Refinement and Separation: Modular Verification of Wandering Trees [⋆]

Gerhard Schellhorn, Stefan Bodenmüller, and Wolfgang Reif

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

Abstract. Flash memory does not allow in-place updates like conventional hard disks. Therefore all file systems must maintain an index that maps identifiers for files and directories to the address of their most recently written version. For efficiency, the index is typically implemented as a Wandering Search Tree. However, the verification of Wandering Trees is challenging since it has to deal with multiple aspects at once: the algorithmic complexity of search trees, trees in RAM that are partially loaded from snapshots on flash, where only modified parts are incrementally saved, and the efficient representation of trees as pointer structures. This paper proposes a modular solution that allows verifying each aspect separately. The solution has been mechanized in the theorem prover KIV.

Keywords: Wandering Trees, Refinement, Interactive Verification

1 Introduction

Flash memory has the constraint that it does not allow direct overwriting of data. It is organized in blocks that can only be written sequentially. Writing new data into a block is possible only after it has been erased as a whole. Therefore, file systems for flash memory (as well as flash translation layers used by SSDs, that mimic an ordinary hard disk that allows overwriting) have to manage an *index* that maps unique keys that identify elements of the file system to the physical address where their latest version can be found. The keys are usually based on *inode numbers* that uniquely identify files and directories, together with page numbers that identify data pages of a file.

An efficient implementation of the index is crucial for the efficiency of the file system. Of course, storing the index simply on flash memory itself is not an efficient solution since, again, incremental updates in place would not be possible. On the other hand, just keeping the index in RAM is not an option either since the index would be lost on a crash (e.g., a power loss). The standard

[⋆] Supported by the Deutsche Forschungsgemeinschaft (DFG), "Correct translation of abstract specifications to C-Code (VeriCode)" (grant RE828/26-1).

solution used nowadays in flash file systems is to use *Wandering Trees* [11] as an efficient solution. In Linux, for example, UBIFS [13] uses this solution.

We will explain Wandering Trees in Section 2. They present a challenge for verification as they combine algorithmic complexity, reasoning about pointer structures, and incremental caching.

The contribution of this paper is a mechanized verification of Wandering Trees using the theorem prover KIV, available online at [14]. The solution allows addressing these verification challenges in isolation using *components* without losing the efficiency of the overall solution. That pointer reasoning can be tackled in a small separate component is a generic aspect that should be reusable in other case studies.

We give some basic information on KIV's logic and specification concepts in Section 3 and discuss the concept of components and subcomponents, which are connected by refinement, in Section 4. The verification we present here is the last gap that was long open in the verification of Flashix, which is a fully verified file system for flash memory, see [1] for an overview.

Section 5 will give an overview of the modularization, and Section 6 gives the core concepts used in the verification. An interesting aspect of the verification is that we could make use of various operators of Separation Logic [19, 23]. In particular, we found Separation Logic and the use of the magic wand to be useful as a concept already when viewing the index abstractly as a map, generalizing its usual use for heaps (or other low-level resources). We also used the sharing separation operator to verify the correct representation of snapshots on flash memory.

Finally, Section 7 concludes.

2 Wandering Trees

Wandering Trees are used to implement an index, which maps keys that identify file system data to addresses on flash memory. Typically, they are organized as B^+-Trees. In this paper, we will use simple binary search trees since they are easier to explain, and rebalancing algorithms for trees are not the focus of this paper. Verification of B^+-Trees alone (without the aspects discussed here) has already been addressed in our paper [7], and we have also verified the rebalancing algorithms of Red-Black Trees in [24]. Several other papers have also discussed the verification of B^+-Trees in isolation, see [16,17]. Concurrent search trees have also been verified in [15].

How Wandering Trees work is shown in Figure 1. The system keeps a current version of the index stored as a search tree in RAM (top row), and an older snapshot is saved on Flash memory (bottom row). The two versions are called the *ram index* and the *flash index*. A new snapshot is saved when the *log* of the file system that sequentially saves changes (log entries record additions, modifications, and deletions) to the file system becomes full. Then the ram index is saved to the flash index in a *commit* operation, which also starts a new log

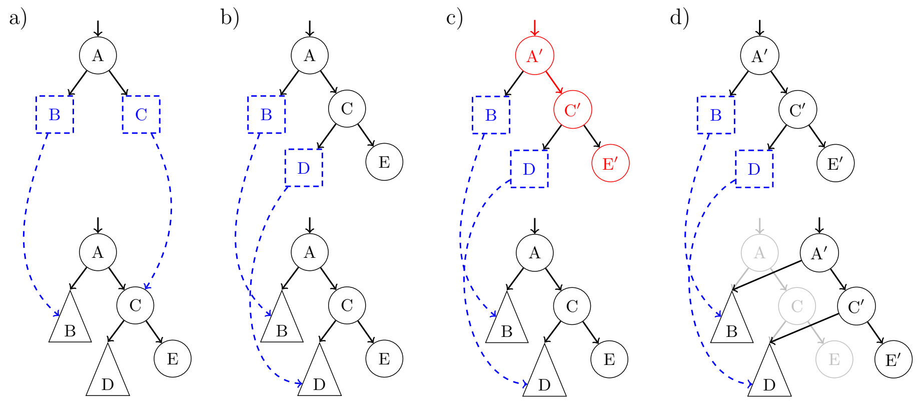

Fig. 1: Exemplary sequence of Wandering Tree operations with blue proxy nodes and red dirty nodes. RAM content shown on top, flash content below it.

(the old log becomes regular memory). A commit is also done when shutting down the file system, so on a reboot, a current flash index is available.

However, the flash index is not immediately loaded to the ram index, as this would be too expensive. Instead, just the root node A is loaded to RAM, as shown in Figure 1 a). Its child nodes B, C are proxy nodes (shown in blue): these just store the address on flash memory of the actual nodes B, C.

Reading data from flash memory will need some part of the index to find their current version. Since files in the same directory and pages of the same file have similar keys, this will typically require accessing some nodes that are close together in some subtree of the flash index. In Figure 1 b), nodes C and E were required to find some content of flash memory. Accessing the nodes will load their actual content. To speed up following reads and writes to these files, they are replaced in the ram index with their actual content. A new proxy node for D is created.

Updating files and directories will modify some nodes of the ram index. Figure 1 c) shows the changes caused by updating node E to point to a new version of some data. This will not just update the content of node E (to E') but it will also set a *dirty* flag in all nodes on the path from the root to E (here A, C, E). *Dirty nodes*, where the flag is set, are shown in red. Precisely the dirty nodes differ between ram index and flash index.

A commit operation, that saves the content of the flash index to the ram index will only save the dirty nodes, causing the flash index to wander (hence the name: Wandering Trees). The effect of a commit can be seen in Figure 1 d): only the dirty nodes A', C' and E' need to be saved on Flash memory, all nodes in the subtrees B and D still remain valid. In the ram index the dirty flag is cleared, since after the commit these nodes agree again with the flash index.

Wandering Trees are crash-safe since the ram index that, like all data in RAM, is lost on a power loss can be reconstructed: a reboot will find a non-empty log on flash memory. Starting with the flash index, a recovery routine will reconstruct the current ram index by replaying the entries of this log. Verification of crash safety is part of this case study, as is ensuring that hardware errors (all operations on flash can return errors) are handled correctly. We refer to [20] for the theory that results in additional proof obligations for each component, but this issue is out of scope for the remainder of the paper. We just note that for crash safety, it is essential that the commit operation has a final atomic step that both switches the ram index from Figure 1 c) to d) and the log to a new empty log. Technically this is realized by switching to a new superblock, see [5].

Verification of Wandering Trees as a monolithic implementation has to deal with three problems at the same time:

- the ram index has to correctly implement a search tree.
- the ram index implements a cache for the flash index. Replacing proxies with actual nodes, the mechanism of having dirty nodes, and committing these have to work correctly.
- Wandering Trees are pointer structures. It has to be verified that they correctly represent trees and that there are no space leaks.

Of course in theory, it is possible to consider all these aspects at once, but early experiments indicated that, at least in our theorem prover, the resulting complexity of invariants and abstraction relations becomes overwhelming.

Therefore, this paper's contribution is a modular structure, explained in the next section, that allows the decoupling of the three aspects and defines intuitive invariants and abstraction relations to make the individual verification tasks manageable.

3 Structured Specifications of Algebraic Data Types

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.

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 Definitions

The standard approach for proving the correctness of algorithms using complex data structures is to specify the data structures algebraically. Binary trees can be defined as a polymorphic free data type $\mathit{Bintree}(\kappa, \alpha)$ with a `Leaf` constructor (which maps one key to a value) and a `Node` constructor. Our case study will use type $\mathit{Bintree}(\mathit{Key}, \mathit{Elem})$ for search trees, where the Key type is assumed to be totally ordered. This allows to formulate a standard invariant $\texttt{isOrdered}(bt)$ for search trees bt: for any node in the tree, the keys stored in its right (left) subtree are bigger (less or equal) than the keys stored in the node.

$$\begin{aligned} \mathit{Bintree}(\kappa,\alpha) \;&= \texttt{Node}(\texttt{.key} : \kappa;\ \texttt{.left} : \mathit{Bintree}(\kappa,\alpha);\ \texttt{.right} : \mathit{Bintree}(\kappa,\alpha)) \\ &\mid \texttt{Leaf}(\texttt{.key} : \kappa;\ \texttt{.val} : \alpha) \end{aligned}$$

For a free data type specification, KIV generates all necessary axioms, as well as update functions (written e.g. $bt\texttt{.key:=}\ \mathit{newkey}$), including their definitions. KIV attaches a *domain* to selector functions. When used in programs, this ensures that selecting a value from a `Node` or the left subtree from a `Leaf` 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 modelling 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 that store objects of type ω are specified as a polymorphic non-free data type $\mathit{Heap}(\omega)$.

A heap can be considered a partial function (a "map") that maps a finite number of references to objects, where allocation of references is explicit and the reference type contains a distinguished element `null` that is never allocated (representing the null pointer). Since Separation Logic formulas can be defined (and will be useful for our case study) for maps in general, we define generic maps as a generic type $\mathit{Map}(\kappa,\nu)$ that maps arbitrary keys of type κ to arbitrary values of type ν. Heaps then map references of type $\mathit{Ref}(\omega)$ to objects of type ω, i.e., $\mathit{Heap}(\omega)$ abbreviates $\mathit{Map}(\mathit{Ref}(\omega),\omega)$.

The $\mathit{Map}(\kappa,\nu)$ data type is inductively generated by the empty map $\emptyset$, and an operator $m\texttt{[}k\ \texttt{:=}\ v\texttt{]}$ that stores the value v under key k, either allocating the key if it is new or overwriting the old value stored previously. A predicate $k \in m$ checks whether a key is allocated in the map, i.e., is in its domain, and a function $m\texttt{[}k\texttt{]}$ is used to lookup the value stored under a key (for heaps this corresponds to dereferencing a pointer). Keys and their value can also be removed (deallocated) by the function $m\ \texttt{--}\ k$. The union $m_1 \cup m_2$ of two maps is defined when they have disjoint domains (written $m_1 \perp m_2$). Similar to the selector functions of free data types, lookup $m\texttt{[}k\texttt{]}$ and removal $m\ \texttt{--}\ k$ raise exceptions in programs when the key is not in the map ($\neg\ k \in m$).

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) in KIV. We give some information to explain the notation used in the following. SL formulas are encoded using predicates over maps P with type $\mathit{Map}(\kappa,\nu) \to \mathit{Bool}$, abbreviated as MapPrd. A map predicate P describes the structure of a map m. Predicate `emp` describes an empty map:

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

The maplet $k \mapsto v$ describes a singleton map containing only one key k mapping to a value v. The infix operator $\mapsto$ therefore has type $\kappa \times \nu \to \mathit{MapPrd}$ and is defined as

$$(k \mapsto v)(m) \leftrightarrow m = \emptyset\texttt{[}k \texttt{ := } v\texttt{]}$$

More complex maps can be described using the separating conjunction $P \texttt{ * } Q$ asserting that the map consists of two disjoint parts, one satisfying P and one satisfying Q, respectively. Since it connects two heap predicates, it is defined as a function with type $\mathit{MapPrd} \times \mathit{MapPrd} \to \mathit{MapPrd}$:

$$(P \texttt{ * } Q)(m) \leftrightarrow \exists\ m_1, m_2.\ m_1 \perp m_2 \wedge m = m_1 \cup m_2 \wedge P(m_1) \wedge Q(m_2)$$

Separating conjunction is useful to assert that a map (or specifically: the heap) stores a tree-shaped structure where the left and right subtree are disjoint.

For our case study we will also make use of two more operators of the same type defined as

$$\begin{aligned}(P \texttt{ -* } Q)(m) &\leftrightarrow \forall\ m_1.\ m_1 \perp m \wedge P(m_1) \to Q(m \cup m_1)\\ (P \circledast Q)(m) &\leftrightarrow \exists\ m_1, m_0, m_2.\ m_1 \perp m_0 \wedge m_0 \perp m_2 \wedge m_1 \perp m_2\\ &\qquad \wedge m = m_1 \cup m_0 \cup m_2 \wedge P(m_1 \cup m_0) \wedge Q(m_0 \cup m_2)\end{aligned}$$

The *magic wand* $P \texttt{ -* } Q$ asserts that adding any disjoint map m_1 that satisfies P to the current map m will result in a map that satisfies Q.

Overlapping conjunction [9,12] $P \circledast Q$ asserts that the map m can be split into three disjoint parts m_0, m_1, m_2 such that P holds for the union of m_0 and m_1, while the union of m_0 and m_2 satisfies Q. This is useful, when the map stores a tree-shaped structure where the left and right subtree may have a shared part m_0 (so the structure becomes a directed acyclic graph).

Finally, some of the lemmas that we define will use implication $P \Rightarrow Q$ lifted to map predicates defined as $\forall\ m.\ P(m) \to Q(m)$.

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 [4,26]. 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 [2]: 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, which supports recursive procedures and nondeterminism (details on the syntax can be found in [25]), and establishes the postcondition of the operation. The arguments of a procedure $\mathbf{op}_j\#(in_j; st; out_j)$ are grouped into sequences of input, reference, and output parameters. KIV does not support global variables, these must be added explicitly as reference parameters. Instead of defining initial states directly, we also give a procedure declaration $\mathbf{init}\#(in_{init}; st; out_{init})$ $\{\alpha_{init}\}$.

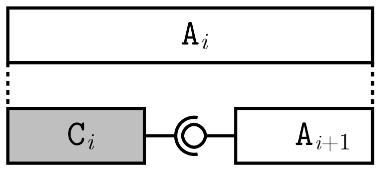

Fig. 2: Data refinement with subcomponents. Implementation $\mathtt{C}_i$ uses specification $\mathtt{A}_{i+1}$ as a subcomponent (depicted by —◎—). Together they are a refinement of specification $\mathtt{A}_i$ (depicted by the dotted lines).

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 post-conditions 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 only use constructs 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 $\mathtt{C} \leq \mathtt{A}$ 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 $\mathtt{C}$ and $\mathtt{A}$ have the same set of operations J). Proofs for such a refinement are usually done by first showing strong enough *invariants* for each component (that are preserved by all operations) and then proving an abstraction relation $\mathtt{abs} \subseteq ST^{\mathtt{A}} \times ST^{\mathtt{C}}$ to be a forward simulation. Proof obligations are based on a weakest precondition calculus for programs α that borrows notation from Dynamic Logic (DL) [10] and uses three modalities: $[\alpha]\varphi$ (corresponding to the weakest liberal precondition $wlp(\alpha, \varphi)$), $\langle\alpha\rangle\varphi$ (there is a terminating execution of α that establishes φ), and $\langle\!|\alpha|\!\rangle\, \varphi$ (corresponding to the weakest precondition $wp(\alpha, \varphi)$).

Proving a forward simulation correct requires to show the following proof obligation for each operation (i.e., $j \in J$) following the contract approach [4,26]:

$$\begin{aligned}&\mathtt{abs}(st^{\mathtt{A}}, st^{\mathtt{C}}) \land pre_j^{\mathtt{A}}(st^{\mathtt{A}}) \land inv^{\mathtt{A}}(st^{\mathtt{A}}) \land inv^{\mathtt{C}}(st^{\mathtt{C}})\\ &\to \langle\!|\mathbf{op}_j^{\mathtt{C}}\#(in_j; st^{\mathtt{C}}; out_j)|\!\rangle\, \langle\mathbf{op}_j^{\mathtt{A}}\#(in_j; st^{\mathtt{A}}; out_j')\rangle(\mathtt{abs}(st^{\mathtt{A}}, st^{\mathtt{C}}) \land out_j = out_j')\end{aligned}$$

The proof obligation assumes a pair of states $st^{\mathtt{A}}$ and $st^{\mathtt{C}}$ such that $\mathtt{abs}$ and the already established invariants hold and that the abstract precondition is true. It asserts that every concrete run terminates (in particular the concrete precondition must be satisfied) and has a corresponding abstract run with the same output, such that the abstraction relation holds for the resulting states (note that the postcondition refers to the final states of both programs). It is crucial that the calculus allows nested modalities (unlike Hoare calculus) to express the proof obligations. For information on symbolic execution and rewriting that are used for verification, see [25].

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 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. 2. A specification component $\mathtt{A}_i$ is refined by an implementation $\mathtt{C}_i$ (dotted lines in Fig. 2) that uses a specification $\mathtt{A}_{i+1}$ as a subcomponent (─◎─ in Fig. 2, we write $\mathtt{C}_i(\mathtt{A}_{i+1})$ for this subcomponent relation). Note that we visualize specification components as white boxes and implementation components as grey boxes throughout (cf. Fig. 2).

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 [6] we have shown that $\mathtt{C} \leq \mathtt{A}$ implies $\mathtt{M(C)} \leq \mathtt{M(A)}$ for a client component $\mathtt{M}$ 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 our Wandering Tree implementation.

5 Modularization

To specify and verify the different aspects of Wandering Trees separately, we employed the refinement hierarchy shown in Fig. 3. Starting from a concise and simple specification, multiple refinement steps are applied down to an efficient, pointer-based Wandering Tree implementation.

5.1 Specification of the Index

The top-level specification component `Index` completely abstracts from the tree data structure. Instead, the index is modeled as a map (i.e., a partial function) from index keys, which must be totally ordered, to values of an unspecified type *Elem*. The distinction between the cached and persisted version is realized by keeping two maps of type $Map(Key, Elem)$: the *ram index* (*ri*) representing the current state of the index and the *flash index* (*fi*) storing the latest *committed* state.

For the actual implementation, the *Elem* type is instantiated with a type *Adr* of flash addresses. However, as both index and data nodes are stored under these addresses in the file system, we keep the type abstract here to avoid confusion with addresses used for storing index data structures introduced with subsequent refinements.

Figure 4 lists the full `Index` specification. Initially, i.e., after formatting the file system, *ri* and *fi* are both empty (operation **init#**, lines 1-3). Modifications to the index are performed only on the ram index: **store#** adds a key/value pair to *ri* resp. updates the value stored under the respective key (lines 5-9),

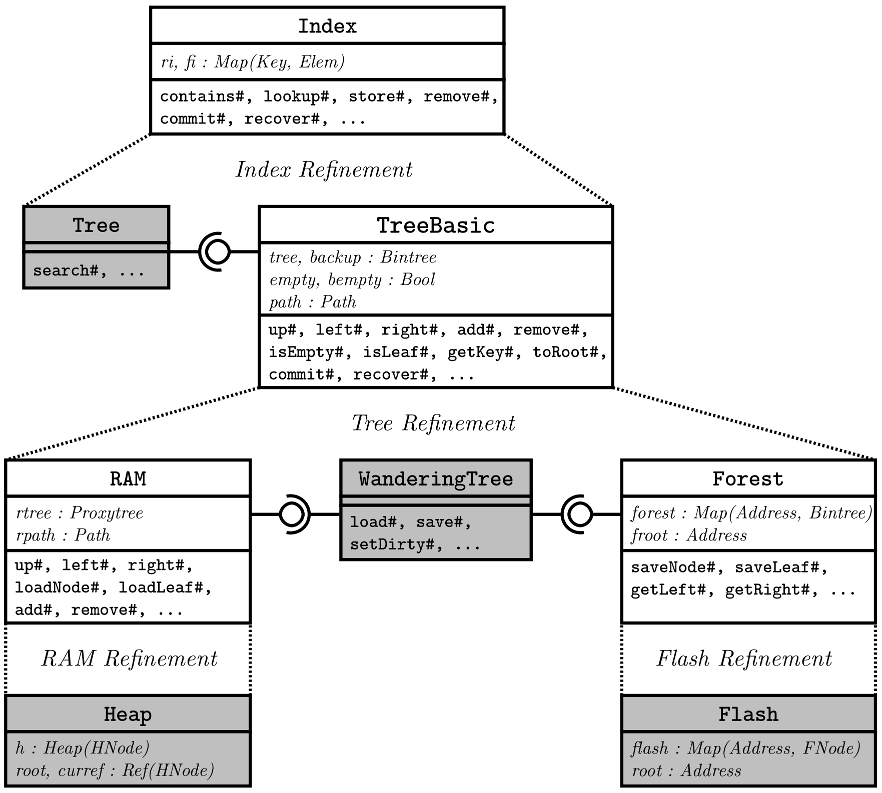
Index
ri, fi : Map(Key, Elem)
contains#, lookup#, store#, remove#, commit#, recover#, ...
Index Refinement
Tree
search#, ...
TreeBasic
tree, backup : Bintree
empty, bempty : Bool
path : Path
up#, left#, right#, add#, remove#, isEmpty#, isLeaf#, getKey#, toRoot#, commit#, recover#, ...
Tree Refinement
RAM
rtree : Proxytree
rpath : Path
up#, left#, right#, loadNode#, loadLeaf#, add#, remove#, ...
WanderingTree
load#, save#, setDirty#, ...
Forest
forest : Map(Address, Bintree)
froot : Address
saveNode#, saveLeaf#, getLeft#, getRight#, ...
RAM Refinement
Flash Refinement
Heap
h : Heap(HNode)
root, curref : Ref(HNode)
Flash
flash : Map(Address, FNode)
root : Address

Fig. 3: The refinement hierarchy for Wandering Trees. Each component lists its state (first compartment) and its most important operations (second compartment). Together, the four refinements *Index Refinement*, *Tree Refinement*, *RAM Refinement*, and *Flash Refinement* guarantee correctness of the combined Wandering Tree implementation, i.e., `Tree`(`WanderingTree`(`Heap`, `Flash`)) $\leq$ `Index`.

and **remove#** deletes an entry from the index (lines 11-15). Note that both operations return a flag *exists* signaling whether an entry for the requested *key* existed before the execution and, if so, the old value of the entry in *old* (otherwise, *old* is set to a random value `?`). **contains#** checks whether a *key* is allocated in *ri* (lines 16-17), and **lookup#** additionally returns the value stored under *key* (lines 19-23). The flash index is altered only during a **commit#** (lines 25-26), where it is updated to the current version stored in *ri*. Conversely, recovery after a crash is specified as restoring *ri* from *fi* (lines 28-29).

5.2 Index Refinement

In a first step, `Index` is refined by the component `Tree` using `TreeBasic` as a subcomponent (written `Tree`(`TreeBasic`) $\leq$ `Index`). This refinement addresses the realization of the index as a (binary) search tree. But instead of switching to a pointer representation directly, `TreeBasic` uses the *algebraic* trees

```
init#()
  initialization
{ $ri := \emptyset,\ fi := \emptyset$ }

store#($key, v;\ ;\ old, exists$) {
  $exists := key \in ri$;
  if $exists$ then $old := ri$[$key$] else $old :=$ ?;
  $ri$[$key$] $:= v$;
}

remove#($key;\ ;\ old, exists$) {
  $exists := key \in ri$;
  if $exists$ then { $old := ri$[$key$]; $ri$ -- $key$ }
  else $old :=$ ?;
}

contains#($key;\ ;\ exists$)
{ $exists := key \in ri$ }

lookup#($key;\ ;\ exists, v$) {
  $exists := key \in ri$;
  if $exists$ then $v := ri$[$key$]
  else $v :=$ ?
}

commit#()
{ $fi := ri$ }

recover#()
{ $ri := fi$ }
```

Fig. 4: Abstract representation of Wandering Trees: the component `Index`.

tree, *backup* : *Bintree*(*Key*, *Adr*) as state (cf. Sec. 3.1), representing the indices *ri* and *fi*, respectively. The state of `TreeBasic` also contains boolean flags *empty* and *bempty*, which are necessary to model that the respective tree is empty since *Bintree* only defines non-empty trees (consisting of at least a `Leaf`).

`TreeBasic` provides an interface for fine-grained manipulations of *tree* at a given location, like inserting a key/value pair in form of a `Leaf` with **add#** or reading the key of the current node with **getKey#**. These operations are used by the client component `Tree` to implement the interface of `Index`. For this, `TreeBasic` also has a state *path* : *Path* describing the way from the root of the tree to the current location, where *Path* is defined as a sequence of `LEFT` and `RIGHT` markers, i.e., *Path* $\equiv$ *List*(`LEFT` | `RIGHT`). To navigate within the tree, `Tree` calls the operations **up#**, **left#**, and **right#** to extend or shorten *path*.

The models use several algebraic operations for accessing and updating the trees: the predicate $p \in t$ checks whether a path p is valid for a tree t, i.e., p points to a subtree within t, the function $t\texttt{[}p\texttt{]}$ selects the subtree of t that is reached when traversing t along p (if $p \in t$), and the function $t\texttt{[}p \texttt{ := } t_0\texttt{]}$ yields the tree t where the subtree $t\texttt{[}p\texttt{]}$ is replaced with the tree t_0 (again, only defined for $p \in t$). These operations are defined recursively over the tree structure, for example, the following axioms are given for $t\texttt{[}p\texttt{]}$:[1]

$$t\texttt{[[]]} = t$$
$$\texttt{Node}(key, t_0, t_1)\texttt{[LEFT} + p\texttt{]} = t_0\texttt{[}p\texttt{]}$$
$$\texttt{Node}(key, t_0, t_1)\texttt{[RIGHT} + p\texttt{]} = t_1\texttt{[}p\texttt{]}$$

Using these axiomatized operations, the main part of the tree algorithms (and thus, their complexity) is placed in `Tree`, while the specification of `TreeBasic` is kept as simple as possible (most of the operations consist only of a single assignment). The verification of the `Tree` algorithms only needs to consider algebraic trees, abstracting from the more efficient but more complex tree representation used for the implementation of Wandering Trees.

[1] `[]` denotes the empty path, $a + p$ denotes a path consisting of one leading element a and a remaining path p.

```
1  search#(key; ; exists) {
2    let isEmpty = ? in {
3      tree_basic_isEmpty#(; ; isEmpty); // isEmpty := empty
4      if isEmpty then exists := false
5      else let isLeaf = ?, $key_0$ = ? in {
6        tree_basic_toRoot#(); // path := []
7        tree_basic_isLeaf#(; ; isLeaf); // isLeaf := tree[path].leaf?
8        while ¬ isLeaf do {
9          tree_basic_getKey#(; ; $key_0$); // $key_0$ := tree[path].key
10         if $key_0 < key$ then
11           tree_basic_right#(); // path := path + RIGHT
12         else
13           tree_basic_left#(); // path := path + LEFT;
14         tree_basic_isLeaf#(; ; isLeaf); // isLeaf := tree[path].leaf?
15       };
16       tree_basic_getKey#(; ; $key_0$); // $key_0$ := tree[path].key
17       exists := ($key = key_0$);
18     }
19   }
20 }
```

Fig. 5: `Tree` auxiliary procedure for performing a binary search for an element *elem* within the tree.

Fig. 5 shows how the interface of `TreeBasic` is used by `Tree` to implement a standard binary search. The auxiliary routine **search#** is used in most interface operations of `Tree` and searches for a *key*, returning a boolean flag whether a corresponding entry *exists* in the tree (set in lines 4 or 17, respectively). The comments in green show the implementation of the primitive operations of `TreeBasic`. For a non-empty tree (the state variable *empty* is set to true iff the tree does not contain any nodes), the tree is traversed by incrementally extending *path* with `LEFT` or `RIGHT` markers, depending on how the key key_0 of the current node compares to *key*, stopping when a leaf is reached (loop in lines 8-15). When **search#** has finished, *path* points to the location at which the actual modification (or lookup) of the respective `Tree` operation can be performed, e.g., adding/removing a node or reading the value.

At this level, **commit#** and **recovery#** are still modeled as simple atomic assignments of complete trees ($backup := tree$ and $tree := backup$, respectively). Furthermore, *tree* always describes the complete index tree, so optimizations like lazy loading or caching modifications of individual nodes are not yet considered. These aspects are introduced with the next refinement step.

5.3 Tree Refinement

The refinement $\texttt{WanderingTree}(\texttt{RAM}, \texttt{Forest}) \leq \texttt{TreeBasic}$ introduces the concepts of multiple tree snapshots stored on flash memory and partially loaded trees in volatile memory. However, both aspects are again modeled using algebraic trees instead of pointer structures.

The `Forest` component stores multiple binary trees in a map *forest* of type $ForestMap \equiv Map(Adr, Bintree(Key, Elem))$, where each entry represents a snapshot of the (full) tree created during a **commit#**. These snapshots are

```
load#() {
  let $isProxy$ = ? in {
    ram_isProxy#(; ; $isProxy$); // $isProxy$ := $rtree$[$rpath$].proxy?
    if $isProxy$ then let $adr$ = ?, $key$ = ?, $isLeaf$ = ? in {
      ram_getAdr#(; ; $adr$); // $adr$ := $rtree$[$rpath$].fadr
      forest_getKey#($adr$; ; $key$); // $key$ := $forest$[$adr$].key
      forest_isLeaf#($adr$; ; $isLeaf$); // $isLeaf$ := $forest$[$adr$].leaf?
      if $isLeaf$ then let $v$ = ? in {
        forest_getValue#($adr$; ; $v$); // $v$ := $forest$[$adr$].val
        ram_loadLeaf#($key$, $v$);
        // $rtree$[$rpath$] := Leaf($key$, $rtree$[$rpath$].fadr, $v$)
      } else let $ladr$ = ?, $radr$ = ? in {
        forest_getLeft#($adr$; ; $ladr$);
        // $ladr$ with $ladr \in forest \land forest$[$adr$].left = $forest$[$ladr$]
        forest_getRight#($adr$; ; $radr$);
        // $radr$ with $radr \in forest \land forest$[$adr$].right = $forest$[$radr$]
        ram_loadNode#($key$, $ladr$, $radr$);
        // $rtree$[$rpath$] := Node($key$, $rtree$[$rpath$].fadr, Proxy($ladr$), Proxy($radr$))
      }
    }
  }
}
```

Fig. 6: `WanderingTree` auxiliary procedure for instantiating a *proxy* node at the current path *rpath* by loading the node from flash.

stored at an abstract *Adr* on flash, and *froot* stores the address of the most recent snapshot, which is crucial for **recover#**. Note that since all trees in *forest* are algebraic, the snapshots are completely disjoint and do not share any subtrees (in contrast to the final implementation, cf. situation d) in Fig. 1). Thus, verification on this level does not have to cope with the additional complexity of sharing/aliasing.

The `RAM` component introduces *proxy nodes*, using a $Proxytree(Key, Elem)$ instead of a *Bintree* for the volatile version of the tree. The polymorphic data type $Proxytree(\kappa, \alpha)$ is specified analogously to $Bintree(\kappa, \alpha)$ but uses an additional constructor `Proxy` that represents not yet loaded subtrees and extends `Node` and `Leaf` nodes to also contain a field `.fadr` storing the address of the corresponding persisted tree on flash:

$$\begin{aligned} Proxytree(\kappa, \alpha) = {} & \texttt{Node}(\texttt{.key} : \kappa;\ \texttt{.fadr} : Adr;\ \texttt{.left} : Proxytree(\kappa, \alpha); \\ & \qquad \texttt{.right} : Proxytree(\kappa, \alpha)) \\ & \mid \texttt{Leaf}(\texttt{.key} : \kappa;\ \texttt{.fadr} : Adr;\ \texttt{.val} : \alpha) \mid \texttt{Proxy}(\texttt{.fadr} : Adr) \end{aligned}$$

For *dirty* (sub)trees, i.e., trees that contain uncommitted modifications, there is no corresponding flash address. These utilize the invalid `null` address to signal their *dirty* state. Note that, for the concepts presented in this paper, a simple boolean flag instead of the `.fadr` field would be sufficient to distinguish between *dirty* and *clean* trees. We store the addresses to identify flash locations that become *garbage* during a commit, which must be freed during a *garbage collection* algorithm (cf. [5]). However, this mechanism is implemented on another level of the file system, so it is outside the scope of this paper.

To implement the functionality of `TreeBasic`, `WanderingTree` accesses `RAM` and `Forest`. Recall that this functionality comprises only fine-grained accesses

```
1  setDirty#() {
2    let isRoot = ? in {
3      ram_isRoot#(; ; isRoot);
4      while ¬ isRoot do {
5        ram_up#();
6        ram_setAdr#(null);
7        ram_isRoot#(; ; isRoot);
8      }
9    }
10 }
```

Fig. 7: `WanderingTree` procedure for marking modified parts of the index dirty.

```
1  commit#() {
2    let isEmpty = ?,  adr = null in {
3      ram_toRoot#();
4      ram_isEmpty#(; ; isEmpty);
5      if ¬ isEmpty then {
6        save#(; ; adr);
7      };
8      forest_setRoot#(adr);
9    }
10 }
```

Fig. 8: `WanderingTree` procedure for committing the RAM index.

to the tree at a single location. Analogously to `TreeBasic`, this location is determined by a path *rpath* as part of the state of `RAM`. Since *rtree* is usually not fully loaded, the operations of `WanderingTree` first check whether *rpath* points to a `Proxy` tree in `RAM` and load the node from `Forest` if necessary. This is done with the auxiliary routine **load#** listed in Fig. 6. Again, the comments in green show the implementation of the primitive operations of the subcomponents `RAM` and `Forest`, respectively. If *rtree*`[`*rpath*`]` is a `Proxy` node, its flash address is used to determine whether the node is a `Leaf` and to read its *key* from flash (lines 3-7). For a leaf node, its value v is read from flash, and the proxy tree in *rtree* is replaced with a `Leaf` storing the loaded key/value pair (lines 9-11).[2] For a non-leaf, the proxy tree is replaced by a `Node`, which in turn stores only `Proxy` trees for its children (containing their loaded flash addresses, lines 13-18). That way, the `RAM` tree is loaded only as far as the respective `Tree` operation requires.

The other important responsibility of `WanderingTree` is to cache index updates in RAM until they are committed to flash. To determine the parts of the index that have to be written to flash during a commit, the path between an updated location and the root must be marked *dirty*. Therefore, the auxiliary operation **setDirty#** (shown in Fig. 7) is called after each modification: starting from the current location in `RAM` (determined by *rpath*, which points to the just modified subtree), *rtree* is traversed bottom-up until the root is reached (lines 4-8), setting all `.fadr` fields to `null` (line 6).

The **commit#** operation (shown in Fig. 8) calls an auxiliary operation **save#** (line 6), which saves a new tree in *forest* for each *dirty* node in *rtree*`[`*rpath*`]`, returning the new root address *adr* of the saved tree. Crucially, the root address *froot* of `Forest` is only updated at the very end of the operation (line 8), which ensures atomicity of the commit process w.r.t. crashes. In case of a crash during **commit#**, the RAM tree is initialized with the old *froot* address, under which the latest committed version of the index is still present.

The **save#** operation (listed in Fig. 9) has an interplay between `RAM` and `Forest` similar to **load#**: For a *dirty* `Leaf` in *rtree*, a corresponding `Leaf` is stored in *forest* (lines 8-10). For a *dirty* `Node` in *rtree*, the tree is traversed recursively (recursive calls in lines 13 and 16 for its left and right subtrees, respectively) before the corresponding `Node` is saved in *forest* with the flash

[2] The assignment *tree*`[`*path*`]` $:= t_0$ is an abbreviation for *tree* $:=$ *tree*`[`*path* `:=` t_0`]`.

```
 1  save#(; ; adr) {
 2    let isDirty = ? in {
 3      ram_isDirty#(; ; isDirty); // isDirty := rtree[rpath].dirty?
 4      if isDirty then let key = ?, isLeaf = ? in {
 5        ram_isLeaf#(; ; isLeaf); // isLeaf := rtree[rpath].leaf?
 6        ram_getKey#(; ; key); // key := rtree[rpath].key
 7        if isLeaf then let v = ? in {
 8          ram_getValue#(; ; v); // v := rtree[rpath].val
 9          forest_saveLeaf#(key, v; ; adr);
10          // adr with $\neg$ adr $\in$ forest $\wedge$ adr $\neq$ null in forest[adr] := Leaf(key, v)
11        } else let ladr = ?, radr = ? in {
12          ram_left#(); // rpath := rpath + LEFT
13          save#(; ; ladr);
14          ram_up#(); // rpath := rpath.butlast
15          ram_right#(); // rpath := rpath + RIGHT
16          save#(; ; radr);
17          ram_up#(); // rpath := rpath.butlast
18          forest_saveNode#(key, ladr, radr; ; adr);
19          // adr with $\neg$ adr $\in$ forest $\wedge$ adr $\neq$ null in
20          //    forest[adr] := Node(key, forest[ladr], forest[radr])
21        };
22        ram_setAdr#(adr); // rtree[rpath].fadr := adr
23      } else {
24        ram_getAdr#(; ; adr); // adr := rtree[rpath].fadr
25      }
26    }
27  }
```

Fig. 9: `WanderingTree` auxiliary procedure for recursively saving all *dirty* parts of *rtree* to flash, starting from the node at the current path *rpath*.

addresses *ladr* and *radr* of its stored children (lines 18-20). In both cases, the tree is stored under a new address *adr* in *forest*. Finally, the flash addresses of committed nodes in *rtree* are also updated and thus marked *clean* (line 22).

5.4 Pointer Structures in the Heap and on Flash

Finally, the algebraic trees are implemented by pointer structures: *RAM Refinement* realizes the proxy tree of `RAM` by a structure of *HNode*s in the heap, which is explicitly given as state $h : Heap(HNode)$ with references $Ref(HNode)$, and *Flash Refinement* realizes the binary trees of `Forest` as structures of *FNode*s stored on flash memory, which is modeled as a map $flash : Map(Adr, FNode)$. In [24], we used a similar modularization and refinement methodology to verify an efficient Red-Black Tree implementation: we also utilized an algebraic tree as an intermediate representation to abstract from the complexity of pointer-based rotations for maintaining balance. In this work, we revisit the methodology but apply it to simplify reasoning over the lazy loading and caching mechanisms at the level of `WanderingTree`(`RAM`, `Forest`). For the refinements `Heap` $\leq$ `RAM` and `Flash` $\leq$ `Forest`, the only (real) remaining proof task is to show that the pointer structures form correct trees (matching their algebraic counterparts) and that navigation via paths can be realized by dereferencing pointers.

Since [24] covers the general concept of refining algebraic trees with pointer trees in detail, we just give a brief overview of the used tree data structures.

$$\begin{aligned} \mathit{HNode} \;=\; & \texttt{HNode}(\texttt{.key} : \mathit{Key};\ \texttt{.fadr} : \mathit{Adr};\ \texttt{.parent} : \mathit{Ref}(\mathit{HNode});\\ & \qquad \texttt{.left} : \mathit{Ref}(\mathit{HNode});\ \texttt{.right} : \mathit{Ref}(\mathit{HNode}))\\ & \mid \texttt{HLeaf}(\texttt{.key} : \mathit{Key};\ \texttt{.fadr} : \mathit{Adr};\ \texttt{.parent} : \mathit{Ref}(\mathit{HNode});\ \texttt{.val} : \mathit{Elem})\\ & \mid \texttt{HProxy}(\texttt{.fadr} : \mathit{Adr};\ \texttt{.parent} : \mathit{Ref}(\mathit{HNode})) \end{aligned} \tag{1}$$

$$\begin{aligned} \mathit{FNode} \;=\; & \texttt{FNode}(\texttt{.key} : \mathit{Key};\ \texttt{.left} : \mathit{Adr};\ \texttt{.right} : \mathit{Adr})\\ & \mid \texttt{FLeaf}(\texttt{.key} : \mathit{Key};\ \texttt{.val} : \mathit{Elem}) \end{aligned} \tag{2}$$

Analogously to *Proxytree*, the *heap node* data type *HNode* is defined with constructors for inner nodes (`HNode`), leaf nodes (`HLeaf`), and proxy nodes (`HProxy`). The in-memory tree structure is formed via `.left`/`.right` pointers of `HNode`s. Additionally, each node contains a pointer to its parent node, which is necessary to navigate efficiently within the tree, e.g., for implementing **up#**. All heap nodes store the address of their persistent counterpart, i.e., the corresponding *FNode* on flash, in the `.fadr` field (or `null` if the node is *dirty*).

The *flash node* data type *FNode* defines a standard pointer representation of binary trees with constructors for inner (`FNode`) and leaf nodes (`FLeaf`), but flash addresses are used instead of heap references to the left and right children. Parent pointers are omitted since the tree is traversed top-down only.

Note that the representation as a map in `Flash` is not the final representation of data in the full implementation of the Flashix file system. There, an *Adr* storing an *FNode* will be allocated in blocks of flash memory. Thus, in the actual integration into the full Flashix file system, the `Flash` component will call memory allocation from a subcomponent again instead of just choosing a new *Adr* when allocating. The interested reader can find information on these lower level components, which do caching and a mapping from logical to physical blocks, in [20, 21].

6 Verification

This section gives a rough overview of the verification of the individual refinements. For brevity, we only give the most important invariants of the components and the definitions of the abstraction relations needed to verify the individual refinements. This should enable the reader to understand the main ideas used in the proofs, which are available online [14]. Since the concepts for the refinement `Heap` $\leq$ `RAM` are basically the same as in [24], we skip them here, although the refinement is crucial for having an efficient implementation that is based on efficiently modifying pointer structures.

6.1 Correctness of the Tree Component

Verification of the top-level refinement `Tree`[`TreeBasic`] $\leq$ `Index` first has to prove total correctness of **search#** with precondition $\mathit{empty} \lor \texttt{isOrdered}(\mathit{tree})$

and postcondition

$$\begin{aligned}&\neg\ \mathit{empty} \rightarrow (\mathit{path} \in \mathit{tree} \wedge \mathit{tree}[\mathit{path}]\texttt{.isLeaf}\\ &\quad \wedge (\texttt{neighbor}(\texttt{LEFT}, \mathit{path}) = \texttt{[]} \vee \mathit{tree}\texttt{[neighbor}(\texttt{LEFT}, \mathit{path})\texttt{.butlast].key} < \mathit{key})\\ &\quad \wedge (\texttt{neighbor}(\texttt{RIGHT}, \mathit{path}) = \texttt{[]} \vee \mathit{key} \leq \mathit{tree}\texttt{[neighbor}(\texttt{RIGHT}, \mathit{path})\texttt{.butlast].key}))\end{aligned}$$

which asserts that the path found indeed points to the right node. In the formula $\texttt{neighbor}(\texttt{LEFT}/\texttt{RIGHT}, p)$ computes the previous/next path of p in the tree. For the leftmost path (of the form $\texttt{LEFT}^*$), the previous path is empty, and for $p = p_0\,\texttt{RIGHT}\,\texttt{LEFT}^*$, it is $p_0\,\texttt{LEFT}$. $p\texttt{.butlast}$ removes the last element from this path. The keys of the previous and next path are the relevant ones to verify that subsequent modifications of the tree at *path* preserve `isOrdered`. The loop invariant used is similar to the postcondition.

The main proof of refinement has to verify that the two abstract maps $\mathit{fi}, \mathit{ri} : \mathit{Map}(\mathit{Key}, \mathit{Adr})$ are correctly represented by the two trees *tree* and *backup* and the two boolean flags *empty* and *bempty* that indicate that the empty map needs no tree for representation. The definition of the abstraction relation[3]

$$(\mathit{empty} \supset \texttt{emp}; \texttt{absI}(\mathit{tree}))(\mathit{ri}) \wedge (\mathit{bempty} \supset \texttt{emp}; \texttt{absI}(\mathit{backup}))(\mathit{fi}))$$

uses predicate `absI` : $\mathit{Bintree}(\mathit{Key}, \mathit{Adr}) \rightarrow \mathit{MapPrd}$ defined as

$$\texttt{absI}(\texttt{Leaf}(\mathit{key}, v)) = \mathit{key} \mapsto v \qquad \texttt{absI}(\texttt{Node}(\mathit{key}, \mathit{lt}, \mathit{rt})) = \texttt{absI}(\mathit{lt}) \texttt{ * } \texttt{absI}(\mathit{rt})$$

Note that separating conjunction is used here for the maps of the abstract specification; the common case is to use it to represent an abstract structure by a pointer structure as in the refinement $\texttt{Heap} \leq \texttt{RAM}$. The refinement proofs use the magic wand to prove correctness of adding and removing an element. The main lemma needed is that for a valid path $p \in \mathit{bt}$ we have

$$\texttt{absI}(\mathit{bt}) \Rightarrow \texttt{absI}(\mathit{bt}\texttt{[}p\texttt{]}) \texttt{ * } (\texttt{absI}(\mathit{rt}) \texttt{ -* } \texttt{absI}(\mathit{bt}\texttt{[}p \texttt{ := } \mathit{rt}\texttt{]}))$$

which says that the tree bt consists of two parts: the old subtree $\mathit{bt}\texttt{[}p\texttt{]}$ that is replaced in the operation by a new subtree rt, resulting in the new tree $\mathit{bt}\texttt{[}p \texttt{ := } \mathit{rt}\texttt{]}$. For removal of a *key*, when $\mathit{bt}\texttt{[}p\texttt{]} = \texttt{Node}(\mathit{key}', \texttt{Leaf}(\mathit{key}, v), \mathit{rt})$ we then have

$$\begin{aligned}&\texttt{absI}(\texttt{Node}(\mathit{key}', \texttt{Leaf}(\mathit{key}, v), \mathit{rt})) \texttt{ * } (\texttt{absI}(\mathit{rt}) \texttt{ -* } \texttt{absI}(\mathit{bt}\texttt{[}p \texttt{ := } \mathit{rt}\texttt{]}))\\ =\ &\texttt{absI}(\texttt{Leaf}(\mathit{key}, v)) \texttt{ * } \texttt{absI}(\mathit{rt}) \texttt{ * } (\texttt{absI}(\mathit{rt}) \texttt{ -* } \texttt{absI}(\mathit{bt}\texttt{[}p \texttt{ := } \mathit{rt}\texttt{]}))\\ =\ &(\mathit{key} \mapsto v) \texttt{ * } \texttt{absI}(\mathit{rt}) \texttt{ * } (\texttt{absI}(\mathit{rt}) \texttt{ -* } \texttt{absI}(\mathit{bt}\texttt{[}p \texttt{ := } \mathit{rt}\texttt{]}))\\ \Rightarrow\ &(\mathit{key} \mapsto v) \texttt{ * } \texttt{absI}(\mathit{bt}\texttt{[}p \texttt{ := } \mathit{rt}\texttt{]}))\end{aligned}$$

where the last step applies the standard modus ponens rule for magic wand. Removing the maplet $\mathit{key} \mapsto v$ from the map (with $m \texttt{ -- } \mathit{key}$), we get that $\texttt{absI}(\mathit{bt}\texttt{[}p \texttt{ := } \mathit{rt}\texttt{]})(m)$ holds, i.e., the new map is correctly represented by the updated tree. The proof for adding a key/value pair is similar.

6.2 Correctness of Wandering Trees

The refinement $\texttt{WanderingTree}(\texttt{RAM}, \texttt{Forest}) \leq \texttt{TreeBasic}$ is the most complex of the case study since it has the most complex algorithms. However, since it is purely based on recursive definitions over algebraic structures, the proofs of this

[3] $(\varphi \supset e1; e2)$ abbreviates **if** φ **then** $e1$ **else** $e2$

refinement were best to automate. As a first step of the proof, all operations of the `Forest` component have to maintain the critical invariant `invForest` for the map *forest* that stores the flash index on this level. The invariant states that, whenever some $\texttt{Node}(lt, key, rt)$ is stored as a value in the map, its two children *lt* and *rt* are also stored under some address, i.e., the map is closed against subtrees. This guarantees that the operations **getLeft#** and **getRight#**, which return the addresses of the children (see lines 13 and 15 in Figure 6), are always successful. The invariant is preserved since the only operation modifying *forest* is **save#**, which is called with a node which has its children saved already.

The main correctness argument for the refinement then uses a function $\texttt{merge} : \mathit{Proxytree}(\mathit{Key}, \mathit{Adr}) \times \mathit{ForestMap} \to \mathit{Bintree}(\mathit{Key}, \mathit{Adr})$ defined as

$$\begin{aligned}
&\texttt{merge}(\texttt{Leaf}(k, v, adr), \mathit{forest}) = \texttt{Leaf}(k, v)\\
&\texttt{merge}(\texttt{Node}(k, lt, rt, adr), \mathit{forest}) = \texttt{Node}(k, \texttt{merge}(lt, \mathit{forest}), \texttt{merge}(rt, \mathit{forest}))\\
&\texttt{merge}(\texttt{Proxy}(adr), \mathit{forest}) = \mathit{forest}\texttt{[}adr\texttt{]}
\end{aligned}$$

which replaces all proxies with the content stored in *forest*. This function is used in the main invariant $\texttt{invMerge}(\mathit{rtree}, \mathit{forest})$ of component `WanderingTree` which asserts that, for all clean nodes of *rtree* that store a non-`null` address *adr*, the subtree below the node agrees with the tree stored at $\mathit{forest}\texttt{[}adr\texttt{]}$. The invariant is again defined recursively as

$$\begin{aligned}
&\texttt{invMerge}(t, \mathit{forest}) \leftrightarrow (t\texttt{.fadr} \neq \texttt{null} \to \mathit{forest}\texttt{[}t\texttt{.fadr]} = \texttt{merge}(t, \mathit{forest}))\\
&\quad \wedge (t\texttt{.node?} \to \texttt{invMerge}(t\texttt{.left}, \mathit{forest}) \wedge \texttt{invMerge}(t\texttt{.right}, \mathit{forest}))
\end{aligned}$$

It is preserved since new nodes are allocated dirty with $t\texttt{.fadr} = \texttt{null}$, so `invMerge` does not make an assertion for those. Again, only **save#** commits (the top-level node of a) tree t for which $t\texttt{.fadr} \neq \texttt{null}$, but for such a node the definition of `merge` directly computes $\mathit{forest}\texttt{[}t\texttt{.fadr]}$.

The `merge` function is also used to define the abstraction relation

$$\begin{aligned}
&\mathit{rpath} = \mathit{path} \wedge (\mathit{empty} \leftrightarrow \mathit{rtree} = \texttt{Proxy}(\texttt{null}))\\
&\wedge (\neg\ \mathit{empty} \to \mathit{tree} = \texttt{merge}(\mathit{rtree}, \mathit{forest}))\\
&\wedge (\mathit{bempty} \leftrightarrow \mathit{froot} = \texttt{null}) \wedge (\neg\ \mathit{bempty} \to \mathit{backup} = \mathit{forest}\texttt{[}\mathit{froot}\texttt{]})
\end{aligned}$$

which states that the abstract *tree* can be reconstructed by replacing proxies in *rtree*, and that the current flash index *backup* is stored under *froot*.

As one part of the refinement proof, the **load#** program has to be shown to be correct, which modifies the ram index *rtree* by replacing proxy nodes with real nodes (the transition from a) to b) in Figure 1). Essentially this proof reduces to showing that the result of $\texttt{merge}(\mathit{rtree}, \mathit{forest})$ is unchanged by the operation. Similar lemmas have to be proved for the **setDirty#** from Figure 7 and the **save#** program from Figure 9, which do the main work in the transitions from b) to c) to d) in Figure 1.

6.3 Correctness of Flash Representation

The refinement $\texttt{Flash} \leq \texttt{Forest}$ has to show that the abstract representation of flash memory as a $\mathit{forest} : \mathit{ForestMap}$, which stores whole trees under addresses, can be replaced with a node-based representation as $\mathit{flash} : \mathit{Map}(\mathit{Adr}, \mathit{FNode})$

(see (2) for the definition of type *FNode*). Since the various snapshots of the flash tree now share parts (as can be seen in Figure 1 d)), we found it convenient to define the abstraction relation absF in terms of the sharing separation operator from Section 3.2.

$$\begin{aligned}&\texttt{absF}(\texttt{Leaf}(key, v), adr) = adr \mapsto \texttt{FLeaf}(key, v) * \texttt{true}\\&\texttt{absF}(\texttt{Node}(key, lt, rt), adr) =\\&\quad \exists\ ladr, radr.\ (adr \mapsto \texttt{FNode}(key, ladr, radr)) \circledast \texttt{absF}(lt, ladr) \circledast \texttt{absF}(rt, radr)\end{aligned}$$

Note that the base case has a "* true" clause that allows the representation to contain extra old data, e.g., the old node A in Figure 1 d). The full abstraction relation states that the root is unchanged, both maps have the same domain (selected with dom) of addresses, and that any tree in *forest* can be reconstructed from the pointer representation:

$$\begin{aligned}&\quad froot = root \land \texttt{dom}(flash) = \texttt{dom}(forest)\\&\land \forall\ adr.\ adr \in \texttt{dom}(forest) \to \texttt{absF}(forest\texttt{[}adr\texttt{]}, adr)(flash);\end{aligned}$$

Since nodes are always added but never deleted on flash memory, using overlapping conjunction made the proof of this refinement really simple.

7 Conclusion

In this paper we have presented a modular decomposition of the concept of Wandering Trees into 4 components, each consisting of an abstract specification and an implementation. The decomposition allows the verification of the three main verification problems in isolation: correctness of search trees, caching with proxies and dirty nodes, representation as pointer structures.

In particular, most of the problems could be addressed on the abstract level of algebraic trees, keeping verification effort manageable. Overall the effort for the case study was ca. one month to work out the concepts and ca. three months for verification. Our approach could be combined with one of the many techniques to further automate individual component proofs, e.g., [3, 8, 18, 22, 27].

Given that we could again (as in [24]) split away a small component that reasons about pointers from the main part, which addresses other correctness issues, increases our confidence that the component decomposition we employ should be usable in many other case studies that use pointer structures to represent abstract data types without having to compromise efficiency.

The final implementation, which composes all implementations (the grey parts of Figure 3), is purely imperative and does not use any functional data structures like trees but pointer structures only. After inlining calls to subcomponents (as done by our code generator for Scala and C), the code is almost identical to and as efficient as a monolithically programmed version.

Acknowledgement We would like to thank our student Felix Pribyl who has done a lot of the work presented in this paper as part of his Master thesis.

References

1. 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.
2. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, 2003.
3. 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.
4. J. Derrick and E. Boiten. *Refinement in Z and in Object-Z : Foundations and Advanced Applications.* FACIT. Springer, 2001. second, revised edition 2014.
5. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Inside a Verified Flash File System: Transactions & Garbage Collection. In *Proc. of Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 9593 of *LNCS*, pages 73–93. Springer, 2015.
6. 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).
7. 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.
8. M. Faella and G. Parlato. Reasoning about Data Trees using CHCs. In *CAV*, volume 13372 of *LNCS*, pages 249—271. Springer, 2022.
9. P. A. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for javascript. *SIGPLAN Not.*, 47(1):31–44, 2012.
10. D. Harel, J. Tiuryn, and D. Kozen. *Dynamic Logic.* MIT Press, 2000.
11. F. Havasi. An improved b+ tree for flash file systems. In *SOFSEM 2011: Theory and Practice of Computer Science*, volume 6543 of *LNCS*, pages 297–307, 2011.
12. A. Hobor and J. Villard. The ramifications of sharing in data structures. In *Proc. POPL*, pages 523—-536, New York, NY, USA, 2013. Association for Computing Machinery.
13. A. Hunter. A brief introduction to the design of UBIFS. 2008. URL: `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`.
14. KIV Proofs for the Correctness of Wandering Trees, 2023. `https://kiv.isse.de/projects/WanderingTrees.html`.
15. S. Krishna, N. Patel, D. Shasha, and T. Wies. Verifying concurrent search structure templates. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI 2020, page 181–196. ACM, 2020.
16. J.G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *Proceedings POPL 2010*, pages 237—-248. ACM, 2010.
17. Niels N. Mündler and T. Nipkow. A Verified Implementation of B^+-Trees in Isabelle/HOL. In *Theoretical Aspects of Computing – ICTAC 2022*, volume 13572 of *LNCS*, pages 324–341. Springer, 2022.
18. T. Nipkow. Automatic Functional Correctness Proofs for Functional Search Trees. In *ITP*, volume 9807 of *LNCS*, pages 307–322. Springer, 2016.
19. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic, 15th International Workshop*, pages 1–19, 08 2001.

20. J. Pfähler, G. Ernst, S. Bodenmüller, G. Schellhorn, and W. Reif. Modular Verification of Order-Preserving Write-Back Caches. In *IFM: 13th International Conference, 2017, Proceedings*, volume 10510 of *LNCS*, pages 375–390. Springer, 2017.
21. J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. Formal Specification of an Erase Block Management Layer for Flash Memory. In *Proc. of Haifa Verification Conference (HVC)*, volume 8244 of *LNCS*, pages 214–229. Springer, 2013.
22. 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.
23. 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.
24. G. Schellhorn, S. Bodenmüller, M. Bitterlich, and W. Reif. Separating Separation Logic — Modular Verification of Red-Black Trees. In *Verified Software. Theories, Tools and Experiments.*, volume 13800 of *LNCS*, pages 129—-147. Springer, 2022.
25. 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.
26. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement.* Prentice Hall International Series in Computer Science, 1996.
27. 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.