Formalized Verification of Snapshotable Trees: Separation and Sharing

Hannes Mehnert, Filip Sieczkowski, Lars Birkedal, and Peter Sestoft

IT University of Copenhagen
{hame, fisi, birkedal, sestoft}@itu.dk

Abstract. We use separation logic to specify and verify a Java program that implements snapshotable search trees, fully formalizing the specification and verification in the Coq proof assistant. We achieve *local* and *modular* reasoning about a tree and its snapshots and their iterators, although the implementation involves shared mutable heap data structures with no separation or ownership relation between the various data.
The paper also introduces a series of four increasingly sophisticated implementations and verifies the first one. The others are included as future work and as a set of challenge problems for full functional specification and verification, whether by separation logic or by other formalisms.

1 Introduction

This paper presents a family of realistic but compact challenge case studies for modular software verification, and shows how separation logic can address one challenge from that family using an approach similar to ramified frames [10]. The specification and verification are fully formalized in Coq, and we believe this is the first mechanical formalization of this approach to modular reasoning about implementations that use shared heap data with no separation or ownership between the various data.

The family of case studies consists of a single interface specification for snapshotable trees, and four different implementations. A *snapshotable tree* is an ordered binary tree that represents a set of items and supports taking readonly *snapshots* of the set, in constant time, at the expense of slightly slower subsequent updates to the tree. A snapshotable tree also supports iteration (enumeration) over its items as do, e.g., the Java collection classes. The four implementations of the snapshotable tree interface all involve shared heap data as well as increasingly subtle uses of destructive heap update.

For practical purposes it is important that the same interface specification can support verification of multiple implementations with varying degrees of internal sharing and destructive update. Moreover, the specification must accommodate any number of data structure (tree) instances, each having any number of iterators and snapshots, each of which in turn can have any number of iterators. Most importantly, we show how we can have local reasoning (a frame rule) even though the tree and its snapshots share mutable heap data.

We welcome other solutions to the specification and verification of this case study; indeed Leino has already made one (unpublished) using Dafny [11].

The Java source code of the case studies of all four implementations and the Coq source is available at `http://www.itu.dk/people/hame/snapshots.tar.gz`.

Section 2 presents the interface of the case study data structure, shows an example use, and outlines four implementations. Section 3 gives a formal specification of the interface using separation logic and verifies the example code. Sections 4 and 5 verify the first implementation.

2 Case Study: Snapshotable Trees

The case study is a simplified version of snapshotable treesets from the C5 collection library [8]. Section 2.1 presents the common interface, Section 2.2 presents client code for a simple application, and Section 2.3 gives an overview of four possible implementations of increasing sophistication.

2.1 Interface: Operations on Snapshotable Trees

Conceptually, a snapshot of a treeset is a readonly copy of the treeset. Subsequent updates to the tree do not affect any of its snapshots, so one can update the tree while iterating over a snapshot. Taking a snapshot must be a constant time operation, but subsequent updates to the tree may be slower after a snapshot has been taken. Implementations (Section 2.3) typically achieve this by making the tree and its snapshots share parts of their representation, gradually unsharing it as the tree gets updated, in a manner somewhat analogous to copy-on-write memory management schemes in operating systems.

All tree and snapshot implementations implement the same ITree interface:

```
public interface ITree extends Iterable<Integer> {
  public boolean contains(int x);
  public boolean add(int x);
  public ITree snapshot();
  public Iterator<Integer> iterator();
}
```

These operations have the following effect:

- `tree.contains(x)` returns true if the item is in the tree, otherwise false.
- `tree.add(x)` adds the item to the tree and returns true if the item was not already in the tree; otherwise does nothing and returns false.
- `tree.snapshot()` returns a readonly snapshot of the given tree. Updates to the given tree will not affect the snapshot. A snapshot cannot be made from a snapshot.
- `tree.iterator()` returns an iterator (also called enumerator, or stream) of the tree's items. Any number of iterators on a tree or snapshot may exist at the same time. Modifying a tree will invalidate all iterators on that tree (but not on its snapshots), so that the next operation on such an iterator will throw ConcurrentModificationException.

We include the somewhat complicated `iterator()` operation because it makes the distinction between a tree and its snapshots completely clear: While it is illegal to modify a tree while iterating over it, it is perfectly legal to modify the tree while iterating over one of its snapshots. Also, this poses an additional verification challenge when considering implementations with rebalancing (cases A2B1 and A2B2 in Section 2.3) because `tree.add(item)` may rebalance the tree in the middle of an iteration over a snapshot of the tree, and that should be legal and not affect the iteration.

Note that for simplicity, items are here taken to be integers; using techniques from [20] it is straightforward to extend our formal specification and verification to handle a generic version of snapshotable trees.

2.2 Example Client Code

To show what can be done with snapshots and iterators (and not without), consider this piece of client code. It creates a treeset `t`, adds three items to it, creates a snapshot `s` of the tree, and then iterates over the snapshot's three items while adding new items (6 and 9) to the tree:

```
ITree t = new Tree();
t.add(2); t.add(1); t.add(3);
ITree s = t.snapshot();
Iterator<Integer> it = s.iterator();
boolean lc = it.hasNext();
while (lc) {
  int x = it.next();
  t.add(x * 3);
  lc = it.hasNext();
}
```

2.3 Implementations of Snapshotable Trees

One may consider four implementations of treesets, spanned by two orthogonal implementation features. First, the tree may be left unbalanced (A1) or it may be actively rebalanced (A2) to keep depth $O(\log n)$. Second, snapshots may be kept persistent, that is, unaffected by tree updates, either by path copy persistence (B1) or by node copy persistence (B2):

	Without rebalancing	With rebalancing
Path copy persistence	A1B1	A2B1
Node copy persistence	A1B2	A2B2

The implementation closest to that of the C5 library [8, section 13.10] is A2B2, which is still somewhat simplified: only integer items, no comparer argument, no update events, and so on. In this paper we formalize and verify only implementation A1B1; the verification of the more sophisticated implementations A1B2, A2B1 and A2B2 will be addressed in future work.

Nevertheless, for completeness and in the hope that others may consider this verification challenge, we briefly discuss all four implementations and the expected verification challenges here.

With *path copy persistence* (cases AxB1), adding an item to a tree will duplicate the path from the root to the added node, if this is necessary to avoid modifying any snapshot of the tree. Thus an update will create $O(d)$ new nodes on average where d is the depth of the tree.

With *node copy persistence* (cases AxB2), each tree node has a spare child reference. The first update to a node uses this spare reference, does not copy the node and does not update its parent; the node remains shared between the tree and its snapshots. Only the second update to a node copies it and updates its parent. Thus an update does not replicate the entire path to the tree root; the number of new nodes per update is amortized $O(1)$. See Driscoll [6] or [8].

To implement ordered trees without rebalancing (cases A1By), we use a Node class containing an item (here an integer) and left and right children; `null` is used to indicate the absence of a child. A tree or snapshot contains a stamp (indicating the "time" of the most recent update) and a reference to the root Node object; `null` if the tree is empty.

To implement rebalancing of trees (cases A2By), we use left-leaning red-black trees (LLRB) which encode 2-3 trees [1, 19], instead of general red-black trees [7] as in the C5 library. This reduces the number of rebalancing cases.

To implement iterators on a tree or snapshot we use a class TreeIterator that holds a reference to the underlying tree, a stamp (the creation "time" of the iterator) and a stack of nodes. The stamp is used to detect subsequent updates to the underlying tree, which will invalidate the iterator. Since snapshots cannot be updated, their iterators are never invalidated. The iterator's stack holds its current state: for each node in the stack, the node's own item and all items in the right subtree have yet to be output by the iterator.

Case A1B1 = no rebalancing, path copy persistence In this implementation there is shared data between a tree and its snapshots, but the shared data is not being mutated because the entire path from the root to an added node gets replicated. Hence no node reachable from the root of a snapshot, or from nodes in its iterators' stacks, can be affected by an update to the live tree; therefore no operation on a snapshot can be affected by operations on the live tree. Although this case is therefore the simplest case, it already contains many challenges in finding a suitable specification for trees, snapshots and iterators, and in proving the stack-based iterator implementation correct.

Case A2B1 = rebalancing, path copy persistence In this case there is potential mutation of shared data, because the rebalancing rotations seem to be able to affect nodes just off the fresh unshared path from a newly added node to the root. This could adversely affect an iterator of a snapshot because a reference from the iterator's node stack might have its right child updated (by a rotation), thus wrongly outputting the items of its right subtree twice or not at all. However, this does not happen because the receiver of a rotation (to be moved down) is always a fresh node (we're in case B1 = path copy persistence) and moreover we consider only `add` operations (not `remove`), so the child being rotated (moved up) is also a fresh node and thus not on the stack of any iterator – the rebalancing was caused by this child being "too deep" in the tree. Hence if we were to support `remove` as well, we must consider whether the implementation of rotations must be refined.

Case A1B2 = no rebalancing, node copy persistence In this case, there is mutation of shared data not observable for the client. For example, a left-child update to a tree Node that is also part of a snapshot will move the snapshot's left-child value to Node's extra reference field, and destructively update the left child as required for the live tree. There should be no observable change to the snapshot, despite the change to the data representing it. The basic reason for correctness is that any snapshot traversing an updated node will use the extra reference and hence not see the update; this is true for nodes reachable from the root of a snapshot as well as for nodes reachable from the stack of an iterator. When we need to update a node whose extra reference is already in use, we leave the old node alone and create a fresh copy of the node for use in the live tree; again, existing snapshots and their iterators do not see the update.

Case A2B2 = rebalancing, node copy persistence In this case there is mutation of shared data (due both to moving child fields to the extra reference in nodes, and due to rotations), not observable for the client. Since the updates caused by rotations are handled exactly like other updates, the correctness of rebalancing with respect to iterators seems to be more straightforward than in case A2B1.

3 Abstract Specification and Client Code Verification

We use higher-order separation logic [18, 3] to specify and verify the snapshotable tree data structure. We build on top of an intuitionistic formalization of higher-order separation logic in Coq [2], developed as part of our research project.

To allow implementations to share data between a tree, its snapshots, and iterators and still make it possible for clients to reason locally (to focus only on a single tree / snapshot / iterator), we will use an idea from [10] (see also the verification of Union-Find in [9]). The idea is to introduce an abstract predicate, here named H, global to each tree data structure consisting of a single tree, multiple snapshots and multiple iterators. This abstract predicate H is parameterized by a finite set of disjoint *abstract structures*. We have three kinds of abstract structures: either a Tree, a Snap, or an Iter. The use of H enables a client of our specification to consider each abstract structure to be separate or disjoint from the rest of the abstract structures and thus the client can reason modularly about client code using only those abstract structures she needs; the rest can be framed out. Since the abstract predicate H is existentially quantified, the client has no knowledge of how an implementation defines H (see [3, 16] for more on abstract predicates in higher-order separation logic). The implementor of the tree data structure has a global view on the tree with its snapshots and iterators, and is able to define which parts of the abstract structures are shared in the concrete heap. In Section 4 we define H for the A1B1 case (Section 2.3).

The Tree abstract structure consists of a handle (reference) to the tree and a model, which is an ordered finite set, containing the elements of the tree. The Snap structure is similar to Tree. The Iter structure consists of a handle to the iterator and a model, which is a list containing the remaining elements for iteration. Because H is tree-global, exactly one Tree structure must be present ("the tree"), while the number of Snap and Iter structures is not constrained.

The remainder of this section contains the abstract specification of the ITree interface and the Iterator interface. In Section 3.3 we verify the client code presented in Section 2.2.

In a method's postcondition, variable `ret` is the return value.

3.1 Specification of the ITree Interface

We now present the formal abstract specification of the ITree interface informally described in Section 2.1. The specification is parametrized over a class C and the above-mentioned predicate H, and each method specification is universally quantified over a finite set of integers τ and a finite set of abstract structures ϕ.

`interface ITree` {

$\{H(\{Tree(\mathtt{this}, \tau)\} \uplus \phi)\}$ `contains(x)` $\{\mathtt{ret} = \mathtt{x} \in \tau \wedge H(\{Tree(\mathtt{this}, \tau)\} \uplus \phi)\}$
$\{H(\{Snap(\mathtt{this}, \tau)\} \uplus \phi)\}$ `contains(x)` $\{\mathtt{ret} = \mathtt{x} \in \tau \wedge H(\{Snap(\mathtt{this}, \tau)\} \uplus \phi)\}$
$\{H(\{Tree(\mathtt{this}, \tau)\} \uplus \phi)\}$ `add(x)` $\{\mathtt{ret} = \mathtt{x} \notin \tau \wedge H(\{Tree(\mathtt{this}, \{\mathtt{x}\} \cup \tau)\} \uplus \phi)\}$
$\{H(\{Tree(\mathtt{this}, \tau)\} \uplus \phi)\}$ `snapshot()` $\{H(\{Snap(\mathtt{ret}, \tau)\} \uplus \{Tree(\mathtt{this}, \tau)\} \uplus \phi)\}$
$\{H(\{Snap(\mathtt{this}, \tau)\} \uplus \phi)\}$ `iterator()` $\{H(\{Iter(\mathtt{ret}, [\tau])\} \uplus \{Snap(\mathtt{this}, \tau)\} \uplus \phi) \wedge \mathtt{ret} <: Iterator\}$

(a) $H(\{Tree(\mathtt{t}, \tau\})\} \uplus \phi) \vdash \mathtt{t} : C$
(b) $H(\{Snap(\mathtt{s}, \tau)\} \uplus \phi) \vdash \mathtt{s} : C$
(c) $\tau = \tau' \wedge H(\{Tree(\mathtt{t}, \tau)\} \uplus \phi) \vdash H(\{Tree(\mathtt{t}, \tau')\} \uplus \phi)$
(d) $H(\{Snap(\mathtt{s}, \tau)\} \uplus \phi) \vdash H(\phi)$
(e) $H(\{Iter(\mathtt{it}, \alpha)\} \uplus \phi) \vdash H(\phi)$
}

These specifications can be read as follows:

- `contains` requires either a Snap or Tree structure (written as separate specifications) for the `this` handle and some set τ. The structure is unmodified in the postcondition, and the return value is true if the item `x` is in the set τ, otherwise false.
- `add` requires a Tree structure for the `this` handle and some set τ. The postcondition states that the given item `x` is added to the set τ. The return value indicates whether the tree was modified, which is the case if the item was not already in the set τ.
- `snapshot` requires a Tree structure for the `this` handle (the interface in Section 2.1 does not support snapshots of snapshots) and some set τ. The postcondition constructs a Snap structure for the returned handle `ret` and the same set τ as the tree.

– `iterator` requires a Snap structure for the `this` handle and some set τ. The postcondition constructs an Iter structure with the return handle and the set τ converted to an ordered list, written $[\tau]$. The returned handle fulfills the Iterator specification (written <:), given in the next subsection.

The five axioms state that (a) the static type of the tree is the given class C; (b) the static type of a snapshot is C; (c) the model τ of the tree can be replaced by an equal model τ'; and we can forget about snapshots (d) and iterators (e).

In contrast to the description in Section 2.1 we do not consider iterators over the tree. An iterator over the tree becomes invalid when the tree is modified, we can express this using the ramification operator [10].

The abstract separation can be observed, e.g., in the specification of `add`: it only modifies the model of the Tree structure and does not affect the rest of the abstract structures (ϕ is preserved in the postcondition). Hence the client can reason about calls to `add` locally, independently of how many snapshots and iterators there are.

In our Coq formalization we do not have any syntax for interfaces at the specification logic level [2], but represent interfaces using Coq-level definitions. Appendix A contains the formal representation of the above interface ITree.

3.2 Iterator Specification

Our iterator specification is also parametrized over a class IC and a predicate H, and each method specification is universally quantified over a list of integers α and a finite set of abstract structures ϕ.

`interface Iterator<Integer> {`

$\{H(\{Iter(\mathtt{this}, \alpha)\} \uplus \phi)\}$ `hasNext()` $\{\mathtt{ret} = (|\alpha| \neq 0) \wedge H(\{Iter(\mathtt{this}, \alpha)\} \uplus \phi)\}$

$\{H(\{Iter(\mathtt{this}, x :: \alpha)\} \uplus \phi)\}$ `next()` $\{\mathtt{ret} = x \wedge H(\{Iter(\mathtt{this}, \alpha)\} \uplus \phi)\}$

`}`

The specification of the Iterator interface requires an Iter structure with the `this` handle and some list α. The return value of the method `hasNext` captures whether the list α is non-empty. The Iter structure in the postcondition is not modified. The method `next` requires an Iter structure with a non-empty list $(x :: \alpha)$. The list head is returned and the model of the Iter structure is updated to the remainder of the list.

3.3 Client Code Verification

To verify the client code from Section 2.2 we assume given a class C such that ITree C H holds for some H and then verify the client code under the precondition $\{H(\{Tree(\mathtt{t}, \{\})\})\}$.

Figure 1 gives a step-by-step proof of the client code from Section 2.2, with client code lines to the left and their postconditions to the right.

After inserting some items (line 1) to the tree, the model contains these items, {1, 2, 3}. In line 2, a snapshot `s` of the tree `t` is created. The invariant H now consists of the Tree structure and a Snap structure containing the same elements.

	$\{H(\{Tree(\texttt{t}, \{\})\})\}$
1: `t.add(2); t.add(1); t.add(3);`	$\{H(\{Tree(\texttt{t}, \{1,2,3\})\})\}$
2: `ITree s = t.snapshot();`	$\{H(\{Tree(\texttt{t}, \{1,2,3\})\} \uplus \{Snap(\texttt{s}, \{1,2,3\})\})\}$
3: `Iterator<Integer> it = s.iterator();`	$\{H(\{Tree(\texttt{t}, \{1,2,3\})\} \uplus \{Snap(\texttt{s}, \{1,2,3\})\} \uplus \{Iter(\texttt{it}, [1,2,3])\})\}$
4: `boolean lc = it.hasNext();`	$\{\texttt{lc} = \texttt{true} \wedge H(\{Tree(\texttt{t}, \{1,2,3\})\} \uplus \{Snap(\texttt{s}, \{1,2,3\})\} \uplus \{Iter(\texttt{it}, [1,2,3])\})\}$
5: `while (lc) {`	**invariant:** $\exists \alpha, \beta. \alpha @ \beta = [1,2,3] \wedge \texttt{lc} = (\lvert\beta\rvert \neq 0) \wedge H(\{Tree(\texttt{t}, \{1,2,3\} \cup \{3z \mid z \in \alpha\})\} \uplus \{Snap(\texttt{s}, \{1,2,3\})\} \uplus \{Iter(\texttt{it}, \beta)\})$
6: `int x = it.next();`	$\{\beta = \texttt{x} :: \beta' \wedge H(\{Tree(\texttt{t}, \{1,2,3\} \cup \{3z \mid z \in \alpha\})\} \uplus \{Snap(\texttt{s}, \{1,2,3\})\} \uplus \{Iter(\texttt{it}, \beta')\})\}$
7: `t.add(x * 3);`	$\{H(\{Tree(\texttt{t}, \{1,2,3\} \cup \{3z \mid z \in \alpha\} \cup \{3\texttt{x}\})\} \uplus \{Snap(\texttt{s}, \{1,2,3\})\} \uplus \{Iter(\texttt{it}, \beta')\})\}$
8: `lc = it.hasNext();`	$\{\texttt{lc} = (\lvert\beta'\rvert \neq 0) \wedge H(\{Tree(\texttt{t}, \{1,2,3\} \cup \{3z \mid z \in \alpha\} \cup \{3\texttt{x}\})\} \uplus \{Snap(\texttt{s}, \{1,2,3\})\} \uplus \{Iter(\texttt{it}, \beta')\})\}$
9: `}`	$\{H(\{Tree(\texttt{t}, \{1,2,3,6,9\})\} \uplus \{Snap(\texttt{s}, \{1,2,3\})\})\}$

Fig. 1. Client code verification

For the client the abstract structures are disjoint, but in an implementation, they will be realized using sharing. Indeed, for the A1B1 implementation, the concrete heap will be as shown in Figure 2, where all the nodes are shared between the tree and the snapshot.

In line 3 an iterator `it` over the snapshot `s` is created. To apply the call rule of the `iterator` method, only the Snap structure is taken into account, the rest (the Tree structure) is framed out inside of H (via appropriate instantiation of ϕ in the `iterator` specification). The result is that an Iter structure is constructed, whose model contains the same values as the model of the snapshot, but converted to an ordered list. We introduce the loop variable `lc` in line 4, and again use abstract framing to call `hasNext`.

Lines 5–9 contain a while loop with loop condition `lc`. The loop invariant splits the iteration list $[1,2,3]$ into the list α containing the elements already iterated over and the list β containing the remainder. The loop variable `lc` is false iff β is the empty list. The invariant H contains the Tree structure whose model is the initial set $\{1, 2, 3\}$ joined with the set of the elements of α, each multiplied by 3. H also contains the Iter and the Snap structures.

We omit detailed explanation of the remaining lines of verification.

Note that in the final postcondition, the client sees two disjoint structures (axiom (e) is used to forget the empty iterator), but in the A1B1 implementation, the concrete heap will involve sharing, as shown in Figure 3. Only the left subtree is shared by the tree and the snapshot; the root and right subtree were unshared by the first call to `add` in the loop.

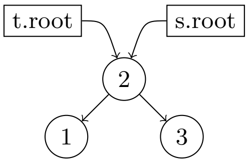

Fig. 2. Heap after snapshot construction

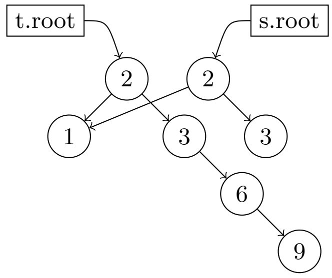

Fig. 3. Live heap after loop

In summary, we have shown the following theorem, which says that given any classes C and IC satisfying the ITree and Iterator interface specifications, the client code satisfies its specification. The postcondition says that snapshot `s` contains items 1, 2 and 3, and tree `t` contains also items 6 and 9.

Theorem 1. $\forall H.\forall C, IC. ITree\ C\ H \wedge Iterator\ IC\ H$
$\vdash \{H(\{Tree(\mathtt{t}, \{\})\})\}\ client\_code\ \{H(\{Tree(\mathtt{t}, \{1,2,3,6,9\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\})\}$

4 Implementation A1B1

In this section we show partial correctness verification of the A1B1 implementation, that it satisfies the abstract specification from the previous section. This involves defining a concrete H and showing that the methods satisfy the required specifications for this concrete H. The development has been formally verified in Coq (as has the client program verification above).

The Coq formalization uses a shallow embedding of higher-order separation logic in Coq, developed for verification of OO programs using interfaces. See [2].

Invariant H is radically different depending on whether snapshots of the tree are present or not. The reason is that method `add` mutates the existing tree if there are no snapshots present, see Section 5 for details. Here we focus on the case where snapshots are present.

The A1B1Tree class stores its data in three fields: the `root` node, a boolean field `isSnapshot`, indication whether it is a snapshot, and a field `hasSnapshot`, indicating whether it has snapshots. The stamp field mentioned in Section 2.3 is only required for iterators over the tree and so not further discussed here.

The Node class is a nested class of the A1B1Tree with three fields, `item` containing its value, and a handle to the right (`rght`) and left (`left`) subtree.

In the following we use standard separation logic connectives, in particular the separating conjunction $*$ and the points to predicate $\mapsto$.

We now define our concrete H and also the realization of the abstract structures. We first explain the realization of Tree and Snap; the Iter structure is described in Section 4.1. Recall that ϕ ranges over finite sets of abstract structures (Tree, Snap, Iter), with exactly one Tree structure, and recall that H, given a ϕ, returns a separation logic predicate. The definition of H is:

$$H(\phi) \triangleq \exists\sigma.(wf(\sigma) \wedge heap(\sigma)) * \sigma \vDash \phi$$

Here σ is a finite map of type ptr $\rightarrow$ ptr $\times \mathbb{Z} \times$ ptr, with ptr being the type of Java pointers (handles), corresponding to the Node class. The map σ must be well-formed, which simply means that all pointers in the codomain of σ are either `null` or in the domain of σ. When σ is extended (only in `addRecursive`), the pointer of the domain is always fresh, so there cannot be cycles in σ.

The *heap* function maps σ to a separation logic predicate, which describes the realization of σ as a linked structure in the concrete heap.

$$heap(\sigma) \triangleq \forall^* \mathtt{p} \in dom(\sigma).\exists \mathtt{pl}, v, \mathtt{pr}.\ \sigma[\mathtt{p}] = (\mathtt{pl}, v, \mathtt{pr}) \wedge \\ \mathtt{p.left} \mapsto \mathtt{pl} * \mathtt{p.item} \mapsto v * \mathtt{p.rght} \mapsto \mathtt{pr}$$

The iterating separating conjunction ($\forall^*$) is used here to ensure that all nodes in σ are disjoint in the heap.

Finally, we present the definition of $\sigma \vDash \phi$ (we defer the definition of $\sigma \vDash \{Iter(\_,\_)\}$ to the following subsection):

$$\begin{aligned}
\sigma \vDash \phi \uplus \psi &\triangleq \sigma \vDash \phi * \sigma \vDash \psi \\
\sigma \vDash \{Tree(\mathtt{ptr}, \tau)\} &\triangleq \exists \mathtt{p}.Node(\sigma, \mathtt{p}, \tau) \wedge \mathtt{ptr.root} \mapsto \mathtt{p} * \\
&\quad \mathtt{ptr.isSnapshot} \mapsto \mathtt{false} * \mathtt{ptr.hasSnapshot} \mapsto \mathtt{true} \\
\sigma \vDash \{Snap(\mathtt{ptr}, \tau)\} &\triangleq \exists \mathtt{p}.Node(\sigma, \mathtt{p}, \tau) \wedge \mathtt{ptr.root} \mapsto \mathtt{p} * \\
&\quad \mathtt{ptr.isSnapshot} \mapsto \mathtt{true} * \mathtt{ptr.hasSnapshot} \mapsto \mathtt{false}
\end{aligned}$$

The spatial structure of all the Nodes is covered by $heap(\sigma)$ so $\sigma \vDash \phi$ just needs to describe the additional heap taken up by Tree, Snap, and Iter structures.

The pure $Node(\sigma, \mathtt{p}, \tau)$ predicate, defined by induction on τ below, is used to express that τ is the finite set of items reachable from `p` in σ.

$$\begin{aligned}
Node(\sigma, \mathtt{p}, \tau) \triangleq\ & \big(\mathtt{p} = \mathtt{null} \wedge \tau = \{\}\big) \vee \\
& \big(\mathtt{p} \in dom(\sigma)\ \wedge \exists \mathtt{pl}, v, \mathtt{pr}.\ \sigma[\mathtt{p}] = (\mathtt{pl}, v, \mathtt{pr}) \wedge \\
& \exists \tau_l, \tau_r.\tau = \tau_l \cup \{v\} \cup \tau_r \wedge \\
& (\forall x \in \tau_l.x < v) \wedge (\forall x \in \tau_r.x > v) \wedge \\
& Node(\sigma, \mathtt{pl}, \tau_l) \wedge Node(\sigma, \mathtt{pr}, \tau_r)\big)
\end{aligned}$$

4.1 Iterator

The TreeIterator class implements the Iterator interface. It contains a single field, `context`, which is a stack of Node objects.

The constructor of the TreeIterator pushes all nodes on the leftmost path of the tree onto the stack. The method `next` pops the top node from the stack and returns the value held in that node. Before returning, it pushes the leftmost path of the node's right subtree (if any) onto the stack. The method `hasNext` returns true if and only if the stack is empty.

The verification of the iterator depends on the following specification of a stack class, where the notation $\widehat{f}$ lifts the function f such that it operates on expressions rather than values (a detailed explanation of this lifting is in [2]).

$$
\begin{aligned}
\mathit{Stack\_spec} \triangleq\ & \forall T : \mathit{Type}.\\
& \quad \exists SR : \mathit{classname} \to (\mathit{val} \to T \to \mathit{HeapAsn}) \to \mathit{val} \to T^* \to \mathit{HeapAsn}.\\
& \quad (\forall C : \mathit{classname}.\ \forall P : \mathit{val} \to T \to \mathit{HeapAsn}.\\
& \qquad \texttt{Stack::new}() \mapsto \{\top\}\_\{\texttt{r}.\ \widehat{SR}\ C\ P\ \texttt{r}\ \mathsf{nil}\}\\
& \quad \wedge (\forall \alpha : T^*.\ \texttt{Stack::empty}(\texttt{this}) \mapsto\\
& \qquad \{\widehat{SR}\ C\ P\ \texttt{this}\ \alpha\}\_\{\texttt{r}.\ \widehat{SR}\ C\ P\ \texttt{this}\ \alpha \wedge \texttt{r} = (\alpha = \mathsf{nil})\})\\
& \quad \wedge (\forall \alpha : T^*.\ \forall t : T.\ \texttt{Stack::push}(\texttt{this}, \texttt{x}) \mapsto\\
& \qquad \{\widehat{SR}\ C\ P\ \texttt{this}\ \alpha * \widehat{P}\ \texttt{x}\ t \wedge \texttt{x} : C\}\_\{\widehat{SR}\ C\ P\ \texttt{this}\ (t :: \alpha)\})\\
& \quad \wedge (\forall \alpha : T^*.\ \forall t : T.\ \texttt{Stack::pop}(\texttt{this}, \texttt{x}) \mapsto\\
& \qquad \{\widehat{SR}\ C\ P\ \texttt{this}\ (t :: \alpha)\}\_\{\widehat{SR}\ C\ P\ \texttt{this}\ \alpha\})\\
& \quad \wedge (\forall \alpha : T^*.\ \forall t : T.\ \texttt{Stack::peek}(\texttt{this}, \texttt{x}) \mapsto\\
& \qquad \{\widehat{SR}\ C\ P\ \texttt{this}\ (t :: \alpha)\}\_\{\texttt{r}.\ \widehat{P}\ \texttt{r}\ t *\\
& \qquad\ (\forall u : T.\ \widehat{P}\ \texttt{r}\ u \mathrel{-\!\!*} \widehat{SR}\ C\ P\ \texttt{this}\ (u :: \alpha))\})))\\
& \wedge (\forall C : \mathit{classname}.\ \forall P, P' : \mathit{val} \to T \to \mathit{HeapAsn}.\\
& \qquad (\forall v : \mathit{val}.\ \forall t : T.\ (P\ v\ t \vdash P'\ v\ t)) \implies\\
& \qquad\quad \forall v : \mathit{val}.\ \forall \alpha : T^*.\ (SR\ C\ P\ v\ \alpha \vdash SR\ C\ P'\ v\ \alpha))
\end{aligned}
$$

This specification is kept in the style of [17], although we use a different logic.

For the purpose of specifying the iterators over snapshotable trees, we instantiate the type T with $\mathbb{Z}^*$; the model of a node on the stack is a list of integers. Intuitively, this list corresponds to the node value and the element list of its right subtree. The iterator is modelled as a list that is equal to the concatenation of the elements of the stack. We also require that the topmost element of the stack is nonempty (if present). This intuition is formalized in the interpretation of the Iter structure, where SR is a representation predicate of a stack:

$$
\begin{aligned}
\sigma \vDash \{\mathit{Iter}(\texttt{p}, \alpha)\} \triangleq\ & \exists \texttt{st}.\ \texttt{p.context} \mapsto \texttt{st} * \exists \beta. \mathit{stack\_inv}(\beta, \alpha) \wedge\\
& SR\ \texttt{Node}\ (NS\ \sigma)\ \texttt{st}\ \beta.
\end{aligned}
$$

To make this definition complete, we provide the definitions of *stack_inv*, which connects the representation of the stack with the representation of the iterator, and the definition of the NS predicate.

$$
\mathit{stack\_inv}(xss, ys) \triangleq ys = \mathsf{concat}(xss) \wedge \begin{cases} \top & \text{iff } xss = \mathsf{nil} \\ xs \neq \mathsf{nil} & \text{iff } xss = xs :: xss' \end{cases}
$$

$$
NS\ \sigma\ \texttt{node}\ \alpha \triangleq \mathit{Node}(\sigma, \texttt{node}, \tau) \wedge\ \alpha = [\{x \in \tau | x \geq \texttt{node.item}\}]
$$

These definitions, along with an assumption that SR is the representation predicate of Stack (i.e., fulfills all the method specifications and axioms of *Stack_spec*) suffice to show the correctness of Iter-dependent methods. The axiom present in *Stack_spec* is needed to preserve iterators if some new memory is added to σ: it allows us to replace $(NS\ \sigma)$ with $(NS\ \sigma')$ as a representation predicate of stack objects under certain side conditions.

5 On the Verification of Implemented Code

We now give an intuitive description of how the A1B1 implementation was verified, given the concrete H defined above. We verified the complete implementation in Coq but only discuss the `add` method here. We used Kopitiam [13] to transform the Java code into SimpleJava, the fragment represented in Coq.

Method `add` allocates a RefBool cell containing boolean field `value`, then calls method `addRecursive` to insert the item into the binary tree, respecting the ordering. Method `addRecursive`, shown below, must handle several cases:

- if there are no snapshots present, then
  - if the item `x` is already in the tree, then the heap is not modified.
  - if the item `x` is not in the tree, then a new node is allocated and destructively inserted into the tree.
- if there are snapshots present, then
  - if the item `x` is already in the tree, then the heap is not modified.
  - if the item `x` is not in the tree, then a new node is allocated and every node on the path from the root to the added node is replicated, so that the snapshots are unimpaired.

The implementation of `addRecursive` walks down the tree until a node with the same value, or a leaf, is reached. It uses the call stack to remember the path in the tree. If a node was added, either the entire path from the root to the added node is duplicated (if snapshots are present) or the handles to the left or right subtree are updated (happens destructively exactly once, the parent of the added node updates its left or right handle, previously pointing to `null`):

```
Node addRecursive (Node node, int item, RefBool updated) {
  Node res = node;
  if (node == null) {
    updated.value = true;
    res = new Node(item);
  } else {
    if (item < node.item) {
      Node newLeft = addRecursive(node.left, item, updated);
      if (updated.value && this.hasSnapshot)
        res = new Node(node.rght, node.item, newLeft);
      else
        node.left = newLeft;
    } else if (node.item < item) {
      Node newRght = addRecursive(node.rght, item, updated);
      if (updated.value && this.hasSnapshot)
        res = new Node(newRght, node.item, node.left);
      else
        node.rght = newRght;
    } //else item == node.item so no update
  }
  return res;
}
```

We now show the pre- and postcondition of `addRecursive` for the two cases where snapshots are present.

$$\{\texttt{updated.value} \mapsto \texttt{false} * \texttt{this.hasSnapshot} \mapsto \texttt{true}*$$
$$heap(\sigma) * wf(\sigma) \wedge Node(\sigma, \texttt{node}, \tau) \wedge \texttt{item} \in \tau\}$$
$$\texttt{addRecursive(node, item, updated)}$$
$$\{\texttt{updated.value} \mapsto \texttt{false} * \texttt{this.hasSnapshot} \mapsto \texttt{true}*$$
$$heap(\sigma) * \texttt{ret} = \texttt{node}\}$$

The postcondition in the case that the item is added to the tree extends the map σ to σ', for which the heap layout and the well-formedness condition must hold. The Node predicate uses σ' and the finite set is extended with `item`:

$$\{\texttt{updated.value} \mapsto \texttt{false} * \texttt{this.hasSnapshot} \mapsto \texttt{true}*$$
$$heap(\sigma) * wf(\sigma) \wedge Node(\sigma, \texttt{node}, \tau) \wedge \texttt{item} \notin \tau\}$$
$$\texttt{addRecursive(node, item, updated)}$$
$$\{\texttt{updated.value} \mapsto \texttt{true} * \texttt{this.hasSnapshot} \mapsto \texttt{true}*$$
$$\exists \sigma'.\sigma \subseteq \sigma' \wedge heap(\sigma') * wf(\sigma') \wedge Node(\sigma', \texttt{ret}, \{\texttt{item}\} \cup \tau)\}$$

The call to `addRecursive` inside of `add` is verified for each specification of `addRecursive` independently.

To summarize Sections 4 and 5, we state the following theorem, which says that given a stack fulfilling the stack specification, the TreeIterator class meets the Iterator specification and the A1B1 implementation meets the ITree specification, and the constructor for the A1B1Tree establishes the H predicate.

Theorem 2. $\exists H.Stack\_spec \vdash Iterator\ TreeIterator\ H \wedge ITree\ A1B1\ H \wedge \{\top\}\ A1B1Tree()\ \{H(\{Tree(\texttt{ret}, \{\})\})\}$

Thus we can safely link the independently verified client code with the A1B1 implementation!

6 Related Work

Malecha and Morrisett [12] have presented a formalization of a Ynot implementation of B-trees with an iterator method. In their case, the iterator and the tree also share data in the concrete heap. However, they can only reason about "single-threaded" uses of trees and iterators: their specification of the iterator method transforms the abstract tree predicate into an abstract iterator predicate, which prohibits calling tree methods until the client turns the iterator back into a tree. In our setup, we have one tree, but allow for multiple snapshots and iterators, and the tree can be updated after an iterator has been created, as shown in our client code example. Malecha and Morrisett use fractional permissions to allow for the sharing parts of the heap between a tree and an iterator, whereas we use the H predicate to account for the sharing. We remark that Malecha and Morrisett are working in an axiomatic extension of Coq, whereas our proofs are done in a shallowly embedded program logic, since our programs are written in an existing real programming language (Java).

Dinsdale-Young et al. [5] present another approach for reasoning about shared data structures, which gives the client a fiction of disjointness. Roughly speaking, they define a new abstract program logic for each module (they can be combined) for abstract client reasoning. Their approach allows to give a client specification similar to ours, but without using the H and with the abstract structures (Tree / Snap / Iter) being predicates in the (abstract) program logic. This has the advantage that one can use ordinary framing for local reasoning.

Dinsdale-Young et al. [4] have also presented another approach for reasoning about sharing. Sharing can happen in certain regions, and the module implementor has to define a protocol which describes how data in the shared region can evolve. Again, what corresponds to our abstract structures can now be seen as separation logic predicates and thus, again, one can use ordinary framing for local reasoning.

For both approaches [5] and [4] the module implementor has more proof obligations than in our approach: In [5] he has to show that the abstract operations satisfy a number of side conditions related to how the abstract structures are realized in the concrete heap. In [4] he has to show related properties; here phrased in terms of certain stability conditions.

Compared to the work of Dinsdale-Young et al., our approach has the advantage that it is arguably simpler in that we do not need to introduce new separation (or context) algebras for the modules. That is why we could build our formalization on an implementation of standard separation logic in Coq.

7 Conclusion and Future Work

We have presented snapshotable trees as a challenge for formalized reasoning about mutable data structures that use sharing extensively, and given an abstract specification of the ITree interface. Moreover, we have presented a formalization of the A1B1 implementation of snapshotable treees.

The overall size of the formalization effort is roughly 5000 lines of Coq code and it takes 2 hours to qed the proofs. This is quite big compared to other formalization efforts of imperative programs in Coq, such as Hoare Type Theory / Ynot [14, 15]. The main reason is that we are working in a shallowly embedded program logic for a Java-like language, whereas Hoare Type Theory / Ynot is an axiomatic extension of Coq. Thus our formalization includes both the operational semantics of the Java subset and the soundness theorems for the program logic; also, Java program variables cannot simply be represented by Coq variables.

We also plan to verify the even subtler implementations A1B2, A2B1 and A2B2, which are expected to provide further insight into the challenges of dealing with shared mutable data and unobservable state changes. Through those more complex applications of separation logic we hope to learn more about desirable tool support, including how to automate the "obvious" reasoning that currently requires much thought and excessive amounts of proof code. Although we have not formally verified these implementations yet, we are fairly certain they would match the interface specification presented in Section 3. In all four implementations the tree is conceptually separate from its snapshots, which is the property required by the interface, and the invariant H allows us to describe the heap layout very precisely, using techniques shown in Section 4.

Finally, we would like to explore how to combine the advantages of our approach and those of Dinsdale-Young's approach discussed above.

References

1. A. Andersson. Balanced search trees made simple. In F. Dehne et al., editors, *Algorithms and Data Structures. LNCS 709*, pages 60–71. Springer-Verlag, 1993.
2. J. Bengtson, J. B. Jensen, F. Sieczkowski, and L. Birkedal. Verifying object-oriented programs with higher-order separation logic in Coq. In *ITP 2011*, 2011.
3. B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.
4. T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In T. DHondt, editor, *ECOOP 2010*, volume 6183 of *LNCS*, pages 504–528. Springer Berlin / Heidelberg, 2010.
5. T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Abstraction and refinement for local reasoning. In *VSTTE*, pages 199–215, Berlin, Heidelberg, 2010. Springer.
6. J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and Systems Sciences*, 38(1):86–124, 1989.
7. L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th FCS, Ann Arbor, Michigan*, pages 8–21, 1978.
8. N. Kokholm and P. Sestoft. The C5 Generic Collection Library for C# and CLI. Technical Report ITU-TR-2006-76, IT University of Copenhagen, January 2006.
9. N. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, 2011. Forthcoming.
10. N. R. Krishnaswami, L. Birkedal, and J. Aldrich. Verifying event-driven programs using ramified frame properties. In *TLDI*, pages 63–76. ACM, 2010.
11. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning. LNCS 6355*, pages 348–370, 2010.
12. G. Malecha and G. Morrisett. Mechanized verification with sharing. In *7th International Colloquium on Theoretical Aspects of Computing*, Sept. 2010.
13. H. Mehnert. Kopitiam: Modular incremental interactive full functional static verification of java code. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617, pages 518–524. Springer, 2011.
14. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In J. Hook and P. Thiemann, editors, *Proc. of 13th ACM ICFP 2008*, pages 229–240. ACM, 2008.
15. A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *Proceedings of POPL*, 2010.
16. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, pages 247–258, 2005.
17. R. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A realizability model for impredicative hoare type theory. In S. Drossopoulou, editor, *ESOP 2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 337–352. Springer, 2008.
18. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *IEEE Proc. of 17th Symp. on Logic in CS*, Nov 2002.
19. R. Sedgewick. Left-leaning red-black trees. At http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf.
20. K. Svendsen, L. Birkedal, and M. Parkinson. Verifying generics and delegates. In *ECOOP'10*, pages 175–199. Springer-Verlag, 2010.

A Appendix

We define here the ITree and the Iterator interface specification as Coq definitions. We use the name *SPred* for the finite set of abstract structures containing exactly one Tree structure and any number of Snap and Iter structures.

A detailed explanation of the notation can be found in [2].

$$
\begin{aligned}
\mathit{ITree} \triangleq\ & \lambda C : \mathit{classname}.\ \lambda H : \mathcal{P}_{fin}(\mathit{SPred}) \to \mathit{HeapAsn}.\\
&(\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{contains}(\mathtt{this}, \mathtt{x}) \mapsto\\
&\quad \{\widehat{H}(\{\widehat{\mathsf{Tree}}(\mathtt{this}, \tau)\} \uplus \phi)\}\_\{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Tree}}(\mathtt{this}, \tau)\} \uplus \phi) \wedge \mathtt{r} = (\mathtt{x} \in \tau)\})\\
&\wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{contains}(\mathtt{this}, \mathtt{x}) \mapsto\\
&\quad \{\widehat{H}(\{\widehat{\mathsf{Snap}}(\mathtt{this}, \tau)\} \uplus \phi)\}\_\{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Snap}}(\mathtt{this}, \tau)\} \uplus \phi) \wedge \mathtt{r} = (\mathtt{x} \in \tau)\})\\
&\wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{add}(\mathtt{this}, \mathtt{x}) \mapsto\\
&\quad \{\widehat{H}(\{\widehat{\mathsf{Tree}}(\mathtt{this}, \tau)\} \uplus \phi)\}\_\{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Tree}}(\mathtt{this}, \{\mathtt{x}\} \cup \tau)\} \uplus \phi) \wedge \mathtt{r} = (\mathtt{x} \notin \tau)\})\\
&\wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{snapshot}(\mathtt{this}) \mapsto\\
&\quad \{\widehat{H}(\{\widehat{\mathsf{Tree}}(\mathtt{this}, \tau)\} \uplus \phi)\}\_\{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Tree}}(\mathtt{this}, \tau), \widehat{\mathsf{Snap}}(\mathtt{r}, \tau)\} \uplus \phi)\})\\
&\wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{iterator}(\mathtt{this}) \mapsto\\
&\quad \{\widehat{H}(\{\widehat{\mathsf{Snap}}(\mathtt{this}, \tau)\} \uplus \phi)\}\_\{\mathtt{r}.\ \exists IC : \mathit{classname}.\ \mathit{Iterator}\ IC\ H \wedge \mathtt{r} : IC \wedge\\
&\qquad \widehat{H}(\{\widehat{\mathsf{Snap}}(\mathtt{this}, \tau), \widehat{\mathsf{Iter}}(\mathtt{r}, [\tau])\} \uplus \phi)\})\\
&\wedge (\forall v : \mathit{val}.\ \forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\\
&\quad (H(\{\mathit{Tree}(v, \tau)\} \uplus \phi) \implies v : C) \wedge (H(\{\mathit{Snap}(v, \tau)\} \uplus \phi) \implies v : C))\\
&\wedge (\forall v : \mathit{val}.\ \forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\\
&\quad (H(\{\mathit{Snap}(v, \tau)\} \uplus \phi) \vdash H(\phi)))\\
&\wedge (\forall v : \mathit{val}.\ \forall \alpha : \mathbb{Z}^*.\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\\
&\quad (H(\{\mathit{Iter}(v, \alpha)\} \uplus \phi) \vdash H(\phi)))\\
&\wedge (\forall v : \mathit{val}.\ \forall \tau, \tau' : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\\
&\quad \tau = \tau' \implies (H(\{\mathit{Tree}(v, \tau)\} \uplus \phi) \vdash H(\{\mathit{Tree}(v, \tau')\} \uplus \phi)))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Iterator} \triangleq\ & \lambda C : \mathit{classname}.\ \lambda H : \mathcal{P}_{fin}(\mathit{SPred}) \to \mathit{HeapAsn}.\\
&(\forall \alpha : \mathbb{Z}^*.\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{hasNext}(\mathtt{this}) \mapsto\\
&\quad \{\widehat{H}(\{\widehat{\mathsf{Iter}}(\mathtt{this}, \alpha)\} \uplus \phi)\}\_\{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Iter}}(\mathtt{this}, \alpha)\} \uplus \phi) \wedge \mathtt{r} = (\alpha \neq \mathsf{nil})\})\\
&\wedge (\forall x : \mathbb{Z}.\ \forall \alpha : \mathbb{Z}^*.\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{next}(\mathtt{this}) \mapsto\\
&\quad \{\widehat{H}(\{\widehat{\mathsf{Iter}}(\mathtt{this}, x{::}\alpha)\} \uplus \phi)\}\_\{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Iter}}(\mathtt{this}, \alpha)\} \uplus \phi) \wedge \mathtt{r} = x\})
\end{aligned}
$$