily utilizing algebraic functions and non-determinism. The approach is as general as specifying pre- and postconditions since the program **choose** st', out' **with** $post(st', out')$ **in** $st, out := st', out'$ can be used to establish any postcondition $post$ over state st and output

out. Implementations are typically deterministic and use constructs only that allow generating executable Scala or C code from them with our code generator.

The functional correctness of implementation components is then proven by a data refinement of the corresponding specification components (we write $\mathtt{C} \le \mathtt{A}$ if $\mathtt{C} = (ST^{\mathtt{C}}, \mathtt{Init}^{\mathtt{C}}, (\mathtt{Op}_j^{\mathtt{C}})_{j \in J})$ is a refinement of $\mathtt{A} = (ST^{\mathtt{A}}, \mathtt{Init}^{\mathtt{A}}, (\mathtt{Op}_j^{\mathtt{A}})_{j \in J})$ where C and A have the same set of operations J). Proofs for such a refinement are done with a forward simulation $R \subseteq ST^{\mathtt{A}} \times ST^{\mathtt{C}}$ using commuting diagrams. This results in correctness proof obligations for all $j \in J$ (an extra obligation ensures that $\mathtt{Init}^{\mathtt{A}}$ and $\mathtt{Init}^{\mathtt{C}}$ establish matching states).

$$R(st^{\mathtt{A}}, st^{\mathtt{C}}) \land pre_j^{\mathtt{A}}(st^{\mathtt{A}})$$
$$\to \langle\!|\mathbf{op}_j^{\mathtt{C}}\#(in_j; st^{\mathtt{C}}; out_j)|\!\rangle\, \langle \mathbf{op}_j^{\mathtt{A}}\#(in_j; st^{\mathtt{A}}; out'_j)\rangle (R(st^{\mathtt{A}}, st^{\mathtt{C}}) \land out_j = out'_j)$$

Note that the obligation refers to two procedure runs ($\mathbf{op}_j^{\mathtt{C}}\#$ and $\mathbf{op}_j^{\mathtt{A}}\#$), stringing together a *strong diamond* and a *diamond* program formula. Thus, $st^{\mathtt{A}}$ and $st^{\mathtt{C}}$ in the postcondition of the obligation refer to the changed states after the runs of $\mathbf{op}_j^{\mathtt{A}}\#$ and $\mathbf{op}_j^{\mathtt{C}}\#$, respectively. Informally, one has to prove that, when starting in R-related states, for each run of an operation $\mathbf{op}_j^{\mathtt{C}}\#$ of C, there must be a matching run of $\mathbf{op}_j^{\mathtt{A}}\#$ of A that maintains $R(st^{\mathtt{A}}, st^{\mathtt{C}})$ with the same inputs and outputs. The obligation also requires to show that the precondition $pre_j^{\mathtt{A}}(st^{\mathtt{A}})$ is strong enough to establish the precondition $pre_j^{\mathtt{C}}(st^{\mathtt{C}})$ if $R(st^{\mathtt{A}}, st^{\mathtt{C}})$ holds. This obligation is implicit as the call rule creates this premise for a procedure with a precondition.

For each component, invariant formulas $inv(st)$ over the state st can be given, which must be maintained by all $(\mathtt{Op}_j)_{j \in J}$. This simplifies (or even makes it possible in the first place) to prove the correctness proof obligations of a refinement as invariants $inv^{\mathtt{A}}(st^{\mathtt{A}})$ and $inv^{\mathtt{C}}(st^{\mathtt{C}})$ are added as assumptions. If an invariant is given for a component, additional proof obligations for all its operations are generated that ensure that the invariant holds. Additionally, one can give an individual postcondition $post_j(st)$ for an operation, which extends its invariant contract.

$$pre_j(st) \land inv(st) \to \langle\!|\mathbf{op}_j\#(in_j; st; out_j)|\!\rangle\, (inv(st) \land post_j(st))$$

These invariant contracts can be applied when proving the refinement proof obligations and may further simplify the proofs since symbolic execution of the operation can be avoided.

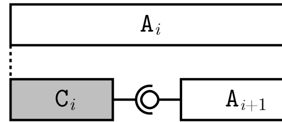

Fig. 1: Data refinement with subcomponents.

To facilitate the development of larger systems, we introduced a concept of modularization in the form of *subcomponents*. A component (usually an implementation) can use one or more components as subcomponents (usually specifications). The client component cannot access the state of its subcomponents directly but only via calls to the interface operations of the subcomponents. Using subcomponents, a refinement hierarchy is composed of multiple refinements like in Fig. 1. A specification component $\mathtt{A}_i$ is refined by an implementation $\mathtt{C}_i$ (dotted lines in Fig. 1) that uses

state $\quad rbs : set(tord)$

init#()
  initialization
$\{ \; rbs := \emptyset \; \}$

insert#(*elem*; ; *exists*)
  interface
$\{ \; exists := elem \in rbs, \;\; rbs := rbs \cup \{elem\} \; \}$

remove#(*elem*; ; *exists*)
  interface
$\{ \; exists := elem \in rbs, \;\; rbs := rbs \setminus \{elem\} \; \}$

isEmpty#(; ; *empty*)
  interface
$\{ \; empty := rbs = \emptyset \; \}$

lookup#(*elem*; ; *exists*)
  interface
$\{ \; exists := elem \in rbs \; \}$

getMin#(; ; *elem*)
  interface
  pre $rbs \neq \emptyset$
$\{ \; elem \;\; := rbs.\mathtt{min} \; \}$

Fig. 2: Abstract representation of red-black trees: the component `RBSET`.

a specification $\mathtt{A}_{i+1}$ as a subcomponent (—◎— in Fig. 1, we write $\mathtt{C}_i(\mathtt{A}_{i+1})$ for this subcomponent relation). This pattern then repeats in the sense that $\mathtt{A}_{i+1}$ is refined further by an implementation $\mathtt{C}_{i+1}$ that again uses a subcomponent $\mathtt{A}_{i+2}$ and so on. If it is not the top-level specification, $\mathtt{A}_i$ may also be used as a subcomponent of an implementation $\mathtt{C}_{i-1}$. The complete implementation of the system then results from composing all individual implementation components $\mathtt{C}_0(\mathtt{C}_1(\mathtt{C}_2(...)))$. In [13] we have shown that $\mathtt{C} \leq \mathtt{A}$ implies $\mathtt{M}(\mathtt{C}) \leq \mathtt{M}(\mathtt{A})$ for a client component `M` which ensures that the composed implementation is a correct refinement of its top-level specification $\mathtt{A}_0$, i.e. $\mathtt{C}_0(\mathtt{C}_1(\mathtt{C}_2(...))) \leq \mathtt{A}_0$. This allows us to divide a complex refinement task into multiple, more manageable ones, as demonstrated in the following sections for a red-black tree implementation.

5 Implementation of Destructive Red-Black Trees

Red-black trees are typically used as an efficient data structure for ordered sets (or multisets). In order to abstract from the complex implementation details of red-black trees (traversal, rotations, . . .), the simple specification component `RBSET` given in Fig. 2 can be used. Other components then can use `RBSET` as a subcomponent, which simplifies formal reasoning about the client component while the resulting system still uses an efficient heap implementation.

The state of `RBSET` is just a set of elements that are totally ordered. It is determined by a state variable *rbs* of type *set*(*tord*), where *set* is a polymorphic non-free data type (cf. Sec. 3.2), and a strict total order is given over elements of type *tord*. Initially, *rbs* is empty ($\emptyset$), and it can be modified by inserting or removing elements *elem* (by **insert#** and **remove#**). Additional interface procedures check whether the set is empty and whether an element is in the set. The minimal element

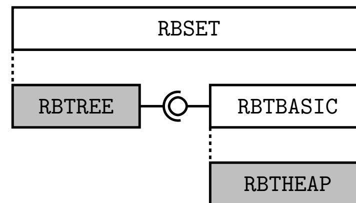

Fig. 3: The refinement hierarchy for red-black trees.

can be selected. In Fig. 2 and below, state variables are omitted from the parameters of operation declarations but are implicitly added as reference arguments.

This component is refined by a pointer-based implementation of red-black trees. However, this refinement is split into two parts to reduce the complexity of the necessary reasoning about the heap done with Separation Logic. The result is the refinement hierarchy shown in Fig. 3. In the first refinement step `RBTREE`(`RBTBASIC`) $\leq$ `RBSET`, we show that the a red-black tree can implement the set abstraction and that this implementation maintains all red-black tree properties (cf. Sec. 6). But instead of using a heap data structure, we do this using the algebraic datatype *rbtree* presented in Sec. 3.1.

The second refinement step `RBTHEAP` $\leq$ `RBTBASIC` proves that a heap implementation conforms to this algebraic datatype. The goal of this separation is to keep the operations of `RBTBASIC` (and hence those of `RBTHEAP`) as simple as possible. The more complex algorithmic parts are handled in `RBTREE` while `RBTBASIC` only provides an interface for primitive manipulations of *rbtree*. This includes for example the insertion of an element at a given point within the tree, or a single left- or right-rotation of one particular subtree. To specify a location in the tree on the abstract level, we simply use paths, i.e. lists of an enumeration type $lrdesc := \texttt{LEFT} \mid \texttt{RIGHT}$, and define functions like $p \in rbt$, $rbt[p]$ and $rbt[p := rbt']$ that check a path p to be in a tree rbt, select the subtree at a path p, or update the subtree at a path p with a new tree rbt', respectively.

When used directly, these operations are inefficient since `RBTBASIC` has to traverse the paths at every access. However, note that we can already generate Scala code from `RBTREE`(`RBTBASIC`): the resulting code can be used for testing invariants and results of example runs, which often avoids unsuccessful proof attempts of properties that do not hold in the first place.

Since the algorithms on red-black trees use at most two paths and data refinement can refine state only (not input/output), we place two paths in the state of `RBTBASIC`. The implementation in `RBTHEAP` replaces these paths with two references that point to a heap storing individual tree nodes. The nodes of the implementation use parent pointers, so shortening or lengthening one of the two paths by one (which are operations of `RBTBASIC`) can be implemented by simply dereferencing a pointer. Hence, the state of the component `RBTBASIC` consists of an algebraic red-black-tree *rbt* using totally ordered elements *tord* and two paths *curPath* and *auxPath*. Most of the time *curPath* is used only, but for removal, it is necessary to store a second path *auxPath* that points to the element after the deleted one.

$$\textbf{state} \quad rbt : rbtree(tord),\ curPath : list(lrdesc),\ auxPath : list(lrdesc)$$

The corresponding state of `RBTHEAP` contains a heap *rbh* and a pointer *rootRef* to the root of the tree, together with pointers *curRef* and *auxRef* matching *curPath* and *auxPath*, respectively.

$$\begin{aligned}\textbf{state} \quad & rbh : heap(rbnode),\ rootRef : ref(rbnode),\\ & curRef : ref(rbnode),\ auxRef : ref(rbnode)\end{aligned}$$

```
remove#(elem; ; exists)
  interface
  post elems(rbt) = elems(rbt`) -- elem ∧ (exists ↔ elem ∈ elems(rbt`));
{
1   rbtbasic_reset#(); // curPath := [], auxPath := []
2   search#(elem); // sets curPath to position of elem or to a leaf
3   rbtbasic_isLeaf#(; ; exists); // exists := (rbt[curPath] = SENTINEL)
4   if ¬ exists then {
5     let doFix = ?, isLeftChild = ?, cond = ? in {
6       rbtbasic_hasLeft#(; ; cond); // cond := curPath + LEFT' ∈ rbt;
7       if ¬ cond then {
8         rbtbasic_replRight#(; ; doFix, isLeftChild);
9         // replace node at curPath with its right child
10        // and move curPath up one node
11      } else {
12        rbtbasic_hasRight#(; ; cond); // analogous to hasLeft
13        if ¬ cond then {
14          rbtbasic_replLeft#(; ; doFix, isLeftChild);
15        } else {
16          rbtbasic_initD#(); // auxPath := curPath
17          rbtbasic_right#(); // curPath := curPath + RIGHT
18          leftMost# // extend curPath to leftmost inner node
19          rbtbasic_getElem#(; ; elem); // elem := rbt[curPath].elem
20          rbtbasic_replRight#(; ; doFix, isLeftChild);
21          rbtbasic_setElemD#(elem); // rbt[auxPath].elem := elem
22      } };
23      if doFix then removeFixup#(isLeftChild);
24      // restore balance and red-black tree properties
25      exists := true; }
26  } else exists := false // element was not found
}
```

Fig. 4: RBTREE procedure for removing an element *elem* from the tree.

The heap stores nodes of type *rbnode*, which contain an element and a color like the Nodes of *rbtree* but use references that point to their left and right subtrees. A parent pointer is added to allow efficient traversal upwards in the tree.

$$rbnode := \texttt{Node}(\texttt{.elem} : tord;\ \texttt{.color} : rbcolor;\ \texttt{.parent} : ref(rbnode);\ \texttt{.left} : ref(rbnode);\ \texttt{.right} : ref(rbnode))$$

Figure 4 lists the implementation of **remove#** in the RBTREE component as an example of how this state is modified via the interface of RBTBASIC. For primitive RBTBASIC operations, the comments in green show their implementation. All operations start by resetting the paths to point to the root (line 1, [] denotes an empty list). Then the tree is traversed to an element of interest by performing a binary search in the procedure **search#**(*elem*). For removal, the element to be removed (*elem*) is searched and *curPath* will be updated to point to *elem* if it is found. When a SENTINEL is reached (checked in line 3), the search is stopped and the removal is aborted with *exists* := false (line 26). If *elem*

was found, the element must be replaced in order to restore the red-black tree properties. In case the node has a SENTINEL as left or right child (line 8 resp. 14), substitution of the node is performed simply by replacing it with the other child. Otherwise, *curPath* is stored in *auxPath* (line 16) and is then updated by **rbtbasic_right#** and **leftMost#** to point to the next greater element. This element is the minimal element of the right subtree of *elem* and thus cannot have a left child ($curPath + \texttt{LEFT}$ points to a SENTINEL). Therefore, the element can be moved to *auxPath* (line 21) and the tree at *curPath* can be replaced with its right child (line 20). Finally, the routine **removeFixup#** is called to fix up the tree starting at *curPath* since the removal may have broken the red-black tree properties and the tree may need to be rebalanced.

The RBTHEAP operations work similar to RBTBASIC but on the pointer structure. So instead of selecting a subtree at *curPath*, which requires a traversal of the complete path, the node at *curRef* is accessed simply by dereferencing the pointer ($rbh[curRef]$). Analogously, instead of adjusting the paths, e.g. with $curPath := curPath + \texttt{RIGHT}$, pointers are updated by following the references of the current node, e.g. with $curRef := rbh[curRef]\texttt{.right}$.

The algorithm is essentially the same as the one in [9]. However, we implement SENTINEL nodes as null pointers instead of using a dummy node that would be necessary to get the parent of a leaf. This does not change the *insert* algorithm, but results in *remove* working on the parent of the deleted node. Our algorithm therefore has to explicitly pass the information whether the left or right child was deleted (*isLeftChild* in Figure 4). On the other hand, our **removeFixup#** is not called when *curPath* points to a leaf (and only when there is something to fix), so the loop test in the original **removeFixup#** that checks for a leaf or the root (which can only be true in the first iteration) is removed. Otherwise the various cases and rotations are identical to [9].

6 Verification of Destructive Red-Black Trees

For verification, the functional properties must be specified as invariants of RBTREE. In between interface calls, *rbt* must be a valid red-black tree (expressed by $\texttt{isRbtree}(rbt)$) and must be a valid search tree, i.e. its elements must be ordered (described by the predicate $\texttt{isOrdered}(rbt)$).

A non-empty red-black tree is characterized by three main properties: the root of the tree is BLACK, both children of a RED node have to be BLACK, and each path of any node to a leaf must contain the same number of BLACK nodes.

$$\texttt{isRbtree}(rbt) \leftrightarrow \big(rbt = \texttt{SENTINEL} \vee (rbt\texttt{.color} = \texttt{BLACK} \wedge \texttt{redCorrect}(rbt,\ \texttt{RED}) \wedge \texttt{sameBlacks}(rbt))\big)$$

The predicate $\texttt{redCorrect}(rbt, parCol)$ ($parCol$ is the color of the parent node) specifies the first two properties, $\texttt{sameBlacks}(rbt)$ the last. Both are defined recursively over the structure of the tree, as is $\texttt{isOrdered}(rbt)$. As an example,

the axiom for a `RED` node for `redCorrect` is

$$\texttt{redCorrect}(\texttt{Node}(e,\ \texttt{RED},\ left,\ right),\ parCol) \leftrightarrow$$
$$parCol = \texttt{BLACK} \wedge \texttt{redCorrect}(left,\ \texttt{RED}) \wedge \texttt{redCorrect}(right,\ \texttt{RED})$$

For the deletion of a node, additional predicates must be defined that allow the properties to be violated at a specific path in the tree (we indicate them by attaching `D`). They allow the tree to be characterized during the procedure **remove#**. For instance, the following definition describes the violation of `redCorrect` at the current node ($path = \texttt{[]}$).

$$\texttt{redCorrectD}(\texttt{Node}(e,\ col,\ left,\ right),\ parCol,\ \texttt{[]}) \leftrightarrow$$
$$\texttt{redCorrect}(left,\ col) \wedge \texttt{redCorrect}(right,\ col)$$

While `isOrdered`(rbt) is maintained quite easily by the operations of `RBTREE`, e.g. **insert#** adds the new element directly at a position that maintains the order property, complex fixing mechanisms are necessary to re-establish `isRbtree`(rbt). The main proof effort is to show that these mechanisms are actually correct. To keep proof size manageable, we split the procedures into several subroutines, and formulated and proved contracts for these separately. We will not go into details of the verification of the routines (**insertFixup#** for insertion and **removeFixup#** for removal), the KIV code and proofs can be found online [23].

The refinement `RBTREE(RBTBASIC)` $\leq$ `RBSET` is proven by the following forward simulation, where `elems` calculates the set of elements stored in the tree.

abstraction relation $\quad rbs = \texttt{elems}(rbt)$

This simple abstraction allows to encode the set modifications of `RBSET` into the contracts of `RBTREE`. For example, the contract of **remove#** in Fig. 4 states that *elem* is removed from the tree ($\texttt{elems}(rbt) = \texttt{elems}(rbt') \texttt{ -- } elem$ where rbt' denotes the value of rbt just before the execution of the procedure). This modification happens within **rbtbasic_replRight#** and **rbtbasic_replLeft#**, the contracts of all other modifying auxiliary procedures, e.g. **removeFixup#**, ensure that they do not change the set of elements stored in the tree ($\texttt{elems}(rbt) = \texttt{elems}(rbt')$). Similar contracts are given for the other interface procedures, so the refinement is proven mainly by applying these contracts. Note that the refinement proofs do not require the invariant `isRbtree`(rbt) (an unbalanced tree would also refine a set correctly). However, they do require `isOrdered`(rbt) since otherwise tree search is not correct (and thus correctness of **remove#**, **lookup#**, and **getMin#** could not be shown).

For none of these proofs, it is necessary to reason about the heap implementation. In particular, the main invariant properties `isOrdered` and `isRbtree` are proved solely over algebraic trees. What remains to prove is that the pointer-based implementation in `RBTHEAP` is a correct refinement of `RBTBASIC`.

Most of the operations of `RBTBASIC` are just single assignments, for example recolorings of the node at *curPath* or one of its relatives, or changes of *curPath* or *auxPath*. `RBTHEAP` implements these operations analogously with lookups at *curRef* and updates of *curRef* or *auxRef* by following the parent- or child-

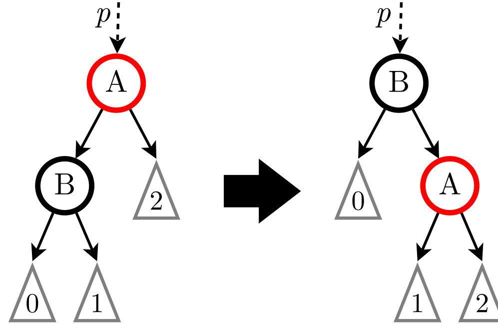

Fig. 5: Exemplary right-rotation at a path p.

```
rotateRight#(p)
  auxiliary
  pre p + LEFT ∈ rbt;
{
1   let rbt_0 = rbt[p] in
2   let rbt_1 = rbt_0.left in
3   let rbt_2 = Node(rbt_0.elem, rbt_0.color, rbt_1.right, rbt_0.right) in
4     rbt[p] := Node(rbt_1.elem, rbt_1.color, rbt_1.left, rbt_2)
}
```

Fig. 6: Procedure for right-rotations at a path p of component `RBTBASIC`.

pointers. The only more complex operations are rotations used in the fixing routines. For example, Fig. 5 shows the effect of a right-rotation at some location within the tree, i.e. at some path p (note that the left child B of A must be an actual `Node` for a valid right-rotation, while the subtrees 0, 1, and 2 can be `SENTINEL`s). In `RBTBASIC`, such rotations are performed using the auxiliary procedure **rotateRight#** listed in Fig. 6 (or a symmetric version **rotateLeft#** for left-rotations). The operation takes a path p as an argument and performs a right-rotation at the corresponding location. It selects the subtree at p, builds the rotated subtree, and then inserts it at p again (the program $rbt[p] := rbt_0$ is an abbreviation for $rbt := rbt[p \mathrel{\texttt{:=}} rbt_0]$ which replaces $rbt[p]$ with rbt_0 in rbt). The `RBTBASIC` interface then provides operations for rotations at different locations (at *curPath*, *auxPath*, or one of their relatives), all of which use one of the two auxiliary procedures with respective arguments.

Figure 7 shows the corresponding implementation of `RBTHEAP`. Instead of a path, it takes a reference *ref* as an input. The heap implementation performs the rotation by updating the pointers of the node at *ref* as well as those of its parent and left child. First, the link between the node at *ref* and its new left child is established (lines 2 and 3). Then the link between the new root of the subtree (*lRef*) and its new parent is created (lines 5-8). Finally, *ref* is linked to *lRef* as its new right child (lines 10 and 11). In contrast to the algebraic variant in Fig. 6 (the assignment in line 4 would copy the whole tree *rbt*), all updates are destructive. For example in C, the assignment in line 2 corresponds to a

```
    rotateRight#(ref)
      auxiliary
      pre ref ∈ rbh ∧ rbh[ref].left ∈ rbh;
    {
1     let lRef = rbh[ref].left in {
2       rbh[ref].left := rbh[lRef].right;
3       if rbh[lRef].right ≠ null then rbh[rbh[lRef].right].parent := ref;
4       if lRef ≠ null then rbh[lRef].parent := rbh[ref].parent;
5       if rbh[ref].parent ≠ null then {
6         if ref = rbh[rbh[ref].parent].right
7         then rbh[rbh[ref].parent].right := lRef;
8         else rbh[rbh[ref].parent].left := lRef;
9       } else rootRef := lRef;
10      rbh[lRef].right := ref;
11      if ref ≠ null then rbh[ref].parent := lRef;
    }}
```

Fig. 7: Procedure for right-rotations at a reference *ref* of component `RBTHEAP`.

statement `ref->left = lRef->right` where both `ref` and `lRef` as well as the fields `left` and `right` are pointers to a struct `rbnode`.

The refinement is proven using Separation Logic (see Sec. 3.2) and the following abstraction that does not refer to any red-black tree properties.

$$\textbf{abstraction relation} \quad \begin{aligned} & rbh[rootRef,\ curPath] = curRef \\ \wedge\ & rbh[rootRef,\ auxPath] = auxRef \\ \wedge\ & \texttt{abs}(rootRef,\ \texttt{null},\ rbt)(rbh) \end{aligned}$$

The first two formulas assert that the references *curRef* and *auxRef* correspond to the paths *curPath* and *auxPath*, respectively, i.e. they point to the same locations in the tree. Here, the function $rbh[r,\ p]$ yields the reference reached when traversing the heap rbh along the path p, starting at r. The heap predicate $\texttt{abs} : (ref(rbnode) \times ref(rbnode) \times rbtree(tord)) \to heap(rbnode) \to bool$ abstracts the pointer tree in *rbh* starting at *rootRef* to the algebraic tree *rbt*. The second *ref* (*rbnode*) argument specifies the parent of the root, for the complete tree it is `null`. `abs` is defined recursively over the structure of *rbt*:

$$\texttt{abs}(rootRef,\ pRef,\ \texttt{SENTINEL})(rbh) \leftrightarrow rootRef = \texttt{null} \wedge rbh = \emptyset$$

$$\begin{aligned} & \texttt{abs}(rootRef,\ pRef,\ \texttt{Node}(elem,\ col,\ l,\ r))(rbh) \leftrightarrow \\ & \quad \exists\ lRef,\ rRef.\ (\ \ (rootRef \mapsto \texttt{Node}(elem,\ col,\ pRef,\ lRef,\ rRef)) \\ & \qquad\qquad * \texttt{abs}(lRef,\ rootRef,\ l) \\ & \qquad\qquad * \texttt{abs}(rRef,\ rootRef,\ r))(rbh) \end{aligned}$$

For a `SENTINEL`, the heap *rbh* must be empty. This ensures the absence of memory leaks since one has to prove that all nodes have been deallocated when they are removed from the tree. For a `Node`, the heap is separated into three disjoint parts: a root node containing the same element and color as the algebraic node and two trees that abstract to the left and right algebraic subtree.

The abstraction relation uses ordinary conjunction, and we found it easy to support updating the first and second conjunct on heap updates via suitable rewrite rules: modifications happen either at one of the two paths or below them (in the latter case no update is necessary at all). Most Separation Logic based provers (e.g. [20,26]) support separating conjunction only, which would require to define several versions of `abs` with additional paths and references as arguments, depending on which of them is contained in a subtree.

The proof exploits that each operation modifies at most one location inside the tree. This allows to split up the abstraction at this location, prove that the operation has the expected local behavior (e.g. that it rotates the referenced subtree correctly), and then merge the abstraction again with the updated subtree. For this, two fundamental theorems were formulated. The first theorem splits the abstraction of a tree rbt at a path p.

$$\begin{aligned} p \in rbt \rightarrow (\texttt{abs}(rootRef,\ pRef,\ rbt)(rbh) \leftrightarrow \\ \exists\ pthRef,\ pPthRef. \\ (\ \ \texttt{abspath}(rootRef,\ pRef,\ rbt,\ p,\ pthRef,\ pPthRef) \\ *\ \texttt{abs}(pthRef,\ pPthRef,\ rbt[p]))(rbh)) \end{aligned}$$

The heap predicate `abspath` is a weaker version of `abs`: the tree represented in rbh must match rbt except for the subtree starting at p. The references $pthRef$ and $pPthRef$ are used to fix the references of the root of the subtree and its parent. Thus, the split-off subtree $rbt[p]$ can be abstracted separately using `abs` with $pthRef$ as root and $pPthRef$ as parent reference. Conversely, reconnecting a detached subtree rbt_0 at path p with the original tree rbt is done via

$$\begin{aligned} &(\ \ \texttt{abspath}(rootRef,\ pRef,\ rbt,\ p,\ pthRef,\ pPthRef) \\ &\ *\ \texttt{abs}(pthRef,\ pPthRef,\ rbt_0))(rbh) \\ &\rightarrow \texttt{abs}(rootRef,\ pRef,\ rbt[p := rbt_0])(rbh) \end{aligned}$$

The theorem allows to attach an arbitrary tree rbt_0 (that is not necessarily related to the originally separated subtree $rbt[p]$). In practice, rbt_0 typically results from a simple modification of $rbt[p]$ like a recoloration or a rotation.

7 Related Work

Our concept with components (machines) and subcomponents is similar to the 'include' of B machines into others (see [1], chapter 7), although our individual operations (called events in B) are not assumed to be atomic (B disallows recursion, loops and sequential composition in operations).

Our approach is related but goes beyond the standard technique to use an abstraction relation (or function) that maps the pointer structure to a (free) algebraic structure, which is e.g. supported by Verifast [20] or was used e.g. in Automath [30]. We use such an abstraction function, but only to verify the core refinement from `RBTBASIC` to `RBTHEAP`.

We are aware of two alternatives to our approach. First, there is Cogent [28] which restricts programs by using a linear type system to propagate data

structures linearly. This allows to generate destructive code immediately but places severe limitations on the programming language.

Another alternative is to optimize code with a code generator that transforms algebraic types into pointer structures and tries to optimize non-destructive operations to destructive ones using checks for linear use. The approach developed in [25] follows this idea. It additionally generates proof obligations showing that the destructive implementation behaves the same way as the abstract one. It may be possible to encode our approach into this approach, as it is also based on data refinement, though the approach seems to be targeted towards individual algorithms, not (state-based) abstract data types.

Our code generator in KIV works similarly: it would transform trees into a tree-shaped pointer structure and use destructive updates on the pointer structure whenever a data flow analysis suggests this is possible (current work is to optimize this). However, we currently do not verify the correctness of these transformations; this is up to future work. The strategy is often sufficient to get efficient code. However, without the refinement of `RBTBASIC` to `RBTHEAP`, paths would be represented as doubly- linked lists, which would still be inefficient.

We are aware of several other works that verify red-black trees. Partial verifications, where the emphasis is on automation, are [7] (proving insertion without establishing that on every path to a leaf the number of black nodes must be equal) and [27] (just proving the ordering property). [2, 3, 16, 29] are complete verifications of algebraic implementations that produce functional (nondestructive) code. Our approach follows that idea but expands it to refine the functional to an imperative (destructive) implementation.

There are two complete verifications of destructive code we are aware of. One is described in the recent paper [4] using VerCors. The implementation is directly in Java, and the main routines use recursion and no parent pointers. But typical C implementations of red-black trees do without a recursion stack, so they are somewhat more efficient. Recursion simplifies the proofs considerably, as then a recursive call transforms a red-black subtree into another one and an invariant that combines `isRbtreeD` on our upper level with *abspath* from the lower level can be avoided. With VerCors being an automatic verifier backed by an SMT solver, proofs are guided by adding suitable annotations to the programs instead of directly interacting with a GUI during a proof as in KIV. Overall, the user input necessary for the final proof seems somewhat smaller in VerCors than in KIV, but we expect that finding why a proof fails to be significantly harder since SMT solvers do not give a reason why a goal is not provable, while KIV's proof trees allow inspecting residual goals that have been maximally simplified.

From the data given, the effort seems to have been somewhat higher than with our approach. The case study described also includes the verification of an extra concurrent operation that merges two red-black trees using lists as an intermediate representation. Verification of our case study took two regular students of computer science (who had done a KIV course) two months each under the mentoring of one of the authors.

The other complete verification of a pointer-implementation of red-black trees we are aware of is mentioned in [36] and can be found in Isabelle's AFP library. The version verified is derived from [22], which is a version of red-black trees intended to be used in a functional language with garbage collection, originally Haskell (the Scala-Library also uses it to implement *immutable* sets). This version of the algorithm uses recursion and no parent pointers. For verification, it has been modified to use destructive updates instead of copying the modified branch. The algorithm has a simpler, different rebalancing strategy than the original algorithm, making it less efficient than the original algorithm: when backing out of the recursion, a check for rebalancing is necessary on every level, resulting in logarithmic effort for fixing the tree. The original algorithm however has only one of four cases, where fixing a leaf after an insert (one in six cases for delete) has to traverse upwards to fix the parent. The resulting geometric series ($1 + 1/4 + 1/16 + \ldots = 4/3$ nodes are traversed upwards on average) results in *constant* average complexity for fixing the tree (see also [21], Chapter 7.4ff). That a full traversal upwards is unnecessary is the main reason, why red-black trees are more efficient than other versions of search trees like AVL trees, that need rebalancing checks on every level. We were unable to get a figure on the proof effort spent for the verification in [36], the automation using an automated saturation prover called auto2 implemented on top of Isabelle however is quite impressive. Apart from specific setup instructions for the prover, the proof seems to be fully automated requiring just a few lemmas.

Partial verification of destructive code can also be found in [12] (C code) and [11] (SPARKS, a subset of Ada). Both have analyzed insertion only and left away the `sameBlacks` property. The last approach is interesting since it uses an array-representation of red-black trees that would be suitable for real-time use (the array needs to be large enough to hold all tree nodes). It should not be too difficult to replace our heap-based representation in the lower refinement with their array-based one, exploiting that we do not have to re-verify any of the invariants of red-black trees to do this.

8 Conclusion

In this paper, we have demonstrated a refinement-based approach that allows to separate considerations of algorithmic correctness and of using destructive updates, aliasing and memory allocation into two individual refinements. The two core ideas are to abstract pointers as used by the algorithm to paths over an algebraic structure, and to use an interface that encapsulates primitive manipulations on abstract structures and paths. Verification is then split into functional correctness of the relevant algorithms on a purely algebraic level and a small part that shows that primitive operations can be correctly refined to pointer updates.

Our approach is not intended to compete with the best-automated techniques but to demonstrate that a clean separation of functional and pointer correctness is possible without compromising the final algorithm's efficiency. Our approach also should enable using one of the many techniques that automate proofs, e.g.

[31] for Separation Logic used in the lower refinement, or [15] for the algebraic trees of the upper refinement.

KIV's code generator generates C code from this implementation that is available together with all KIV specifications and proofs online [23].

The approach is successful in generating optimal code: it runs as fast (within a margin of $\pm 5\%$) of the code in stdc++ library when elements are inserted, looked up, or deleted randomly (the code also uses the rotations as given in [9] to fix red-black trees after inserting/deleting elements). For comparison, we also programmed the recursive red-black algorithms verified in [36] in KIV (without any modular structure or verification) and generated C code from them. The resulting code runs ca. 10% slower than our code. The KIV programs, resulting C code, and the benchmarks are included in the Web presentation [23] too.

There are two features we have not implemented compared to this code: one is to have an additional pointer to the minimal and maximal element. As a consequence accessing the maximum skips traversing the tree to the rightmost leaf, so inserting (or removing) elements in ascending order is still 10-20% faster. The algorithm in the library also has the option to cache deleted nodes to avoid deallocation and reallocation. We have not used this option in the comparison. Our current modularization, however, is certainly able to add these features.

We currently work on applying the approach to other data structures, namely (wandering) B+ Trees [19], which are the last not yet verified component of our flash filesystem [5], where we so far have achieved partial results [14] only.

Though we have defined an extension of the component concept to concurrency [34], it is future work to research how the approach given here could be extended to concurrent data structures.

Acknowledgement We would like to thank our students Nikolai Glaab and Felix Pribyl, who have done large parts of the verification of red-black trees.

References

1. J. R. Abrial, A. Hoare, and Pierre Chapron. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. R. Affeldt, J. Garrigue, X. Qi, and K. Tanaka. Proving Tree Algorithms for Succinct Data Structures. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:19, 2019.
3. A. Appel. Efficient Verified Red-Black Trees, 2011.
4. L. Armborst and M. Huisman. Permission-Based Verification of Red-Black Trees and Their Merging. In *Proc. of FormaliSE 21*, pages 111–123, 2021.
5. S. Bodenmüller, G. Schellhorn, M. Bitterlich, and W. Reif. Flashix: Modular Verification of a Concurrent and Crash-Safe Flash File System. In *Logic, Computation and Rigorous Methods: Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday*, volume 12750 of *LNCS*, pages 239–265. Springer, 2021.
6. E. Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 1 5(1–2):237–257, 2003.

7. A. Charguéraud. Program Verification through Characteristic Formulae. In *Proc. of ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 321–332. Association for Computing Machinery, 2010.
8. A. Charguéraud. Higher-Order Representation Predicates in Separation Logic. In *Proc. of ACM SIGPLAN Conference on Certified Programs and Proofs (CPP)*, pages 3–14. Association for Computing Machinery, 2016.
9. Cormen, T. and Leiserson, C. and Rivest, R. and Stein, C. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
10. J. Derrick and E. Boiten. *Refinement in Z and in Object-Z : Foundations and Advanced Applications*. FACIT. Springer, 2001. second, revised edition 2014.
11. C. Dross and Y. Moy. Auto-Active Proof of Red-Black Trees in SPARK. In *NASA Formal Methods*, pages 68–83. Springer, 2017.
12. J. Elgaard, A. Møller, and M. I. Schwartzbach. Compile-time debugging of C programs working on trees. In *Proc. 9th European Symposium on Programming (ESOP)*, volume 1782 of *LNCS*, pages 182–194. Springer, 2000.
13. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular, Crash-Safe Refinement for ASMs with Submachines. *Science of Computer Programming*, 131:3 – 21, 2016. Abstract State Machines, Alloy, B, TLA, VDM and Z (ABZ 2014).
14. G. Ernst, G. Schellhorn, and W. Reif. Verification of B^+ Trees: An Experiment Combining Shape Analysis and Interactive Theorem Proving. In *Proceedings of SEFM*, volume 7041 of *LNCS*, pages 253–268. Springer, 2011.
15. M. Faella and G. Parlato. Reasoning about Data Trees using CHCs. In *CAV*. Springer LNCS, 2022. (to appear, available from the author).
16. J.-C. Filliâtre and P. Letouzey. Functors for Proofs and Programs. In *13th European Symposium on Programming (ESOP)*, LNCS, pages 370–384. Springer, 2004.
17. L. J. Guibas and R. Sedgewick. A Dichromatic Framework for Balanced Trees. In *Proc. 19th Symp. on Foundations of Comp. Sci. (SFCS)*, pages 8–21. IEEE, 1978.
18. D. Harel, J. Tiuryn, and D. Kozen. *Dynamic Logic*. MIT Press, 2000.
19. F. Havasi. An Improved B+ Tree for Flash File Systems. In *SOFSEM*, volume 6543 of *LNCS*, pages 297–307. Springer, 2011.
20. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proc. NFM*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.
21. P. Sanders K. Mehlhorn. *Algorithms and Data Structures - The Basic Toolbox*. Springer, 2008.
22. S. Kahrs. Red-Black Trees with Types. *J. of Functional Programming 11(4)*, pages 182–196, July 2001.
23. KIV Proofs for the Correctness of Red-Black Trees, 2022. `https://kiv.isse.de/projects/RBtree.html`.
24. P. Lammich. Refinement to Imperative/HOL. In *6th International Conference on Interactive Theorem Proving (ITP 2015)*, volume 9236 of *LNCS*, pages 253–269. Springer, 2015.
25. P. Lammich. Efficient Verified Implementation of Introsort and Pdqsort. In *Automated Reasoning*, pages 307–323, Cham, 2020. Springer International Publishing.
26. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer, 2016.
27. T. Nipkow. Automatic Functional Correctness Proofs for Functional Search Trees. In *ITP*, volume 9807 of *LNCS*, pages 307–322. Springer, 2016.

28. L. O'Connor, Z. Chen, C. Rizkallah, V. Jackson, S. Amani, G. Klein, T. Murray, T. Sewell, and G. Keller. Cogent: Uniqueness Types and Certifying Compilation. *Journal of Functional Programming*, 31:25, 2021.
29. R. Peña. An Assertional Proof of Red–Black Trees Using Dafny. *Journal of Automated Reasoning*, 64, 04 2020.
30. N. Polikarpova, J. Tschannen, and C. A. Furia. A Fully Verified Container Library. In *FM*, volume 9109 of *LNCS*, pages 414–434. Springer, 2015.
31. A. Reynolds, R. Iosif, C. Serban, and T. King. A Decision Procedure for Separation Logic in SMT. In *ATVA*, volume 9938 of *LNCS*, pages 244–261, Cham, 2016. Springer.
32. J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.
33. G. Schellhorn, S. Bodenmüller, M. Bitterlich, and W. Reif. Software & System Verification with KIV. In *The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*, volume 13360 of *LNCS*, pages 408–436. Springer, 2022.
34. G. Schellhorn, S. Bodenmüller, J. Pfähler, and W. Reif. Adding Concurrency to a Sequential Refinement Tower. In *Rigorous State-Based Methods*, volume 12071 of *LNCS*, pages 6–23. Springer, 2020.
35. R. Sedgewick and K. Wayne. *Algorithms, 4th Edition.* Addison-Wesley, 2011.
36. B. Zhan. Efficient Verification of Imperative Programs Using Auto2. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 10805 of *LNCS*, pages 23–40. Springer, 2018.