Verification of Snapshotable Trees using Access Permissions and Typestate

Hannes Mehnert[1] and Jonathan Aldrich[2]

[1] IT University of Copenhagen, 2300 København, Danmark
hame@itu.dk
[2] School of Computer Science, Carnegie Mellon University, Pittsburgh, USA
aldrich@cs.cmu.edu

Abstract. We use access permissions and typestate to specify and verify a Java library that implements snapshotable search trees, as well as some client code. We formalize our approach in the Plural tool, a sound modular typestate checking tool. We describe the challenges to verifying snapshotable trees in Plural, give an abstract interface specification against which we verify the client code, provide a concrete specification for an implementation and describe proof patterns we found. We also relate this verification approach to other techniques used to verify this data structure.

1 Introduction

In this paper we use access permission and typestate to formally verify snapshotable search trees in Plural [4, Chapter 6]. Snapshotable trees have been proposed as a verification challenge [10], because they contain abstract separation and internal sharing: the implementation uses sharing, while the user sees each tree and snapshot separately. The complete verified code is available at `http://www.itu.dk/people/hame/SnapTree.java`.

We only verify API compliance rather than full functional correctness in this paper. The protocol of the data structure is verified, rather than the tree content. The protocol is intricate, with internal sharing that is hidden from the client. The tree content could be modeled as a set, but in Plural no reasoning about sets is implemented.

We will first recapitulate the snapshotable tree verification challenge [10], typestate, and access permissions. Then we will briefly describe Plural and introduce our solution to the challenge.

To our knowledge this is the first formal verification of a tree data structure using access permissions and typestate. The verification of the Composite pattern [6], which consists of a tree data structure, used non-formalized extensions of Plural and was not formalized in Plural.

Snapshotable Search Trees A snapshotable search tree is an ordered binary tree with the additional method `snapshot`, which returns a handle to a read-only persistent view of the tree. Both the tree and the snapshot implement the same interface ITree. While the client can think of a tree and a snapshot as disjoint, the actual implementation requires that `snapshot` be computed in constant time. This is achieved by sharing the nodes between the tree and its snapshots. If a new node is inserted into the tree, the nodes are lazily duplicated (copy on write).

There are two implementation strategies, *path copy persistence* and *node copy persistence* [8]. While the former duplicates the entire path from the root node to the freshly inserted node, the latter has an additional handle in each node, which is used for the first mutation of the node.

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

The methods of the ITree interface have the following effects:

– `contains` return true if the given item is in the tree, otherwise false.
– `add` inserts the given item into the tree. If the item was already present, this method does not have any effect and its return value is false, otherwise true.
– `snapshot` returns a readonly view of the current tree. Taking a snapshot of a snapshot is not supported.
– `iterator` returns an iterator of the tree's (or snapshot's) items.

We consider only iterators over snapshots for the remainder of the paper. There is no limit to the number of iterators over a snapshot. Iterators over a snapshot are valid even if the original tree is mutated.

Our client code uses this behaviour, and iterates over the snapshot while mutating the original tree:

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

The client code adds some elements to a ITree, creates a snapshot, and iterates over the snapshot while it adds more items to the underlying tree. The client code is computationally equivalent to the original challenge [10], we do not introduce an unnecessary boolean variable for the loop condition.

Typestate Typestate systems [12] were developed to enhance reliability of software. A developer specifies the API usage protocol (a finite state machine) directly in the code. These protocols are statically checked. Empirical results [2] have shown that API protocol definitions occur three times more often than definitions of generics in object-oriented (Java) code. In Plaid, an upcoming programming language, typestate is a first-class citizen [13] and has been incorporated into the type system.

A motivating example for typestate is the *File* class, shown in Figure 1. Reading a file is only valid if it is open, thus the abstract *File* class has two states, *Opened* and *Closed*, and the method `read` is only defined in the *Opened* state. The method `open` (only defined in the *Closed* state) transitions the object from the *Closed* to the *Opened* state (indicated by >>), and vice versa for `close`.

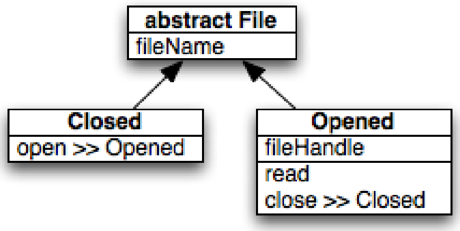

Fig. 1. Typestate example

This prevents common usage violations, like trying to read a closed file or opening a file multiple times.

Access Permissions A developer can annotate references with alias information [1] by using access permissions [4]. Access permissions are used for controlling the flow of linear objects. In the presented system there are five different permissions: exclusive access (*unique*), exclusive write access with others possibly having read access (*full*), shared write access (*share*), read-only access with others possibly having write access (*pure*) and immutable access in which no others can write either (*immutable*).

Boyland et al [7] presented fractions to reason about permissions. This allow us to split and join permissions: for example a *unique* permission can be split into a *full* and a *pure*, which can later be merged back together.

Plural The Plural[3] tool does sound and modular typestate checking; it employs fractional permissions to provide flexible alias control.

[3] `https://code.google.com/p/pluralism/`

Plural was implemented as a plugin for Eclipse, on top of the Crystal framework[4]. It consists of a static dataflow analysis which tracks constraints about permissions in a lattice and infers local permissions.

A developer can annotate each interface with abstract states, specified by name. Interface methods can be annotated with pre- and postconditions (required and ensured permissions and states).

Each class can be annotated with concrete states, which consist of a name and an invariant: a linear logic formula consisting of the access permission to a field in a specific state, or the (boolean or non-null) value of a field.

In a formula the standard linear logic conjunctions are available: *implies* ($\multimap$, written `=>`), *and* ($\otimes$, written `*`), *or* ($\oplus$, written `+`) and external choice ($\&$).

Each state can be a refinement of another state; a state can be refined multiple times. We use this to refine the default *alive* state.

In order to access the fields of an object, the object must be unpacked, allowing temporary violations of the object's state invariants. Special care has to be taken to not unpack the same object multiple times (by using different aliases and permissions thereof), because this leads to unsoundness. Plural enforces the restriction that only a single object can be unpacked at a time. Before a method is called, all objects must be packed. Plural makes an exception to this rule for objects with unique permission, which is obviously sound since there cannot be any aliases to these objects.

Overview In Section 2 the interface specification and client code verification will be shown. Section 3 describes some proof patterns used in the verification of the actual implementation. In Section 4 we will describe related work and in Section 5 we conclude and present future work.

Our study is based on Plural, and indeed we observed certain low-level tool-specific artifacts (discussed in the conclusion) and the Plural-specific "ghost method" proof pattern. The main focus of this paper, however, including all other specification and proof patterns in Sections 2–3, is a high-level application of typestate and permission concepts to verify the tree and its clients. This may provide insights useful in other settings based on permissions [1, 7] and/or typestate [12, 13].

2 Interface Specification and Client Code Verification

The verification challenge is to give an abstract specification that does not expose implementation details, is usable by a client, and for each state an invariant can be specified by an implementor to verify her implementation.

We will first describe the specification of the interface ITree and Iterator, and afterwards we will show the verification of the client code using those specifications.

[4] `https://code.google.com/p/crystalsaf/`

2.1 Interface ITree

We specify the interface ITree by having two disjoint typestates, *Tree* and *Snapshot*, which keep track whether the object is a tree or a snapshot of a tree. The `marker=true` annotation ensures that the state cannot change during the lifetime of an object. Both states refine the default state *alive*.

```
@States(refined="alive", value={"Tree", "Snapshot"}, marker=true)
interface ITree extends Iterable<Integer> {
  @Pure
  public boolean contains(int item);
  @Full(requires="Tree", ensures="Tree")
  public boolean add(int item);
  @Full(requires="Tree", ensures="Tree")
  @ResultPure(ensures="Snapshot")
  public ITree snapshot();
  @Pure(requires="Snapshot", ensures="Snapshot")
  @ResultUnique
  @Capture(param="underlying")
  public TreeIterator iterator();
}
```

The annotations are intuitive: the method `contains` requires a *pure* permission in any typestate, and returns the very same permission. The method `add` requires a *full* permission in the *Tree* state; the method `snapshot` requires a *full* permission in the *Tree* state and the return value has a *pure* permission in the *Snapshot* state. The `iterator` method requires a *pure* permission in the *Snapshot* state, whereas the resulting iterator will have a *unique* permission. The `Capture` annotation indicates that `this` is captured by the returned `TreeIterator` object.

The access permissions and typestates formalize the informal constraints presented in the description of the ITree interface in Section 1.

2.2 Interface Iterator

Iterators have been specified previously in Plural [3], we include the specification for self-containedness of this paper. We follow similar ideas (namely a non-empty and empty state), whereas our implementation is different (see Section 3.5).

There are three states defined for an iterator, *NonEmpty*, *Empty* and *Impossible*, all refine *alive*. The last one is only for specifying the `remove` method which throws an exception in our implementation.

The method `next` requires *unique* permission to a *NonEmpty* iterator. The `hasNext` method requires *immutable* permission and if it returns true, the object is in the *NonEmpty* state, if false is returned, it is in the *Empty* state.

The need for a *unique* permission is due to recursive calls and Plural's restriction of having only a single unpacked object, mentioned in Section 1. We will discuss this in more detail when we show the iterator implementation in Section 3.5. This is a marginal drawback, since in practice iterators are used on the stack rather than shared via the heap.

This specification actually enforces that *hasNext* is called before each call to *next*, because otherwise the iterator is not known to be in the *NonEmpty* state.

```
@States(refined="alive", value={"NonEmpty", "Empty", "Impossible"})
interface TreeIterator extends Iterator<Integer> {
  @Unique(requires="NonEmpty")
  public Integer next();
  @Imm
  @TrueIndicates("NonEmpty")
  @FalseIndicates("Empty")
  public boolean hasNext();
  @Unique(requires="Impossible")
  public void remove();
}
```

2.3 Client Code Verification

The client code needs only a single annotation, that it has *full* permission in the *Tree* state of the given argument.

```
class ClientCode {
  @Perm(requires="full(#0) in Tree")
  void client (ITree t) {
    t.add(2); t.add(1); t.add(3);
    ITree s = t.snapshot();
    TreeIterator it = s.iterator();
    while (it.hasNext()) {
      int x = it.next();
      t.add(x * 3);
    }
  }
}
```

The method `client` adds the elements 1, 2 and 3 to the tree (line 4), creates a snapshot `s` (line 5) and an iterator `it` over the snapshot (line 6). The body of the while loop (lines 8 and 9) adds more elements to the original tree (line 9).

In this section we have demonstrated that the client code preserves the required permissions and states, using the given specification for the ITree and Iterator interfaces.

3 Proof Patterns and Verification of the Implementation

We have verified the A1B1 implementation [10], which does not implement rebalancing and uses path copy persistence: when a snapshot is present, the complete path from the root down to the newly inserted node is copied in a call to add. This ensures that add does not mutate any node that is shared between the snapshot and the tree.

The specifications of field getters, field setters, and constructors are omitted in the paper: they are straightforward, a field getter requires an *immutable* permission, a field setter a *full* permission and the constructor ensures a *unique* permission.

The SnapTree class, which implements the ITree interface, contains two boolean fields, isSnapshot and hasSnapshot, and a field root, which contains a handle to the root node.

3.1 Formula Guarded by a Boolean Variable and Implication

The invariant for *Snapshot* is straightforward. It contains an *immutable* permission to the root in the *PartOfASnapshot* state; the isSnapshot field is true, and the hasSnapshot field is false.

The field isSnapshot is used in the invariant to distinguish between the *Tree* and *Snapshot* states.

For the *Tree* invariant we distinguish between two cases: either there is a snapshot present, or there is no snapshot present. In the former case the invariant contains an *immutable* permission to the root node in the *PartOfASnapshot* state. This ensures the no node is mutated. In the latter case the invariant contains a *unique* permission to the root node in the *NotPartOfASnapshot* state.

To implement this conditional we use a proof pattern: the permission is guarded by an implication whose left hand side tests a boolean program variable. The variable hasSnapshot is compared to true (or false), and on the right hand side of the implication we have an *immutable* (or *unique*, respectively) permission to the root node in the *PartOfASnapshot* (or *NotPartOfASnapshot*) state.

```
@ClassStates({
  @State(name="Snapshot", inv="immutable(root) in PartOfASnapshot *
    isSnapshot == true * hasSnapshot == false")
  @State(name="Tree", inv="isSnapshot == false *
    (hasSnapshot == true => immutable(root) in PartOfASnapshot) *
    (hasSnapshot == false => unique(root) in NotPartOfASnapshot)"),
})
```

This distinction between the two cases is natural and follows from the program implementation.

3.2 Specification of a Recursive Structure

This implementation either contains a completely immutable tree (if snapshots are present) or a mutable tree. This is specified by the invariants of the states of the node class. Two states are defined, and both refine *alive*: either the node is part of a snapshot (*PartOfASnapshot*) or not part of a snapshot (*NotPartOfASnapshot*). The invariant recursively contains *immutable* (or *unique*) permissions in the *PartOfASnapshot* (or *NotPartOfASnapshot*, respectively) state for the left and right children.

```
@Refine({
  @States(refined="alive",
          value={"PartOfASnapshot","NotPartOfASnapshot"}),
})
@ClassStates({
  @State(name="PartOfASnapshot",
         inv="immutable(left) in PartOfASnapshot *
              immutable(rght) in PartOfASnapshot"),
  @State(name="NotPartOfASnapshot",
         inv="unique(left) in NotPartOfASnapshot *
              unique(rght) in NotPartOfASnapshot")
})
```

The base case for the recursion is that both the left and the right child are null. Plural assumes the possibility that these might be null by default.

3.3 Conditional Composition of Implementations

The method `add` behaves differently for a mutable tree and an immutable one. The `add` method in the SnapTree checks in which case the tree is and calls the correct method, either a mutating or a functional insert. In both cases the precondition and postcondition are a *full* permission to an object in the *Tree* state. The annotation `use=Use.FIELDS` specifies that `this` has to be unpacked in the method body, which is required to access the fields.

The implementation first checks whether `root` is null and instantiates a new `Node` object if that is the case. Otherwise the boolean field `hasSnapshot` is tested to determine whether a mutating insert (`addM`) or a functional insert (`addF`) should be done. The proof goes through because the test is the same as in the invariant of the *Tree* state, thus one guard is false, its implication is eliminated, and the other guarded formula is used.

```
@Full(use=Use.FIELDS, requires="Tree", ensures="Tree")
public boolean add (int i) {
  assert(isSnapshot == false);
  if (root == null) {
    setRoot(new Node(i));
    return true;
  } else
    if (hasSnapshot) {
      RefBool x = new RefBool();
      setRoot(root.addF(i, x));
      return x.getValue();
    } else {
      RefBool x = new RefBool();
      root.addM(i, x);
      return x.getValue();
    }
}
```

The implementation of `addF` requires an *immutable* permission of the node in the *PartOfASnapshot* state, and ensures an *immutable* permission in the *PartOfASnapshot* state for the returned object. It recurses down the tree to find the location at which to insert the given value, and if the value was inserted, it duplicates the entire path (which is on the call stack). It uses some helper methods to get and set fields.

```
@Perm(requires="immutable(this) in PartOfASnapshot",
      ensures="immutable(result) in PartOfASnapshot")
public Node addF (int i, RefBool x) {
  Node node = this;
  if (item > i) {
    Node lef = getLeft();
    Node newL = null;
    if (lef == null) {
      newL = new Node(i);
      x.setValue(true);
    } else
      newL = lef.addF(i, x);
    if (x.getValue()) {
      Node r = getRight();
      node = new Node(newL, item, r);
    }
  } else if (i > item) {
    Node rig = getRight();
    Node newR = null;
    if (rig == null) {
      newR = new Node(i);
      x.setValue(true);
    } else
      newR = rig.addF(i, x);
    if (x.getValue()) {
      Node l = getLeft();
      node = new Node(l, item, newR);
    }
  }
  return node;
}
```

The implementation of `addM` also searches for the correct place by calling itself recursively, and assigns a freshly instantiated Node object to that place.

```
@Unique(use=Use.DISP_FIELDS,
        requires="NotPartOfASnapshot",
        ensures="NotPartOfASnapshot")
public void addM (int i, RefBool x) {
  if (item > i)
    if (left == null) {
      left = new Node(i);
      x.setValue(true);
    } else
      left.addM(i, x);
  else if (i > item)
    if (rght == null) {
      rght = new Node(i);
      x.setValue(true);
    } else
      rght.addM(i, x);
}
```

3.4 Dropping Privileges (Ghost Method)

The method `snapshot` requires a *full* permission to this in the *Tree* state. The `!fr` annotation is equivalent to `use=Use.FIELDS`, but can be used in the more general `Perm` annotation.

The implementation of `snapshot` needs to drop the permissions to all nodes, because they are now shared with the tree and the snapshot. This is achieved in the `snapall` method.

```
@Perm(requires="full(this!fr) in Tree",
      ensures="pure(result) in Snapshot * full(this!fr) in Tree")
public ITree snapshot() {
  assert(!isSnapshot);
  if (hasSnapshot)
    return new SnapTree(root);
  else {
    Node r = root;
    r.snapall();
    hasSnapshot = true;
    return new SnapTree(r);
  }
}
```

The method `snapall` drops the privileges recursively by traversing the tree. It is implemented in the Node class. It does not have any observable computational effect, but it is required because we must drop the permissions for the entire tree and Plural only allows this to occur as each node is unpacked going down the tree. In order to verify it with Plural, we need to specifically assign null to the left/right sibling if it is already null (to associate a bottom permission).

```
@Perm(requires="unique(this!fr) in NotPartOfASnapshot",
      ensures="immutable(this!fr) in PartOfASnapshot")
public void snapall () {
  if (left != null)
    left.snapall();
  else
    left = null;
  if (rght != null)
    rght.snapall();
  else
    rght = null;
}
```

Although the specific technique used here is specialized for Plural, note that an analogous mechanism would be required to convince any tool that the permissions and/or typestates are dropped recursively.

3.5 Iterator

The iterator implementation uses a field `context`, which contains a stack of nodes that have not yet been yielded to the client. This is initially filled recursively with the left path, and whenever an item is popped from the stack, the left path of its right subtree is pushed onto the stack. The Stack class is annotated with a proper specification, but its implementation is not verified (especially that `pop` returns an object in the *PartOfASnapshot* typestate).

```
@Perm(requires="immutable(this) in Snapshot",
      ensures="unique(result)")
public TreeIterator iterator() {
  Node r = this.getRoot();
  TreeIteratorImpl it = new TreeIteratorImpl(this);
  it.pushLeftPath(r);
  return it;
}
```

The concrete class specifies an invariant only for the top-level `alive` state:

```
@ClassStates({
  @State(name="alive",
         inv="immutable(tree) in Snapshot * unique(context) in alive")
})
```

The method `pushLeftPath` calls itself recursively with the left child to push the entire left path onto the stack. It requires a *unique* permission to `this` in order to unpack `this`, access the `context` field, and call a method on the `context` object while `this` remains unpacked. As mentioned in Section 1, for soundness reasons, leaving an object unpacked during a method call is only possible in Plural if there is a *unique* permission to the unpacked object.

```
@Perm(requires="unique(this!fr) in alive * immutable(#0) in PartOfASnapshot",
      ensures="unique(this!fr) in alive * immutable(#0) in PartOfASnapshot")
public void pushLeftPath(Node node) {
  if (node != null) {
    context.push(node);
    pushLeftPath(node.getLeft());
  }
}
```

The `hasNext` method is simply a check whether the stack is non-empty.

```
@TrueIndicates("NonEmpty")
@FalseIndicates("Empty")
@Imm(use=Use.FIELDS)
public boolean hasNext() {
  return !context.empty();
}
```

The method `next` pops the first element of the stack and pushes the left path of the right child onto the stack. In contrast to the original implementation [10], a guard if `hasNext()` is true around lines 4-7 is not needed, because Plural verifies that `next` is only called on a non-empty iterator.

```
@Unique(use=Use.DISP_FIELDS, requires="NonEmpty")
public Integer next() {
  Integer result;
  Node node = context.pop();
  result = node.getItem();
  if (node.getRight() != null)
    pushLeftPath(node.getRight());
  return result;
}
```

Here a *unique* permission is required in order to call `pushLeftPath`.

In this section we described proof patterns used in the verification of the path copy persistence implementation of snapshotable trees. The complete implementation has been automatically verified with Plural.

4 Related Work

The Composite pattern, which is a tree data structure, has been verified using typestate and access permissions [6]. This work differed in multiple aspects: first of all it was not formalized in a tool, then it relied on extensions, like multiple unpacking and equations using pointers, which were not proven to be sound. Also, the verification challenge is different: the Composite pattern exposes all nodes to a user using a *share* permission, and preserves an invariant upwards the tree, namely the number of children of the subtree rooted in each node. This leads to a specification with several typestates in the different dimensions of each node, which fractions are cleverly distributed to allow for bottom-up updates of the count.

A prior iterator verification [3] is similar to our specification, but the implementation of the iterator is completely different. In this paper we present an iterator which shares its content with the snapshot and holds only some elements on the stack, pushing more onto the stack on demand.

Snapshotable trees have been verified using a higher-order separation logic [10]. This approach verified full functional correctness, while this paper can only prove correct API usage: `add` and `snapshot` are always called on the tree, and by having *immutable* permission to the contents of a snapshot, we can verify that it will not be modified. Also, our work verifies that an iterator is always taken on a snapshot, not the original tree, and that `next` is never called on an empty iterator.

We use automation in the proof, which requires only a moderate number of annotations to the source code. The higher-order separation logic proof requires roughly 5000 lines of proof script, while the code and annotations for this paper are together under 400 lines; this is less than 2 lines of annotation for every line of source code.

An unpublished verification of snapshotable trees in Dafny [9], done by Rustan Leino, is similar to the Plural approach. Both are automated systems using a first-order logic. In Dafny functional correctness can be proven. The advantage of Plural is that already existing code written in a widely deployed programming language (Java) can be analyzed, whereas Dafny specifies its own programming language. Dafny uses implicit framing and also relies on annotations by the user, whereas Plural is based on linear logic (access permissions) and typestates. Dafny does not support inheritance, thus no abstract specification is provided.

5 Conclusion and Further Work

There exist several extensions to the access permission system which support verifying full functional correctness: Object propositions [11] combine access permissions with first-order formulae; but there is currently no implementation available. Symplar [5] combines access permissions with JML, thus access permissions are used to reason about aliasing, and JML formulae for full functional correctness.

In order to verify iterators over the tree (vs. its snapshots) we would need to change the *unique* permission of the nodes to *full* in order to share them between the tree and the iterator. Because the proof relies on method calls while a *unique* object is unpacked, we would have to modify Plural in order to achieve this.

There are also more advanced implementations of snapshotable trees [8], namely rebalancing - for which we would need to have partly *unique* and partly *immutable* permissions to the nodes in the tree. An important observation is that rebalancing involves only freshly allocated nodes in the path copy persistence implementation. Thus, we would need to carefully write the code such that Plural can derive this observation.

The node copy persistence implementation is more challenging: parts of a node are immutable while other parts are mutable. Here orthogonal dimensions of state, which are implemented in Plural, might become useful.

To conclude this paper, we successfully verified a snapshotable tree implementation and client code in Plural. In order to achieve that we had to rewrite parts of the reference implementation [10], mainly by adding explicit getter and setter methods, which is good object-oriented style.

An interesting method was `add`, which in the reference implementation calls `addRecursive`, which handles all cases at once: whether a snapshot is present (functional insertion) or no snapshots are present (mutating insert). In the higher-order separation logic proof this leads to three different specifications for `addRecursive`, one for each separate case. In automated tools (Plural and Dafny), it is easier to implement and verify two methods for those two cases, due to size of invariants and automated reasoning. Evidence for this is also provided by Rustan Leino, who implemented insertion in a clean-room setting from the beginning as two different methods. The reference implementation is clearly more compact, but it is arguable which implementation is clearer or more in the object-oriented spirit.

We modified the client code slightly by removing an additional temporary boolean variable, because we found that Plural's inference of boolean values works better this way. The original challenge used a boolean variable because their semantics does not allow for statements (heap access) in the loop condition, but only expressions (stack access).

While doing this proof we found several proof patterns for Plural: using implications instead of multiple typestates, inserting explicit return statements to help Plural with automation, writing explicit alternatives for conditionals, moving methods into the specific class that concerns them because static methods are not as well supported, avoiding choice conjuncts, and assigning null explicitly so that Plural can associate a bottom permission with the field. To get the proof through, we had to write the method `snapall`, which does not have any observable computational effect, but reassigns fields which were null to null.

We consider Plural to be a helpful static analysis tool which prevents runtime bugs: it issues an error when `add` is called on a snapshot or when a snapshot of a snapshot is taken.

One bug in Plural has been found (`while (lc == true)` leads to infinite recursion), which silently crashed Plural, making it appear that the code was proven. This has subsequently been fixed by the author of Plural.

Many thanks to Kevin Bierhoff for helping with specifications and best practices in Plural. The second author was funded by NSF grant CCF-1116907.

References

1. Baker, H.G.: "use-once" variables and linear objects: storage management, reflection and multi-threading. SIGPLAN Not. 30, 45–52 (January 1995)
2. Beckman, N.E., Kim, D., Aldrich, J.: An empirical study of object protocols in the wild. In: ECOOP'11 (2011)
3. Bierhoff, K.: Iterator specification with typestates. In: Proceedings of the 2006 conference on Specification and verification of component-based systems. pp. 79–82. SAVCBS '06 (2006)
4. Bierhoff, K.: Api protocol compliance in object-oriented software. Tech. Rep. CMU-ISR-09-108, CMU ISR SCS (2009)
5. Bierhoff, K.: Automated program verification made symplar. In: Proc of Onward! 2011 (2011)
6. Bierhoff, K., Aldrich, J.: Permissions to specify the composite design pattern. In: Proc of SAVCBS 2008 (2008)
7. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) Static Analysis: 10th International Symposium. Lecture Notes in Computer Science, vol. 2694, pp. 55–72. Springer, Berlin, Heidelberg, New York (2003)
8. Driscoll, J., Sarnak, N., Sleator, D., Tarjan, R.: Making data structures persistent. Journal of Computer and Systems Sciences 38(1), 86–124 (1989)
9. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. LNCS 6355. pp. 348–370 (2010)
10. Mehnert, H., Sieczkowski, F., Birkedal, L., Sestoft, P.: Formalized verification of snapshotable trees: Separation and sharing. In: VSTTE'12 (2012)
11. Nistor, L., Aldrich, J.: Verifying object-oriented code using object propositions. In: Proc of IWACO (2011)
12. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. In: IEEE Transactions on Software Engineering (1998)
13. Sunshine, J., Naden, K., Stork, S., Aldrich, J., Éric Tanter: First-class state change in plaid. In: OOPSLA'11 (2011)