A Verified Implementation of B^{+}-trees in Isabelle/HOL

Niels Mündler
Department of Computer Science, ETH Zurich, Switzerland

Tobias Nipkow
Department of Informatics, Technical University of Munich, Germany

Abstract

In this paper we present the verification of an imperative implementation of the ubiquitous B^{+}-tree data structure in the interactive theorem prover Isabelle/HOL. The implementation supports membership test, insertion and range queries with efficient binary search for intra-node navigation. The imperative implementation is verified in two steps: an abstract set interface is refined to an executable but inefficient purely functional implementation which is further refined to the efficient imperative implementation.

2012 ACM Subject Classification Theory of computation → Separation logic

Keywords and phrases Separation Logic, Verification, Refinement

Digital Object Identifier 10.4230/LIPIcs.IWS.2022.2

Acknowledgements I want to thank Peter Lammich for giving me a very comprehensive introduction separation logic and discussing the many ways to express assertions. I further want to thank Manuel Eberl for his immediate and impressive support with Isabelle proof tactics.

1 Introduction

B^{+}-trees form the basis of virtually all modern relational database management systems (RDBMS) and file systems. Even single-threaded databases are non-trivial to analyse and verify, especially machine-checked. Meanwhile it is important to verify various properties like functional correctness, termination and runtime, since RDBMS are ubiquitous and employed in critical contexts, like the banking sector and realtime systems. The only work in the literature on that topic that we are aware of is the work by Malecha *et al.* [10]. However, it lacks the commonly used range query operation, which returns a pointer to the lower bound of a given value in the tree and allows to iterate over all successive values. This operation is particulary challenging to verify as it requires to mix two usually strictly separated abstractions of the tree in order to reason about its correctness. We further generalize the implementation of node internal navigation. This allows to abstract away from its implementation and simplifies proofs. It further allows us to supply an implementation of efficient binary search, a practical and widespread runtime improvement as nodes usually have a size of several kilobytes. We provide a computer assisted proof in the interactive theorem prover Isabelle/HOL [13] for the functional correctness of an imperative implementation of the B^{+}-tree data-structure and present how we dealt with the resulting technical verification challenges.

2 Contributions

In this work, we specify the B^{+}-tree data structure in the functional modeling language higher-order logic (HOL). The tree is proven to refine a finite set of linearly ordered elements. All proofs are machine-checked in the Isabelle/HOL framework. Within the framework, the

Isabelle Workshop 2022.
Editors: Tobias Nipkow, Larry Paulson and Makarius Wenzel
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

functional specification already yields automatic extraction of executable, but inefficient code.

The contributions of this work are as follows

- The first verification of genuine range queries, which require additional insight in refinement over iterating over the whole tree.
- The first efficient intra-node navigation based on binary rather than linear search.

The remainder of the paper is structured as follows. In Section 1, we present a brief overview on related work and introduce the definition of B$^+$-tree used in our approach. In Sections 4 and 5, we refine a functionally correct, abstract specification of point, insertion and range queries as well as iterators down to efficient imperative code. Finally, we present learned lessons and evaluate the results in Section 6.

The complete source code of the implementation referenced in this research is accessible via GitHub[1].

2.1 Related Work

There exist two pen and paper proofs via a rigorous formal approach. Fielding [5] uses gradual refinement of abstract implementations. Sexton and Thielecke [16] show how to use separation logic in the verification. These are more of a conceptual guideline on approaching a fully machine checked proof.

There are two machine checked proofs of imperative implementations. In the work of Ernst *et al.* [4], an imperative implementation is directly verified by combining interactive theorem proving in KIV [14] with shape analysis using TVLA [15]. The implementation lacks shared pointers between leaves. This simplifies the proofs about tree invariants. However, the tree therefore also lacks iterators over the leaves, and the authors present no straightforward solution to implement them. Moreover, by directly verifying an imperative version only, it is likely that small changes in the implementation will break larger parts of the proof.

Another direct proof on an imperative implementation was conducted by Malecha *et al.* [10], with the Ynot extension to the interactive theorem prover Coq. Both works use recursively defined shape predicates that describe formally how the nodes and pointers represent an abstract tree of finite height. The result is both a fairly abstract specification of a B$^+$-tree, that leaves some design decisions to the impertive implementation, and an imperative implementation that supports iterators.

Due to the success of this approach, we follow their example and define these predicates functionally. One example of the benefits of this approach is that we were able to derive finiteness and acyclicity only from the relation between imperative and functional specification. In contrast to previous work, the functional predicates describing the tree shape are kept completely separated from the imperative implementation, yielding more freedom for design choices within the imperative refinement. Both existing works rely on linear search for intra-node navigation, which we improve upon by providing binary search. We extend the extraction of an iterator by implementing an additional range query operation.

3 B$^+$-trees and Approach

The B$^+$-tree is a ubiquitous data structure to efficiently retrieve and manipulate indexed data stored on storage devices with slow memory access [3]. They are k-ary balanced search

[1] `https://github.com/nielstron/bplustrees`

trees, where k is a free parameter. We specify them as implementing a set interface, where all elements in the leaves comprise the content of an abstract set. The inner nodes only contain separators instead of the set content. These separators have the same type as the set content, but are only used to guide the recursive navigation through the tree by bounding all set values in the neighboring subtrees. Further the leaves usually contain pointers to the next leaf, allowing for efficient iterators and range queries. A more formal and detailed outline of B$^+$-trees can be found in Section 3.2.

The goal of this work is to define this data structure and implement and verify efficient heap-based imperative operations on them. For this purpose, we introduce a functional, algebraic definition and specify all invariants on this level that can naturally be expressed in the algebraic domain. It is important to note that this representation is not complete, as aliased pointers are left out on the algebraic level. However, important structural invariants, such as sortedness and balancedness can be verified.

In a second step an imperative definition is introduced, that takes care of the refinement of lists to arrays in the heap and introduces (potentially shared) pointers instead of algebraic structures. Using a refinement relationship, we can prove that an imperative refinement of the functional specification preserves the structural invariants of the imperative tree on the heap. The only remaining proof obligation on this level is to ensure the correct linking between leaf pointers.

The above outlined steps are performed via manual refinement in Imperative HOL [2]. We build on the library of verified imperative utilities provided by the Separation Logic Framework [9] and the verification of B-Trees [11], namely list interfaces and partially filled arrays. The implementation is defined with respect to an abstract imperative operation for node-internal navigation. This means that within each node, we do not specify how the correct subtree for recursive queries is found, but only constrain some characteristics of the result. We provide one such operation that employs linear search, and one that conducts binary search. All imperative programs are shown to refine the functional specifications using the separation logic utilities from the Isabelle Refinement Framework by Lammich [8].

3.1 Notation

Isabelle/HOL conforms to everyday mathematical notation for the most part. For the benefit of the reader who is unfamiliar with Isabelle/HOL, we establish notation and in particular some essential datatypes together with their primitive operations that are specific to Isabelle/HOL. We write t *::* *'a* to specify that the term t has the type *'a* and *'a* ⇒ *'b* for the type of a total function from *'a* to *'b*. The type for natural numbers is *nat*. Sets with elements of type *'a* have the type *'a set*. Analogously, we use *'a list* to describe lists, which are constructed as the empty list *[]* or with the infix constructor *#*, and are appended with the infix operator *@*. The function *concat* concatenates a list of lists. The function *set* converts a list into a set. For optional values, Isabelle/HOL offers the type *option* where a term *opt* *::* *'a option* is either *None* or *Some a* with *a* *::* *'a*.

3.2 Definitions

We first define an algebraic version of B$^+$-trees. Proofs about the correctness of operations and the preservation of invariants are only done on the abstract level, where they are much simpler and many implementation details can be disregarded. It will serve as a reference point for the efficient imperative implementation.

The algebraic B$^+$-tree is defined as follows:

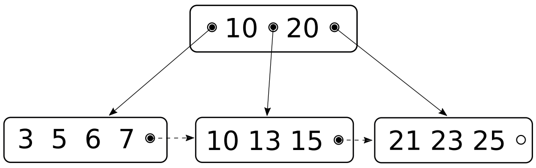

Figure 1 Nodes contain several elements, the internal list/array structure is not depicted. The dotted lines represent links to following leaf nodes that are not present in the algebraic formulation.

```
fun inbetween where
  inbetween f l [] t u = f l t u |
  inbetween f l ((sub,sep)#xs) t u = (f l sub sep ∧ inbetween f sep xs t u)

fun aligned where
  aligned l (Leaf ks) u = (l < u ∧ (∀x ∈ set ks. l < x ∧ x ≤ u)) |
  aligned l (Node ts t) u = (inbetween aligned l ts t u)

fun Laligned where
  Laligned (LNode ks) u = (∀x ∈ set ks. x ≤ u) |
  Laligned (Node ts t) u = (case ts of
      [] ⇒ Laligned t u |
      (sub,sep)#ts' ⇒ (Laligned sub sep) ∧ inbetween aligned sep ts' t u
    )
```

Figure 2 Definition of the alignment property.

```
datatype 'a bplustree =
    Leaf ('a list) |
    Node (('a bplustree × 'a ) list) ('a bplustree)
```

Every node *Node* $[(t_1,a_1), \ldots, (t_n,a_n)]$ t_{n+1} contains an interleaved list of *keys* a_i and *subtrees* t_i. We write as t_i the subtree to the left of a_i and t_{i+1} the subtree to the right of a_i. We refer to t_{n+1} as the *last* subtree. The leaves *Leaf* $[v_1, \ldots, v_n]$ contain a list of *values* v_i. Separators are only used for navigation within the tree. The concatenation of lists of values of a tree t yields all elements contained in the tree. We refer to this list as *leaves t*. A B$^+$-tree with above structure must fulfill the invariants *balancedness*, *order* and *alignment*.

Balancedness requires that each path from the root to a leaf has the same length. In other words, the height of all trees in one level of the tree must be equal, where the height is the maximum path length to a leaf.

The *order* property ensures a minimum and maximum number of subtrees for each node. A B$^+$-tree is of order k, if each internal node has at least $k+1$ subtrees and at most $2k+1$. The root is required to have a minimum of 2 and a maximum of $2k+1$ subtrees. We require that k be strictly positive, as for $k = 0$ the requirements on the tree root are contradictory.

Alignment means that keys are sorted with respect to separators: For a separator k and all keys l in the subtree to the left, $l < k$, and all keys r in the subtree to the right, $k \leq r$. (where $\leq$ and $<$ can be exchanged). Specifically we require for a tree t that *Laligned* t $\top$, where *Laligned* is defined as in Figure 2 and $\top$ is the top element of the linear order.

For the values within the leaves, *sortedness* is required explicitly. We require the even stronger fact that *leaves* t is sorted. This is a useful statement when arguing about the correctness of set operations.

All of these mentioned invariants are proven to be maintained by the abstract set operations. While these abstract operations already yields executable code, they are not translated into particularly efficient code.

The efficient implementation of B$^+$-trees is defined on the imperative level. Each imperative node contains pointers (*ref*) rather than the full subtree. We refine lists with partially filled arrays of capacity $2k$. A partially filled array (a, i) with capacity c is an array a of fixed size c. Only the first i elements are considered content of the array. Unlike dynamic arrays, partially filled arrays are not expected to grow or shrink. This way, the data structures are refined to an imperative level, each imperative node contains the equivalent information to an abstract node. The only addition is that leaves now also contain a pointer to another leaf, which will form a linked list over all leaves in the tree.

```
datatype 'a btnode =
  Btnode (('a btnode ref option × 'a) pfarray) ('a btnode ref) |
  Btleaf ('a pfarray) ('a btnode ref option)
```

It is possible to modify elements on the heap and share pointers with this setup. In order to use the algebraic data structure as a reference point, we introduce a refinement relation. The correctness of operations on the imperative node can then be shown by relating imperative input and output and to the abstract input and output of a correct abstract operation. In particular we want to show that if we assume $R\ t\ t_i$, where R is the refinement relation and t and t_i are the abstract and the imperative version of the "same" tree, $R\ o(t)\ o_i(t_i)$ should hold, where o_i is the imperative refinement of operation o. The relation is expressed as a separation logic formula that links an abstract tree to its imperative equivalent.

The notation for separation logic in Isabelle is quickly summarized in the list below.

- *emp* holds for the empty heap
- *true* and *false* hold for every and no heap respectively
- $\uparrow(P)$ holds if the heap is empty and predicate P holds
- $a \mapsto_r x$ holds if the heap at location a is reserved and contains value x
- $\exists_A x.\ P\ x$ holds if there exists some x such that Px holds on the heap.
- $P_1 * P_2$ denotes the separating conjunction and holds if each assertion P_1 and P_2 hold on non-overlapping parts of the heap
- *is_pfa* c xs xsi expresses that xsi is a partially filled array with capacity c that refines the list xs.
- *list_assn* P xs ys expresses that P *xs[i]* *ys[i]* holds for all $i \leq |xs| = |ys|$.

Separation Logic formulae always express the state of some heap. The assertion P describes all heaps for which the formula P evaluates to true. The entailment $P \Longrightarrow_A Q$ holds iff Q holds in every heap in which P holds. $P = Q$ holds iff $P \Longrightarrow_A Q \wedge Q \Longrightarrow P$. The formulas are usually used in the context of Hoare triples. We write $< P > c < \lambda r.\ Q\ r >$ if, for any heap where P holds, after executing imperative code c that returns value r, formula $Q\ r$ holds on the resulting heap. $< P > c < \lambda r.\ Q\ r >_t$ is a shorthand for $< P > c < \lambda r.\ Q\ r * true >$ More details can be found in the work of Lammich and Meis [9].

The assertion *bplustree_assn* expressing the refinement relation relates an algebraic tree (*bplustree*) and an imperative tree (*btnode ref*), as well as the first and last leaf of the imperative tree. The formal relation is shown in Figure 3.

The main structural relationship between abstract and imperative tree is established by linking abstract list and array via the *is_pfa* predicate, and recursively linking the abstract

```
fun bplustree_assn :: nat ⇒ 'a bplustree ⇒ 'a btnode ref ⇒ 'a btnode ref ⇒ 'a btnode ref
  where
 bplustree_assn k (LNode xs) a r z =
 ∃_A xsi fwd.
    a ↦_r Btleaf xsi fwd
  ∗ is_pfa (2∗k) xs xsi
  ∗ ↑(fwd = z)
  ∗ ↑(r = Some a)
  |
 bplustree_assn k (Node ts t) a r z =
 ∃_A tsi ti tsi' rs.
    a ↦_r Btnode tsi ti
  ∗ is_pfa (2∗k) tsi' tsi
  ∗ ↑(length tsi' = length rs)
  ∗ list_assn ((λ t (ti,r',z'). bplustree_assn k t (the ti) r' z') ×_a id_assn) ts (
      zip (zip (map fst tsi') (zip (butlast (r#rs)) rs))) (map snd tsi')))
  ∗ bplustree_assn k t ti (last (r#rs)) z)
```

Figure 3 The B$^+$-tree is specified by the split factor k, an abstract tree, a pointer to its root, a pointer to its first leaf and a pointer to the first leaf of the next sibling. The pointers to first leaf and next first leaf are used to establish the linked leaves invariant.

subtrees and subtree pointers inside the *list_assn*.

In addition to the refinement relation, the first and last leaf r and z are used to express the structural invariant that the leaves are correctly linked. This property is required for the iterator on the tree in Section 5.1. The structural invariant is ensured by passing the first leaf of the right neighbor to each subtree. We can not explicitly retreive these leaves from the tree structure. The reason is that any functions that follow the pointers of the tree are not guaranteed to terminate without the context of the structural soundness of the tree, which is only established within the refinement relation. Instead, we assume that there exists a list of such leaf pointers *rs*. We ensure that this list is the correct one by passing the supposedly first leaves into each subtree. The pointer is passed recursively to the leaf node, where it is compared to the actual pointer of the leaf. All of this happens in the convoluted *list_assn*, by folding over the list of the leaf pointer list *rs* zipped with itself, offset by one.

There is no abstract equivalent for the next pointers in the leaves, therefore we can only introduce and reason about this invariant on the imperative layer. Due to the constraints of separation logic, we cannot express this invariant in a separate statement from the refinement relation. We need to access the elements in each node to ensure the refinement relation, and in this step we also access the memory that contains the next pointers. Since separation logic only permits us to access the memory location exclusively in each term separated by the separating conjunction, this single access must cover all invariants.

3.3 Node internal navigation

In order to define meaningful operations that navigate the node structure of the B$^+$-tree, we need to find a method that handles search within a node. Ernst *et al.* [4] and Malecha *et al.* [10] both use a linear search through the key and value lists. However, B$^+$-trees are supposed to have memory page sized nodes [3], which makes a linear search impractical.

```
locale split_tree =
    fixes split :: ('a bplustree × 'a) list ⇒ 'a ⇒ (('a bplustree × 'a) list
    split xs p = (ls,rs) ⟹ xs = ls @ rs
    split xs p = (ls@[(sub,sep)],rs); sorted_less (separators xs) ⟹ sep < p
    split xs p = (ls,(sub,sep)#rs); sorted_less (separators xs) ⟹ p ≤ sep
```

Figure 4 Given a list of separator-subtree pairs and a search value x, the function should return the pair (s, t) such that, according to the structural invariant of the B$^+$-tree, t must contain x or will hold x after a correct insertion.

We introduce a context (*locale* in Isabelle) in which we assume that we have access to a function that correctly navigates through the node internal structure. We call this function *split*, and define it only by its behavior. Given a list of separator-subtree pairs and a search value x, the function should return the pair (s, t) such that, according to the structural invariant of the B$^+$-tree, t must contain x or will hold x after a correct insertion. A corresponding function *split_list* is defined on the separator-only lists in the leaf nodes. The formal specification for *split* is given in Figure 4.

In the following sections, all operations are defined and verified based on *split* and *split_list*. Finally, when approaching imperative code extraction, we provide a binary search based function, that refines *split*. This binary search is directly implemented and verified on the imperative level and is eventually plugged into the abstractly defined imperative operations on the B$^+$-tree. Thus we obtain imperative code that makes use of an efficient binary search, without adding complexity to the proofs. The definition and implementation closely follows the approach described in detail in the verification of B-Trees [11].

4 Set operations

B$^+$-trees refine sets on linearly ordered elements. For a tree t, the refined abstract set is computed as *set (leaves t)*. The set interface requires that there should be query, insertion and deletion operations o_t such that *set (leaves* $(o_t$ *t))* $=$ *o (set (leaves t)*. Moreover, the invariants described in Section 3 can be assumed to hold for t and are required for o_t. We provide these operations and show their correctness on the functional layer first, then refine the operations further to the imperative layer. For point queries and insertion, we follow the implementation suggested by Bayer and McCreight [1].

4.1 Functional Point Query

For an inner node t and a searched value x, find the correct subtree s_t such that if a leaf of t contains x, a leaf of s_t must contain x. Then recurse on s_t. Inside the leaf node, we search directly in the list of values. Note that we assume here that a *split* and *isin_list* operation exist, as described in Section 3.3.

```
fun isin:: 'a bplustree ⇒ 'a ⇒ bool where
  isin (LNode ks) x = (isin_list x ks) |
  isin (Node ts t) x = (case split ts x of
     (_,(sub,sep)#rs) ⇒ isin sub x
   | (_,[]) ⇒ isin t x
  )
```

```
partial_function (heap) isin :: 'a btnode ref ⇒ 'a ⇒  bool Heap where
  isin p x = do {
  node ← !p;
  (case node of
     Btleaf xs _ ⇒ imp_isin_list x xs |
     Btnode ts t ⇒ do {
       i ← imp_split ts x;
       tsl ← length ts;
       if i < tsl then do {
         s ← get ts i;
         let (sub,sep) = s in
           isin (the sub) x
       } else
           isin t x
     }
)}
```

Figure 5 The imperative definition of the *isin* function.

Since this function does not modify the tree involved at all, we only need to show that it returns the correct value.

```
theorem assumes sorted_less (leaves t) and aligned l t u
  shows isin t x = (x ∈ set (leaves t))
```

In general, these proofs on the abstract level are based on yet another refinement relation suggested by Nipkow [12]. We say that the B$^+$-tree t refines a sorted list of its leaf values, *leaves t*. We argue that recursing into a specific subtree is equivalent to splitting this list at the correct position and searching in the correct sublist. The same approach was applicable for proving the correctnes of functional operations on B-Trees [11].

The proofs on the functional level can therefore be made concise. We go on and define an imperative version of the operation that refines each step of the abstract operation to equivalent operations on the imperative tree.

4.2 Imperative Point Query

The imperative version of the point query is a partial function. Termination cannot be guaranteed anymore, at least without further assumptions. This is inevitable since the function would not terminate given cyclic trees. However, we will show that if the input refines an abstract tree, the function terminates and is correct. The imperative *isin* refines each step of the abstract operation with an imperative equivalent. The result can be seen in Figure 5.

Again, we assume that *imp_split* performs the correct node internal search and refines an abstract *split*. Note how *imp_split* does not actually split the internal array, but rather returns the index of the pair that would have been returned by the abstract split function. The pattern matching against an empty list is replaced by comparing the index to the length of the list l. In case the last subtree should be recursed into, the whole list l is returned.

In order to show that the function returns the correct result, we show that it performs the same operation on the imperative tree as on the algebraic tree. This is expressed in

Hoare triple notation and separation logic.

```
lemma assumes k > 0 and root_order k t and sorted_less (inorder t)
  and sorted_less (leaves t) shows
  <bplustree_assn k t ti r z>
    isin ti x
  <λy. bplustree_assn k t ti r z ∗ ↑(isin t x = y)>t
```

The proof follows inductively on the structure of the abstract tree. Assuming structural soundness of the abstract tree refined by the pointer passed in, the returned value is equivalent to the return value of the abstract function. We must explicitly show that the tree on the heap still refines the same abstract tree after the operation, which was implicit on the abstract layer. It follows directly, since no operation in the imperative function modifies part of the tree.

4.3 Insertion and Deletion

The insertion operation and its proof of correctness largely line up with the one for point queries. But since insertion modifies the tree, we need to additionally show on the abstract level that the modified tree maintains the invariants of B$^+$-trees.

On the imperative layer, we show that the heap state after the operation refines the tree after the abstract insertion operation. It follows that the imperative operation also maintains the abstract invariants. Moreover, we need to show that the linked list among the leaf pointers is correctly maintained throughout the operation. This can only be shown on the imperative level as there is no abstract equivalent to the shared pointers.

```
lemma assumes k > 0 and sorted_less (inorder t)
   and sorted_less (leaves t) and root_order k t shows
   <bplustree_assn k t ti r z>
   imp_insert k x ti
   <λu. bplustree_assn k (insert k x t) u r z>t
```

We provide a verified functional definition of deletion and a definition of an imperative refinement. Showing the correctness of the imperative version would largely follow the same pattern as the proof of the correctness of insertion. The focus of this work is not on basic tree operations however, but on obtaining an iterator view on the tree.

5 Range operations

This section introduces both how the general iterator on the tree leaves is obtained and the technical challenges involved (Section 5.1) as well as how to obtain an iterator on a specific subset of elements efficiently (Section 5.2).

On the functional level, the forwarding leaf pointers in each leaf are not present, as this would require aliasing. Therefore, the abstract equivalent of an iterator is a concatenation of all leaf contents. When refining the operations, we will make use of the leaf pointers to obtain an efficient implementation.

5.1 Iterators

The implementation of the leaf iterator is straightforward. We recurse down the tree to obtain the first leaf. From there we follow leaf pointers along the fringe of the tree until we

```
fun leaf_nodes_assn where
  leaf_nodes_assn k ((LNode xs)#lns) (Some r) z =
 (∃A xsi fwd.
      r ↦r Btleaf xsi fwd
    ∗ is_pfa (2∗k) xs xsi
    ∗ leaf_nodes_assn k lns fwd z
 ) |
  leaf_nodes_assn k [] r z = ↑(r = z) |
  leaf_nodes_assn _ _ _ _ = false
```

Figure 6 The refinement relation for leaf nodes comprises the refinement of the node content as well as the recursive property of linking correctly to the next node.

reach the final leaf marked by a null next pointer. However, from an assertion perspective the situation is more intricate. It is important to find an explicit formulation of the linked list view on the leaf pointers. Meanwhile, we want to maintain enough information about the remainder of the tree to be able to state that the complete tree does not change by iterating through the leaves. We cannot express an assertion about the linked list along the leaves and the assertion on the whole tree in two independent predicates, as separation logic forces us to not make statements about the contents of any memory location twice. This is an important feature of separation logic, in order to keep the parts of the heap disjoint and thus be able to locally reason about the heap state.

For this, we follow the approach of Malecha *et al.* [10] and try to find an equivalent formulation that separates the whole tree in a view on its inner nodes and the linked leaf node list. The central idea to separate the tree is to express that the linked leaf nodes refine *leaf_nodes t* and that the inner nodes refine *trunk t*, as depicted in Figure 7. These are two independent parts of the heap and therefore the statements can be separated using the separating conjunction.

Formally, we define an assertion *trunk_assn* and *leaf_nodes_assn.* The former is the same as *bplustree_assn* (see Figure 3), except that we remove all assertions about the content of the tree in the *LNode* case. The latter is defined similar to a linked list refining a list of abstract tree leaf nodes, shown in Figure 6. The list is refined by a pointer to the head of the list, which refines the head of the abstract list. Moreover, the imperative leaf contains a pointer to the next element in the list.

With these definitions, we can show that the heap describing the imperative tree may be split up into its leaves and the trunk.

lemma *bplustree_assn k t ti r z* $\Longrightarrow_A$ *leaf_nodes_assn k* (*leaf_nodes t*) *r z* ∗ *trunk_assn k t ti r z*

However, we cannot show that a structurally consistent, unchanged B$^+$-tree is still described by the combination of the two predicates. The reason is that we cannot express that the linked leaf nodes are precisely the leaf nodes on the lowest level of the trunk, depicted in red in Figure 7.

The root of this problem is actually a feature of the refinement approach. When stating that a part of the heap refines some abstract data structure, we make no or little statements about concrete memory locations or pointers. This is useful, as it reduces the size of the specification and the proof obligations. In this case it gets in our way.

We cannot express that the fringe of the trunk refines the same abstract leaves that are refined by the leaf list, as this would violate the disjointness of heaps. Even if we did, this

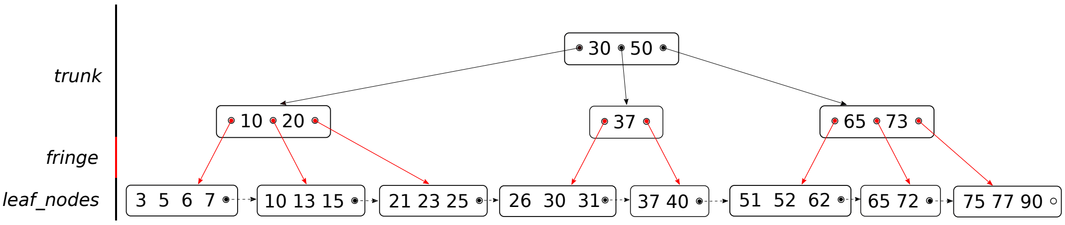

Figure 7 In order to obtain separate assertions about the concatenated leaf list (*leaf_nodes*) and the internal nodes (*trunk*) of the tree, the structure is abstractly split along the pointers marked in red, the *fringe*. In order to be able to combine the *leaf_nodes* and the *trunk* together, the *fringe* has to be extracted and shared explicitly.

statement would not be strong enough to guarantee that the actual memory locations are the same. We need to specifically express that these pointers, and not the abstract structure they refine, are precisely the same in the two statements.

In a second attempt we succeed by making the sharing explicit. We extract from the whole tree the precise list of pointers to leaf nodes, the *fringe* in the correct order. The fringe is then part of the assertion about the tree. Recursively, the fringe of a tree is the concatenation of all fringes in its subtrees. The resulting assertion can be seen in Figure 8. As a convenient fact, this assertion is equivalent to Figure 3.

lemma *bplustree_extract_fringe:*
$bplustree\_assn\ k\ t\ ti\ r\ z = (\exists_A fringe.\ bplustree\_assn\_fringe\ k\ t\ ti\ r\ z\ fringe)$

Using the *fringe*, we can precisely state an equivalent separated assertion. We describe the trunk with the assertion *trunk_assn*, which is the same as *bplustree_assn_fringe*, except that the *LNode* case is changed to only $\uparrow (r = Some\ a \wedge fringe = [a])$. In addition, we extend the definition of *leaf_nodes_assn* to take the *fringe* pointers into account. We now require that the *fringe* of the trunk is precisely the list of pointers in the linked list refining *leaf_nodes*.

lemma *bplustree_view_split:*
$bplustree\_assn\_fringe\ k\ t\ ti\ r\ z\ fringe =$
$leaf\_nodes\_assn\ k\ (leaf\_nodes\ t)\ r\ z\ fringe * trunk\_assn\ k\ t\ ti\ r\ z\ fringe$

To obtain an iterator on the leaf nodes of the tree, we obtain the first leaf of the tree. By the formulation of the tree assertion, we can express the obtained result using the assertion about the complete tree.

lemma assumes $k > 0$ **and** *root_order k t* **shows**
$<bplustree\_assn\ k\ t\ ti\ r\ z>$
first_leaf ti
$<\lambda u.\ bplustree\_assn\ k\ t\ ti\ r\ z * \uparrow(u = r)>_t$

On the result, we can apply lemmas *bplustree_extract_fringe* and *bplustree_view_split*. The transformed expression states that the result of *first_leaf t* is a pointer to *leaf_nodes t*. The tree root *t* remains to refine *trunk t*.

From here, we could define an iterator over the leaf nodes along the fringe, refining the abstract list *leaf_nodes*. However our final goal is to iterate over the values within each array inside the nodes. We introduce a flattening iterator for this purpose. It takes an outer iterator over a data structure *a* that returns elements of type *b*, and inner iterator over

```
fun bplustree_assn_fringe where
    bplustree_assn_fringe k (LNode xs) a r z fringe =
    ∃A xsi fwd.
        a ↦r Btleaf xsi fwd
      ∗ is_pfa (2∗k) xs xsi
      ∗ ↑(fwd = z)
      ∗ ↑(r = Some a)
      ∗ ↑(fringe = [a])
    |
    bplustree_assn_fringe k (Node ts t) a r z fringe =
    ∃A tsi ti tsi′ tsi′′ rs split.
        a ↦r Btnode tsi ti
      ∗ bplustree_assn_fringe k t ti (last (r#rs)) (last (rs@[z])) (last split)
      ∗ is_pfa (2∗k) tsi′ tsi
      ∗ ↑(concat split = fringe)
      ∗ ↑(length tsi′ = length rs)
      ∗ ↑(length split = length rs + 1)
      ∗ list_assn (
          (λ t (ti,r′,z′,fring). bplustree_assn_fringe k t (the ti) r′ z′ fring)
            ×a id_assn
        ) ts (zip
            (zip (map fst tsi′) (zip (butlast (r#rs)) (zip rs (butlast split))))
            (map snd tsi′)
        )
```

Figure 8 An extended version of the B$^+$-tree assertion from Figure 3. In order to be able to correctly relate leaf view and internal nodes, the shared pointers *fringe* are made explicit, without accessing their memory location.

the data structure b. It returns an iterator over the concatenated list of elements. In this case the inner structure would be the partially filled array stored in each leaf. Therefore we need an outer iterator not over the leaves, but over the arrays contained within. The exact implementation of this iterator is left out as a technical detail, and we can find an equivalent formulation of the leaf list and the list of arrays, which we call *leaves_assn*.

We define an iterator on this list assertion, fulfilling the list iterator interface defined by Lammich [7]. The iterator stores the pointer to the next element to be returned from the list. The iterator interface requires some functionality.

- An *init* function that returns the pointer to the head of the list.
- A *has_next* function that checks whether the current pointer is the null pointer.
- A *next* function that returns the the array in the current node and its next pointer.
- Proofs that we can transform the *leaves_assn* statement into a leaf iterator statement and vice versa.

We provide all of it and show that the linked leaf nodes of the B$^+$-tree form a valid list of arrays that can be iterated over. We combine this iterator with the iterator over partially filled arrays in the flattening iterator and obtain an iterator over all leaf values *leaf_values_iter*.

Finally, we want be able to express that the whole tree does not change throughout the iteration. For this, we need to keep track of both the leaf nodes assertion and the trunk assertion on t. The assertion describing the iterator therefore contains both. It also existentially quantifies the fringe, hiding away the fact that it was extracted in the first place from the client perspective. Note how all notion of the explicitly shared leaf pointers has disappeared on this level, as their existence was hidden within the definition of the tree iterator.

definition $\mathit{bplustree\_iter}\ k\ t\ \mathit{ti}\ r\ \mathit{vs}\ \mathit{it} = \exists_A\ \mathit{fringe}.$
  $\mathit{leaf\_values\_iter}\ \mathit{fringe}\ k\ (\mathit{leaf\_nodes}\ t)\ (\mathit{leaves}\ t)\ r\ \mathit{vs}\ \mathit{it}\ *$
  $\mathit{trunk\_assn}\ k\ t\ \mathit{ti}\ r\ \mathit{None}\ \mathit{fringe}$

The initializer using the *first_leaf* operation defined before now allows us to obtain an iterator over all leaf values of the tree. Using the iterator functionalities defined by the flattening operator, the values can be obtained step by step. The operations *bplustree_iter_next* and *bplustree_iter_has_next* are exactly the respective operations defined for *leaf_values_iter*, renamed.

lemma assumes $k > 0$ **and** $\mathit{root\_order}\ k\ t$ **shows**
  $<\mathit{bplustree\_assn}\ k\ t\ \mathit{ti}\ r\ \mathit{None}>$
  $\mathit{bplustree\_iter\_init}\ \mathit{ti}$
  $<\lambda \mathit{it}.\ \mathit{bplustree\_iter}\ k\ t\ \mathit{ti}\ r\ (\mathit{leaves}\ t)\ \mathit{it}>_t$

lemma assumes $\mathit{vs} \neq []$ **shows**
  $<\mathit{bplustree\_iter}\ k\ t\ \mathit{ti}\ r\ \mathit{vs}\ \mathit{it}>$
  $\mathit{bplustree\_iter\_next}\ \mathit{it}$
  $<\lambda(a, \mathit{it}').\ \mathit{bplustree\_iter}\ k\ t\ \mathit{ti}\ r\ (\mathit{tl}\ \mathit{vs})\ \mathit{it}'\ * \uparrow (a = \mathit{hd}\ \mathit{vs})>_t$

lemma
  $<\mathit{tree\_iter}\ k\ t\ \mathit{ti}\ r\ \mathit{vs}\ \mathit{it}>$
  $\mathit{bplustree\_iter\_has\_next}\ \mathit{it}$
  $<\lambda r'.\ \mathit{bplustree\_iter}\ k\ t\ \mathit{ti}\ r\ \mathit{vs}\ \mathit{it} * \uparrow (r' = (\mathit{vs} \neq []))>_t$

lemma $\mathit{bplustree\_iter}\ k\ t\ \mathit{ti}\ r\ \mathit{vs}\ \mathit{it} \Longrightarrow_A \mathit{bplustree\_assn}\ k\ t\ \mathit{ti}\ r\ \mathit{None} * \mathit{true}$

5.2 Range queries

A common use case of B$^+$-trees to obtain all values within a range [6]. We focus on the range of values in the tree bounded only from below by x, denoted by *lrange t x*. An iterator over this range can be obtained in logarithmic time. The operation is similar to the point query operation. On the leaf level, it returns a pointer to the reached leaf, that is interpreted as iterator on the list of linked leaves. The range bounded from below comprises all values returned by the iterator, the lower bound is its first element. Due to the lack of links on the abstract layer, the abstract definition explicitly concatenates all values in the subtrees to the right of the reached node.

```
fun lrange:: 'a bplustree ⇒ 'a ⇒ 'a list where
    lrange (Leaf ks) x = (lrange_list x ks) |
    lrange (Node ts t) x = (
        case split ts x of (_,(sub,sep)#rs) ⇒ (
              lrange sub x @ leaves_list rs @ leaves t
        )
     | (_,[]) ⇒ lrange t x
    )
```

As before, we assume that there exists a function *lrange_list* that obtains the *lrange* from a list of sorted values.

The verification of the imperative version turns out to be not as straightforward as expected, exactly due to this recursive step. The reason is that iterators can only be expressed on a complete tree, where the last leaf is explicitly a null pointer. The issue is a technicality. The *has_next* function in the iterator returns whether there are any remaining elements. We compare the current leaf with the last leaf of the tree. If the last leaf is a valid leaf node and not a null pointer, and the linked list supposedly empty, we need to show that the linked leaf list is not cyclic. We avoid this proof obligation by requiring that the last leaf is a null pointer. The linked list of a subtree is however bounded by valid leaves, precisely the first leaf of the next subtree.

Therefore we introduce an alternative formulation *concat_leaves_range* of the abstract function, similar in thought to how we obtained the iterator on the list from the first leaf of the tree. In a first step, we obtain the list of leaf nodes *leaves_range* (not the contents of them) based on the recursive search through the tree. In a second step, we obtain the head of *leaves_range* and apply *lrange_list*, to skip over the first values in the first array that are not part of the *lrange*. The result is concatenated with the tail of *leaves_range*.

On the imperative layer *leaves_range* can be obtained using only the *leaf_nodes* and *trunk* assertions. Only when we have obtained the list of leaves for the whole tree, we transform the result into an iterator over the leaves. At this point, the list is terminated by a null pointer and not the first leaf of the next sibling, such that we can obtain an iterator with the existing definition.

```
fun leaves_range:: 'a bplustree ⇒ 'a ⇒ 'a bplustree list where
  leaves_range (Leaf ks) x = [Leaf ks] |
  leaves_range (Node ts t) x = (
      case split ts x of (_,(sub,sep)#rs) ⇒ (
            leaves_range sub x @ leaf_nodes_list rs @ leaf_nodes t
      )
   | (_,[]) ⇒ leaves_range t x
  )

fun concat_leaves_range where
  concat_leaves_range t x = (case leaves_range t x of (LNode ks)#list ⇒
    lrange_list x ks @ (concat (map leaves list))
  )
```

Here, we apply the process of abstract refinement again. We first formulate *concat_leaves_range* on the abstract layer and verify that it yields the same result as *lrange*. Then we refine the approach to the imperative layer and can directly deduce that the approach yields the correct result.

```
lemma assumes k > 0 and root_order k t
    and sorted_less (leaves t) and Laligned t u shows
 <bplustree_assn k t ti r None>
  imp_concat_leaves_range ti x
  <tree_iter k t ti r (lrange t x)>t
```

6 Conclusion

We were able to formally verify an imperative implementation of the ubiquitous B$^+$-tree data structure. The implementation features functionality that has not been featured in previous implementations, covering range queries and efficient binary search.

6.1 Lessons learned

Handling separation logic formulae has always been a bit tedious throughout the research. A major alleviation was the introduction of a specialized tool that would substitute multiplicative terms in the formular regardless of the disctribution in the original term. It allowes i.e. the substitution of $a * c = d * e * f$ in the term $a * b * c$, yielding $d * e * f * c$. This was particularly useful for incrementally modifying equivalences of separation logic formulas.

What is currently missing in the implementation of the entailment solving tool is to eliminate multiplicative terms that already entail one another. The entailment $a{*}b{*}c \Rightarrow c{*}e{*}a$ would then be processed to the remaining proof obligation $b \Rightarrow e$ and not stopping without any elimination in case of failure to prove the entailment.

6.2 Evaluation

The B$^+$-tree implemented by Ernst *et al.* [4] features point queries and insertion, however explicitly leaves out pointers within the leaves, which forbids the implementation of iterators. Our work is closer in nature to the B$^+$-tree implementation by Malecha *et al.* [10]. In addition to the functionality dealt with in their work, we extend the implementation with a missing Range iterator and supply a binary search within nodes. Our approach is modular, allowing for the substitution of parts of the implementation with even more specialized and sophisticated implementations.

Regarding the leaf iterator, we noticed that in the work of Malecha *et al.* there is no need to extract the fringe explicitly. The abstract leaves are defined such that they store the precise heap location of the refining node. In this definition, the precise heap location is irrelevant in almost every situation and can be omitted, only its content is relevant to the

	[10]$^+$	[4]^d	Our approach$^+$
Functional code	360	-	413
Imperative code	510	1862	1093
Proofs	5190	350 + 510 + 2940[3]	8663
Timeframe (months)	-	6+	6[4] + 6

Figure 9 Comparison of (unoptimized) Lines of Code and Proof and time investment in related mechanized B$^+$-tree verifications. All approaches are comparable in effort, taking into account implementation specifics. The marker ^{d} denotes that the implementation verifies deletion operations, whereas $^+$ denotes the implementation of iterators.

user. Only when splitting the tree we obtain the memory location of nodes explicitly, and then only those locations that are needed to guarantee that the whole tree is structually sound. It is hard to quantify or evaluate which approach is more elegant in this respect. From a theoretical view point we suggest that an approach that is less strict about the heap state restricts the implementation space less and leaves more design decisions to the person implementing the specification.

With respect to the effort in lines of code and proof as depicted in Figure 9, we see that our approach is similar in effort to the approach by Malecha *et al.*. The numbers do not include the newly defined pure ML proof tactics. It should be also noted that this includes the statistics for the additional binary search and range iterator, that make up around 1000 lines of proof each. The comparison with Ernst *et al.* is difficult. Their research completely avoids the usage of leaf pointers, therefore also omitting iterators completely. The iterator verification makes up a signifant amount of the proof with at least 1000 lines of proof on its own. The leaf pointers also affect the verification of point and insertion queries due to the additional invariant on the imperative level. We conclude that the Isabelle/HOL framework provides a feature set such that verification of B$^+$-trees is both feasible and comparable in effort to using Ynot or KIV/TVLA. The strict separation of a functional and imperative implementation yields the challenge of making memory locations explicit where needed. On the other hand, it permits great freedom regarding the actual refinement on the imperative level.

References

1 Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. doi:10.1007/BF00288683.

2 Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with isabelle/hol. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2008. doi:10.1007/978-3-540-71067-7_14.

[3] The proof integrates TVLA and KIV, and hence comprises explicitly added rules for TVLA (the first number), user-invented theorems in KIV (the second number) and "interactions" with KIV (the second number). Interactions are i.e. choices of an induction variable, quantifier instantiation or application of correct lemmas. We hence interpret them as each one apply-Style command and hence one line of proof.

[4] 6 months include the preceding work on the verification of simple B-Trees. As they share much of the functionality with B$^+$-trees but required their own specifics, the time spent on them cannot be accounted for 1:1.

3 Douglas Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979. doi:10.1145/356770.356776.

4 Gidon Ernst, Gerhard Schellhorn, and Wolfgang Reif. Verification of b$^+$ trees by integration of shape analysis and interactive theorem proving. *Softw. Syst. Model.*, 14(1):27–44, 2015. doi:10.1007/s10270-013-0320-1.

5 Elizabeth Fielding. The specification of abstract mappings and their implementation as b+ trees. Technical Report PRG18, OUCL, 9 1980.

6 Goetz Graefe. Modern b-tree techniques. *Found. Trends Databases*, 3(4):203–402, 2011. doi:10.1561/1900000028.

7 Peter Lammich. Generating verified LLVM from isabelle/hol. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.22.

8 Peter Lammich. Refinement to imperative HOL. *J. Autom. Reason.*, 62(4):481–503, 2019. doi:10.1007/s10817-017-9437-1.

9 Peter Lammich and Rene Meis. A separation logic framework for imperative HOL. *Arch. Formal Proofs*, 2012, 2012. URL: https://www.isa-afp.org/entries/Separation_Logic_Imperative_HOL.shtml.

10 J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 237–248. ACM, 2010. doi:10.1145/1706299.1706329.

11 Niels Mündler. A verified imperative implementation of B-trees. *Arch. Formal Proofs*, 2021, 2021. URL: https://www.isa-afp.org/entries/BTree.html.

12 Tobias Nipkow. Automatic functional correctness proofs for functional search trees. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2016. doi:10.1007/978-3-319-43144-4_19.

13 Tobias Nipkow and Gerwin Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014. doi:10.1007/978-3-319-10542-0.

14 Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Michael Balser. Structured specifications and interactive proofs with kiv. *Automated Deduction - A Basis for Applications*, 2, 09 2000. doi:10.1007/978-94-017-0435-9_1.

15 Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002. doi:10.1145/514188.514190.

16 Alan P. Sexton and Hayo Thielecke. Reasoning about B+ trees with operational semantics and separation logic. In Andrej Bauer and Michael W. Mislove, editors, *Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2008, Philadelphia, PA, USA, May 22-25, 2008*, volume 218 of *Electronic Notes in Theoretical Computer Science*, pages 355–369. Elsevier, 2008. doi:10.1016/j.entcs.2008.10.021.