Challenges of Verifying Persistent Data Structures*

Shengyu Huang

shengyu.huang@epfl.ch

June 14, 2023

Abstract

A partially persistent binary search tree is an extension of a binary search tree that supports lookups on previous versions. Such a conceptually simple data structure turns out to be difficult to verify due to its sophisticated specification and data sharing. Our project experience emphasizes three main topics: the insights into the data structure itself, how triggers affect our proofs, and how heap reasoning works in Dafny.

1 Introduction

A data structure is said to generate a new version when it is modified. A persistent data structure allows users to have access to previous versions and act on them. In contrast, *ephemeral* data structures do not allow us to retrieve the previous versions.

Driscoll et al. introduced several ways to make pointer-based data structures *partially persistent* and *fully persistent* [1]. A partially persistent data structure allows lookup on previous versions, while a fully persistent data structure allows additionally modification on previous versions. Direct and interesting applications of persistent data structures include version control systems [2, 3] and reverse debuggers [4]. Persistent data structures also play a central role in some functional programming languages like Clojure [5].

We used Dafny [6] to implement and verify *the fat node method* illustrated in [1] for partially persistent binary search trees. Although the implementation of the fat node method is rather simple and straightforward, we encountered a lot of unexpected challenges to verify just around 400 lines of Dafny code. Following the suggestions from Dafny manual [7, 8] sometimes worked, but in a lot of situations, it remains unclear how to resolve spurious timeouts[1] or unexpected `unknown`.

This project can be seen as two-stage.

In the first stage, we validated all pure functions. We first spent a lot of efforts in gaining insights of the data structure so that we can write a succinct specification. Readers can refer to Section 2.2 to compare the different versions of our specification. In Section 3, we will focus on a key assertion in our proof of `Find` to introduce the topic of triggers, which plays an important role throughout the whole project.

In the second stage, we attempted to validate the method `Insert`, but only part of method is verified in the end. We speculate the difficulty of verifying methods in this data structure resides in the complex data sharing. In Section 4, we will discuss how Dafny reasons about the heap and how our data structure poses challenges to it.

Although we only used Dafny in this project, our project experience may align with that of other similar projects, since many of the challenges we encountered are not specific to this data structure or the verification tool. Moreover, the challenges documented in this report will very likely be present in other kinds of persistent data structures implemented with methods from [1], since the fat node method for partial persistence is the simplest of them all.

Two natural questions arise from our project experience:

1. Can we avoid "bugs" from SMT solvers? When we write programs, we want to avoid bugs coming from the compiler level. When we write proofs, can we avoid "bugs" coming from the SMT solver level?
2. How can we more effectively debug our proofs?

In Section 5, we will present some code snapshots that indicate the instability of SMT solvers and use them to provide preliminary insights into the two questions above.

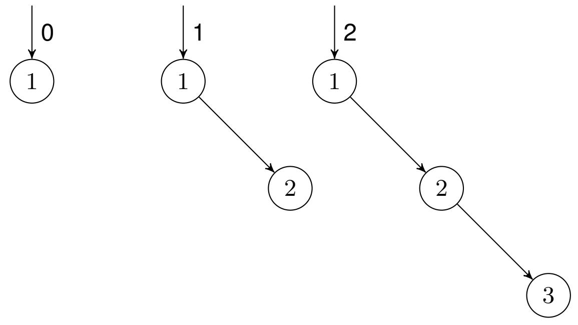
2 Partially Persistent Binary Search Trees

One common way to achieve persistence is to use *path copying method* as shown in Figure 1. The path copying method copies all the nodes it encounters along the path when it modifies the data structure. Since the path copying method does not mutate the data on the heap, it makes verification easy, and such a persistent AVL tree has been verified using Stainless [9] as a course project [10] as a joint work with Tomasz Stanislaw Marzec.

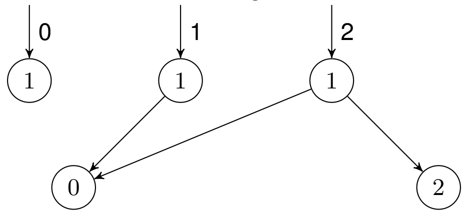
In this project, we will focus on binary search trees that handle only integer values and apply the fat node

*Supervised by Prof. Clément Pit-Claudel and Prof. Viktor Kunčak

[1]We set the timeout to be 5 minutes.



(a) Inserting 1, 2, 3.



(b) Inserting 1, 0, 2.

Figure 1: The path copying method after two different update sequences to a partially persistent binary search tree.

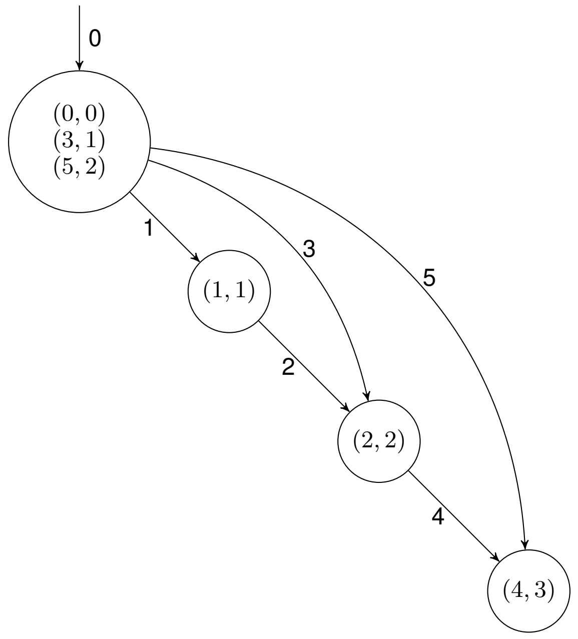


Figure 2: The fat node method after an update sequence `i0, i1, i2, d0, i3, d1` to a partially persistent binary search tree, where `ix` denotes inserting `x` and `dx` denotes deleting `x`. All arrows denote right pointers, and the first element of all tuples denotes versions. Interactive visualization available at https://kumom.io/persistent-bst/.

method [1] to achieve partial persistence, as shown in Figure 2.

The fat node method can be generally applied to other pointer-based data structures as well, and it induces a logarithmic blowup in time for each lookup operation and constant additional time for insertion and deletion. A better approach, called *the node copying method*, achieves amortized constant time for both lookup and modification operations, given that the original data structure has bounded in-degree for every node.

The biggest advantage of the fat node method and the node copying method over the path copying method is the space complexity. The path copying method induces no additional time for all operations. By using an additional array to store all root pointers with their corresponding versions, all operations can be performed the same way as for an ephemeral binary search tree. However, the space required by the path copying method would be costly if a node is big, because this approach would copy all the nodes it encounters along the way when doing modification. On the other hand, the fat node method only induces constant additional space and the node copying method also induces the same space complexity but in amortization.

2.1 The Fat Node Method

The high-level idea of the fat node method is to store updates inside the affected nodes only and allow the nodes to grow as "fat" as they can. From now on, we assume a minimal interface for our ephemeral binary search trees, i.e., a `value` field to store an integer and two pointers to access its left and right subtrees.

To support lookup on previous versions, we use lists to store updates together with their corresponding versions, with the lists of versions being sorted. For example, in Figure 2, the only root pointer indicates the topmost top has been the root since the initial version 0. The root stores `[(0, 0), (3, 1), (5, 2)]` to indicate this node is constructed at version 0 with the initial value 0. At version 3, its value got updated to 1 and at version 5 its value got updated to 2.

Lookup operation involves performing binary search on the sorted list of versions repeatedly and find the largest version that is smaller or equal to the query version, which we will call the *maxmin version* from now on.

If we want to query whether value 1 exists at version 2, we start at the root and find the maxmin version, 0 in our case, for the query version 2. At version 0, this node has value 0. We then proceed as we would in an ephemeral binary search tree, i.e., we look for value 1 in the right subtree, but with the correct responding version. The root has right pointers updated at version 1 and version 3, but we should traverse the pointer marked with version 1 since it is the maxmin version for query version 2. We repeat the same process until we find value 1 and return true in our case.

2.2 Specification

A fat node in our partially persistent binary search tree, denoted by `Node`, would have three sequences to store the values and pointers, three sequences to store the corresponding versions, and three ghost variables to help with specification.

```
class Node {
  ghost var Repr: set<Node>
  ghost var ValueSetsVersions: seq<int>
  ghost var ValueSets: seq<set<int>>
  var leftsVersions: seq<int>
  var lefts: seq<Node?>
  var rightsVersions: seq<int>
  var rights: seq<Node?>
  var valuesVersions: seq<int>
  var values: seq<int>

  ... // predicates, functions and methods omitted
}
```

The original algorithm for the fat node method [1] and our illustration in the previous subsection use a more compact representation, i.e., lists of tuples to store data with versions, but this straightforward implementation leads to slow verification based on our experiments. Therefore, we decouple the versions and their corresponding data and impose a same-length requirement for four pairs of sequences: `valuesVersions` and `values`, `leftsVersions` and `lefts`, `rightsVersions` and `rights`, `ValueSetsVersions` and `ValueSets`.

Since sequences in Dafny are an immutable value type, updating an sequence once has the same cost as the length of that sequence. To achieve the advertised complexity, we should use the mutable type `array` instead of `seq` in Dafny. However, for ease of verification, we started with this simpler setting with `seq`.

There are three ghost variables to help with specification.

`ValueSets` and `ValueSetsVersions` together allow us to infer the values that this partially persistent binary search tree contains at every version. For example, the root node in Figure 2 would have `ValueSets == [{0}, {0, 1}, {0, 1, 2}, {1, 2}, {1, 2, 3}, {2, 3}]` and `ValueSetsVersions == [0, 1, 2, 3, 4, 5]`.

`Repr` is a set of `Node` that makes up the representation of this data structure, including itself and *some of* the nodes `this` can reach through `lefts` and `rights` recursively. In Dafny, a function (or a method) needs to specify the set of previously allocated objects whose fields it might read (or modify). The `reads` and `modifies` clauses together indicate which parts of the memory the data structure will access. For unbounded recursive data structures like ours, defining a ghost variable like `Repr` allows us to use it as the frame expression for the `reads` and `modifies` clauses.[2]

Note that the version sequences might have "gaps". For example, `valuesVersions` in the root node in Figure 2 is `[0, 3, 5]`, but our invariants should still allow us to infer its value at version 1, 2, and 4. Formally, the value of a node at a version `v` is the same as its value at the maxmin version with `v` being the query version. We can infer the value set and the left/right pointer of a node at a certain version in a similar way.

`VersionIndex(queryVersion: int, versions: seq<int>) : (index: int)` is defined to return the index of the maxmin version in `versions` given the `queryVersion`. For example, `VersionIndex(4, [0, 3, 5])` should return 1, since 3 is the maxmin version of `[0, 3, 5]` given the query version 4.

Using `VersionIndex`, we define a series of helper functions including

- `ValueAt(version: int) : (res: int)`
- `LeftAt(version: int) : (res: Node?)`
- `RightAt(version: int) : (res: Node?)`
- `ValueSetAt(version: int) : (res: set<int>)`
- `RightValueSetAt(version: int) : (res: set<int>)`
- `LeftValueSetAt(version: int) : (res: set<int>)`

The helper functions above should be self-explanatory. For example, `RightAt(4)` called on the root node in Figure 2 would return the node with `(2, 2)`, and `RightValueSetAt(4)` called on the same node would return `{2, 3}`. Note that `ValueAt(version: int)` can only return a valid value if the query version is at least as big as the version when the node gets constructed, which is the same as `valuesVersions[0]`. Therefore, we require `version >= valuesVersions[0]` as a precondition of the function `ValueAt`.

Invariants of our data structure include

1. Same-length requirement between `values` and `valuesVersions`, `lefts` and `leftsVersions`, `rights` and `rightsVersions`, `ValueSets` and `ValueSetsVersions`.
2. All version sequences are sorted.
3. `Repr` of the root node should contain all the nodes owned by the data structure.
4. `ValueSets` and `ValueSetsVersion` should allow us to infer all the values that the induced tree contains at every version.
5. The induced tree at every version should preserve the property of an ephemeral binary search tree, i.e., all values in the right subtree of a node are greater than its own and all values in the left subtree are smaller than its own.

We spent about a month and a half iterating the specification because the last three invariants were initially written in a sophisticated way that lead to timeouts even when we tried to verify the simple helper functions. We omit the discussion of the first two invariants as they are easy to specify.

Our initial attempts to write down the specification imitate the style of the binary search tree example in Dafny's GitHub repository [12]. Specifically, we defined a predicate `Valid` that includes all the invariants above. In this following text, we will focus on the differences between

[2]A more detailed explanation of the `reads` and `modifies` clauses in Dafny can be found in [11].

our initial attempts and the current version of the specification.

The definition of `Repr` remains the same across different versions.

```
this in Repr
&& (forall l <- lefts | l != null ::
    l in Repr && this !in l.Repr && l.Repr < Repr && l.Valid())
&& (forall r <- rights | r != null ::
    r in Repr && this !in r.Repr && r.Repr < Repr && r.Valid())
&& (forall r <- rights, l <- lefts | r != null && l != null ::
    l != r && l.Repr !! r.Repr)
```

By recursively specifying `l.Valid()` and `r.Valid()`, we impose the invariants on all nodes in the data structure.

Note that this overapproximate definition of `Repr` implies `Repr` can include nodes that are not part of the data structure. We can make the definition more precise by adding

```
forall node <- Repr ::
  (node == this
  || (exists l | l in lefts && l != null :: node in l.Repr)
  || (exists r | r in rights && r != null :: node in r.Repr))
```

However, this additional formula leads to timeout in different versions of specification.

Allowing an overapproximation of `Repr` is acceptable during the verification of `Find`, but it can potentially complicate the verification of `Insert`. Therefore, we will defer our discussion about subtleties of `Repr` until Section 4.

Initial attempts for the invariant of `ValueSets` We want to specify the fourth invariant such that it allows us to infer the values that the induced tree contains at every version.

Denote the i-th element in `ValueSetsVersions` as v_i. `ValueSets[i]` should be a union of the value at v_i, the value set at v_i of the right node at v_i, and the value set at v_i of the left node at v_i. In our proof, the value set at v_i of the right node at v_i is written as `rights[l].ValueSetsVersions[ll]`, where `rightsVersions[l]` and `rights[l].ValueSetsVersions[ll]` are the maxmin versions of their corresponding version sequences given the query version `ValueSetsVersions[i]`.

Moreover, considering `rights` may be empty and `rights[l]` may be null, we have to adapt the invariant for different cases.

For example, when `lefts` is empty but `rights` is not, the invariant is expressed as

```
|rights| > 0 && |lefts| == 0 ==>
  (forall i | 0 <= i < |ValueSets| ::
    exists j, k | 0 <= j <= i < |values|
                  && 0 <= k <= i < |rights| ::
     MaxMin(ValueSetsVersions[i], j, valuesVersions)
     && MaxMin(ValueSetsVersions[i], k, rightsVersions)
     && (rightsVersions[k] == ValueSetsVersions[i]
        || valuesVersions[j] == ValueSetsVersions[i])
     && (rights[k] != null ==>
          (exists x | 0 <= x < |rights[k].ValueSets| ::
            MaxMin(ValueSetsVersions[i], x, rights[k].ValueSetsVersions)
            && ValueSets[i] == {values[j]} + rights[k].ValueSets[x]))
     && (rights[k] == null ==> ValueSets[i] == {values[j]}))
```

Considering all possible cases for this one invariant already induces 50 lines of formulas, where most of them contain three quantifiers.

Current specification for `ValueSets` The convoluted specification in our initial attempts is mostly attributed to the "gaps" in the version sequences and different cases we need to consider to preserve the well-formedness of formulas. However, we have already defined helper functions like `ValueAt(version: int)`, `LeftValueSetAt(version: int)`, and `RightValueSetAt(version: int)`, which will return the valid result at different versions. For example, `LeftValueSetAt(v)` should return {} for all versions `v` if `lefts` is empty. The fourth invariant hence can be written as

```
forall v | valuesVersions[0] <= v ::
  ValueSetAt(v) == { ValueAt(v) } +
                   LeftValueSetAt(v) +
                   RightValueSetAt(v)
```

Unfortunately, we cannot use the formula above in our `Valid`, because `Valid` is a precondition of all the involved functions, which will lead to infinite recursion.

To enable the aforementioned specification, one approach is to rewrite the `reads` clause of `RightValueSetAt` from `Repr` to `(set x | x in rights)`. As a result, `RightValueSetAt` no longer requires `Valid` as a precondition. Instead, we can explicitly specify its weakest precondition. However, this approach requires proving later that `Valid` implies the weakest precondition, since `Find` uses `Valid` as a precondition and calls these helper functions. Moreover, writing the weakest preconditions explicitly for each helper function can make the proofs less readable.

Due to these concerns, we fragmented `Valid` into three parts: `BasicProp`, `ValueSetProp`, and `BinarySearchProp`.

The `BasicProp` specify the first three invariants and would be used as the precondition of all the helper functions.

`ValueSetProp` and `BinarySearchProp` are defined as follows.

```
ghost predicate ValueSetProp()
  reads Repr
  requires BasicProp()
{
  (forall v | valuesVersions[0] <= v ::
    ValueSetAt(v) == { ValueAt(v) } +
                     LeftValueSetAt(v) + RightValueSetAt(v))
  && (forall v <- lefts | v != null :: v.ValueSetProp())
  && (forall v <- rights | v != null :: v.ValueSetProp())
}

ghost predicate BinarySearchProp()
  reads Repr
  requires BasicProp()
{
  (forall v | v >= valuesVersions[0] :: isBST(v))
  && (forall v <- lefts | v != null :: v.BinarySearchProp())
  && (forall v <- rights | v != null :: v.BinarySearchProp())
}
```

where the predicate `isBST` is defined as

```
ghost predicate isBST(version: int)
```

```
    reads Repr
    requires BasicProp()
  {
    if (version < valuesVersions[0]) then
      true
    else
      var v := ValueAt(version);
      (forall v' <- RightValueSetAt(version) ::
         v' > ValueAt(version))
   && (forall v' <- LeftValueSetAt(version) ::
         v' < ValueAt(version))
  }
```

Note that in `BinarySearchProp` (similarly for `ValueSetProp`), we did not write `forall v <- Repr :: v.BinarySearchProp()` and instead used

```
  (forall v <- lefts | v != null :: v.BinarySearchProp())
  && (forall v <- rights | v != null :: v.BinarySearchProp())
```

because `BasicProp` only imposes this property to hold on nodes that are reachable through `lefts` and `rights`. However, recall that `Repr` may contain nodes that are not owned by this data structure. These nodes may not satisfy `BasicProp`, which is the precondition to call `BinarySearchProp`.

3 Verifying `Find`

The interface of `Find` is simple and it can be verified under 500ms.

```
  function Find(version: int, value: int) : (res: bool)
    reads Repr
    requires BasicProp() && ValueSetProp() && BinarySearchProp()
    ensures res <==> value in ValueSetAt(version)
  {
    if (version < valuesVersions[0]) then
      false
    else
      if version < valuesVersions[0] then
        assert value !in ValueSetAt(version);
        false
      else
        assert isBST(version);
        ...
  }
```

There is a caveat though: omitting the line `assert isBST(version)` makes the solver return `unknown` under 1s with the message

> This postcondition might not hold: `res <==> value in ValueSetAt(version)`.

This comes to how Z3 handles quantifier instantiation. There are two main approaches to handle it in Z3: *E-matching* and *model-based quantifier instantiation* (MBQI). In the Dafny generated input file to Z3, we saw `(set-option :auto_config false)` and `(set-option :smt.mbqi false)` that would disable MBQI. Therefore, we will only focus on the E-matching process in the following text. We will briefly cover the basics of it using the materials mostly reproduced from [13], [14], [15], and [16].

3.1 E-matching and Triggers

A trigger is a set of non-ground terms, and the terms in a trigger need to cover all quantified variables in the input formula. A term is said to be *active* if the current partial model gives it an interpretation. If there exists a substitution σ such that $\sigma(t)$ is active in the partial model M for all terms in the trigger, we say the trigger *matches* in M.

The E-matching process looks for ground terms matching the triggers to determine when to perform quantifier instantiation. This implies E-matching cannot prove even simple properties when no ground terms are available.

Suppose our proof goal is the following formula, where the trigger is denoted as $\{f(g(x))\}$.

$$\forall x.\{f(g(x))\}f(g(x)) \neq x$$

We have additionally three (in)equality assertions $g(a) = c$, $g(b) = c$, and $a \neq b$.

Since there is no active ground term of the form $f(g(t))$, the quantifier is not instantiated. Z3 therefore fails to show that the formula is unsatisfiable and returns `unknown`. If a more permissive trigger $\{g(x)\}$ is used, Z3 will return `unsat`.

Some trigger can create infinite instantiation chains and lead to what we call *matching loops*. For example, in

$$\forall x.\{f(x)\}f(x) = f(g(x)),$$

the quantifier will get instantiated whenever some term of the form $f(t)$ is active, and the instantiation will bring a fresh ground term $f(g(t))$, which causes another instantiation that brings $f(g(g(t)))$, and so on. However, if we have a more restrictive trigger $\{f(g(x))\}$, we can avoid the matching loop. In some cases, matching loops can be hard to resolve, e.g., $\forall x.f(x) = g(f(x)) \wedge g(x) = f(g(x))$.

These examples show that selecting triggers can be quite tricky. Dafny implements its own trigger selection algorithm [14] in the hope of more predictable behaviors, and users can also specify the triggers explicitly. However, this does not eliminate all potential problems arising from triggers. The subsection "Existential Activation" in the paper by Michał Moskal [13] gives a great example, where Z3 introduces a new skolem constant when reasoning about the heap. Since the skolem constant is freshly generated at the SMT solver level, it is difficult for users to specify the trigger at the verifier level. To resolve this issue, they introduced a pattern `ex_act` by hacking Z3 that enforces the annotated terms to be activated when the formula undergoes skolemization. The subsection "Existential Activation" is concluded with a brief remark:

> This pattern was crucial in verification of recursive data structures in VCC.

It is unclear how well the state-of-the-art SMT solvers can handle such situations, given that the paper [13] is published in 2009. Curiously, we would like to know whether we encountered the same situation when we failed to verify `Insert`, but the debugging tools used in [13] are no longer maintained, and we did not have the capacity to debug our proofs at a lower level.

3.2 Unexpected failure due to inactive term `isBST` and unknown reasons

If we hover above the quantifiers of the formulas in `BinarySearchProp`, we can see {`isBST(v)`} is selected as the trigger for `forall v | v >= valuesVersions[0] :: isBST(v)`. Therefore, one must have `isBST(x)` as an active term in the partial model in order to deduce useful information related to `BinarySearchProp`.

However, it remains unclear why this trigger causes failure to prove `res <==> value in ValueSetAt(version)`, since `ValueSetAt(version)` does not depend on `isBST` in anyway. Even after removing `BinarySearchProp()` from the precondition of `Find`, which means nothing inside `Find` directly or indirectly depends on `isBST`, the proof still fails with the same message very quickly. We suspect adding `assert isBST(version)` not only triggers the related formula inside `BinarySearchProp`, but also changes other heuristics employed that lead to the successful verification in the end. For this reason, we inlined `isBST` inside `BinarySearchProp` and assumed this from now on.

4 Verifying `Insert`

`Insert` is the first and only method we attempted to verify, and yet we have only managed to verify a small part of it. In this section, we will talk about how we verified a simple branch of `Insert`. To conclude this section, we will provide our speculation on why our approach did not work on other branches that involve recursive calls.

Here is the specification and the verified part of `Insert`.

```
method Insert(version: int, value: int) returns (res: Node?)
  modifies Repr
  decreases Repr
  requires BasicProp() && BinarySearchProp() && ValueSetProp()
  requires version > ValueSetsVersions[|ValueSetsVersions| - 1]
  ensures fresh(res)
  ensures BasicProp() && BinarySearchProp() && ValueSetProp()
  ensures value in ValueSetAt(version)
{
  var x := Value();
  var right := Right();
  var left := Left();
  ghost var vs := ValueSet();
  if x > value {
    assume false;
  } else if x < value {
    if right == null {
      res := new Node.Init(version, value);
      rights := rights + [res];
      rightsVersions := rightsVersions + [version];
      Repr := Repr + res.Repr;
      ValueSets := ValueSets + [vs + {value}];
      ValueSetsVersions := ValueSetsVersions + [version];

      OrderInvariant(old(rightsVersions),
                     old(ValueSetsVersions), version);
      assert fresh(res);
      assert BasicProp();
      InsertRight(res, version, value);
    } else {
      assume false;
    }
  } else {
    assume false;
  }
}
```

By putting `assume false` in all branches except one, we only need to focus on one single branch where the induced tree at the latest version has an empty right subtree and its value is smaller than the to-be-inserted value.

We access the induced tree at the latest version by simply accessing the last entries in the corresponding sequences. `Left()`, `Right()`, `Value()`, and `ValueSet()` returns the last entry in the sequences `lefts`, `rights`, `values`, and `ValueSets`, respectively.

Although this simplified version of `Insert` only contains six lines of code that involves changes to the fields and does not contain any recursive calls, verifying it takes more than 20s. More precisely, the lemma `InsertRight` takes more than 20s to be verified, and with the help of it, the simplified `Insert` is verified under 3s.

Since the nodes do not have a global view of the whole data structure, we need to pass the parameter `version` to `Insert` explicitly. We can expose a cleaner interface to clients by building another class that holds all the root pointers so that it can infer the latest version.

For a given node, it is only valid to call `Insert` if the passed `version` is bigger than all the versions in its version sequences and the ones of its children. For example, if the largest version in `ValueSetsVersion` is 3, calling `Insert` with a version less than 3 is obviously not valid. Observe that all the version sequences of the children of a node are subsequences of its `ValueSetsVersions`. Therefore, we can enforce this requirement with `version > ValueSetsVersions[|ValueSetsVersions| - 1]`.

The lemma `OrderInvariant(subVersions: seq<int>, versions: seq<int>, v: int)` is used in our proof. This lemma says that if version `v` is greater than all the entries in the sorted `versions`, and `subVersions` is a subsequence of `versions`, then `v` is also greater than all the entries in `subVersions`. Consequently, appending `v` to `versions` or `subVersions` will keep the new sequences sorted. Here, we use `OrderInvariant` to prove the new `rightsVersions` will remain sorted.

4.1 Verifying the simplified `Insert`

Before talking about the key lemma `InsertRight`, we will first discuss how to deal with dynamic frames in Dafny.

Dafny does not automatically infer which part of the heap a method modifies, because this sort of analysis is in general undecidable and it also makes modular verification impractical when aliasing is in place. Instead, Dafny only looks at the `modifies` clause to infer what data on the heap may change after a method returns. To make it more precise, we can use `old` and `unchanged` in the postconditions. In the preconditions of `InsertRight`, we use old and unchanged, which is the same as proving them as the postconditions of the simplified Insert.

Currently, the modifies clause of Insert is Repr. We can also be more precise by using conditional expressions, i.e.,

```
if value > Value() && Right() != null then
  {this} + Right().Repr
else if value < Value() && Left() != null then
  {this} + Left().Repr
else {this}
```

That is when we modify the induced right subtree, fields of objects inside Left().Repr will not change.

For the simplified Insert, it is enough to have modifies this because no recursive calls to Insert are made. For the same reason, the modifies clause should not affect the complexity of our proof in this simplified Insert. The modifies clause only affects in a way that Dafny will check we do not attempt to modify fields of objects that are not included in the modifies frame.

InsertRight The key lemma InsertRight, which is a *twostate lemma*, involves heavy uses of old and unchanged. This seems to suggest that the complexity is attributed to reasoning about the heap.

Syntactically speaking, a twostate lemma allows us to use old or unchanged in its body and preconditions. Under the hood, a twostate lemma has two implicit heap parameters, whereas normal functions only take one implicit heap parameter. Therefore, we can reason about the values of a field in the old heap and the modified heap in a twostate lemma.

Since the proofs of some branches should be symmetric (e.g., when x < value and x > value), using twostate lemmas can remove repetitive proofs and make the verification faster. In this simplified version, we can inline the twostate lemma InsertRight since only one branch is involved, but having a separate lemma makes both the code and the proof more readable.

Preconditions of InsertRight include, for example, rightsVersions == old(rightsVersions) + [version], values == old(values), and

```
forall node <- Repr | node != this && old(allocated(node)) ::
  unchanged(node)
```

that allow us to reason how the heap is modified precisely.

The universally quantified formula above says all the nodes except this that are previously allocated in Repr are not modified. We cannot replace this formula with unchanged(Repr - {this}), because unchanged or old can only reason about objects that have been allocated, and Repr may contain nodes that are not owned by our data structure, and hence it can also contain nodes that have not been allocated.

Moreover, we mentioned in Section 2.2 that Repr includes itself and only *some of* the nodes this can reach through lefts and rights recursively. In other words, some of the nodes that are reachable through lefts or rights are not included in Repr. Consider adding 3 to Figure 2. Since we will only traverse the induced tree at the latest version, we will not encounter the node with (1, 1) and update its Repr. Therefore, the Repr of the node with (1, 1) would not include the newly inserted node. However, this node can reach the newly inserted node through its right pointers recursively.

Making Repr include itself and *all* the nodes it can reach recursively may bring other problems. In the above example, we need to first be able to infer the node with (1, 1) can reach the newly inserted code. Even if it is feasible, it potentially requires us the traverse the whole tree, and the sophisticated recursive calls could make the verification harder to succeed.

To conclude this section, we would like to provide two additional insights into the data structure that may pose challenges for verification.

First, the BinarySearchProp is "version-bounded". To illustrate this, consider Figure 2. The root node holds (5, 2), while a node reached through its right pointer at version 1 holds (1, 1). If we omit the versions, the induced tree has value 1 in the right subtree of a node with value 2. This situation will not appear in ephemeral binary search trees.

Second, we do not have nice separation in rights sequences (similarly for lefts). Precisely, the following formula does not hold in our data structure, and Figure 2 again gives a counterexample.

```
forall r1 <- rights, r2 <- rights |
  r1 != null && r2 != null && r1 != r2 :: r1.Repr !! r2.Repr
```

5 Literature Review: Debugging Proofs

More than two thirds of our time was spent on battling with spurious timeouts and unknown. For example, in one of our code snapshots [17], named as *Example 5*[3], we have three consecutive assertions written inside the body of ValueAt: assert P; assert P ==> Q; assert Q, where P and Q denote two formulas that involve calling another function ValueSetsAt. While the function with only the first two assertions can be verified under 500ms, once we introduce the third one, the proof times out.

Following the suggestions from the manual [7], we made all other functions opaque and the timeout was resolved. However, it remains unclear why making other functions opaque could have such a big impact on proving a seemingly trivial formula.

In another example, named as *Example 4*, we have Valid as the precondition of ValueSetsVersions, and we assert part of Valid as its postcondition. Since functions don't mutate anything, this postcondition should hold trivially. However, the proof simply fails under 500ms saying the postcondition may not hold. Such behaviors occur usually when a term that should be activated is not, but even explicitly adding all possible triggers using {:trigger}, the same behavior persisted.

[3]The examples presented here only contain a small fraction of the code snapshots we have saved. Interested readers are encouraged to explore the codebase [17].

When we upgraded Dafny to v4.0.0, which uses Z3 4.12.1 by default, this error disappeared.

All the peculiar examples we have gathered suggest the "bugs" one can encounter when writing formal proofs can be quite surprising and hard to resolve. Can we do better and write proofs more efficiently and robustly?

Can we avoid "bugs" from SMT solvers? Many of the problems we ask SMT solvers to solve are undecidable [18] and/or intractable [19, 20, 21]. Despite this, SMT solvers demonstrate remarkable efficiency in solving a wide range of problems due to the utilization of sophisticated heuristics. However, a rigorous understanding of these heuristics remains lacking. Some heuristics, like the ones involving trigger selection, are deemed to contribute to the instability of SMT solvers [13, 14]. Recent findings have uncovered regression bugs in certain mainstream SMT solvers [22, 23]. That being said, during our literature review, we found no universally agreed-upon definition of what constitutes a "bug" in SMT solvers.

The nature of the problems we want to solve, combined with the complex heuristics of SMT solvers, makes it challenging to comprehend how our proofs impact verification time. Naively, one might assume that more assertions, especially sophisticated ones, would result in longer verification times. However, this is not always the case. Additional intermediate assertions can actually narrow the search space and expedite the verification process. Furthermore, for seemingly simple propositional formulas, the refutation proofs can grow exponentially long [24].

How to debug our proofs more efficiently? Directly inspecting the input file to the SMT solver is impractical. Even a simple one-line function in Dafny, without any quantifiers, can generate an SMT file of 800 lines.

Debugging at the intermediate language level, specifically inspecting the Boogie file in the context of Dafny, would be more manageable once the suggested improvements in [25] are implemented.

In this report, we have frequently referenced the paper by Michał Moskal [13]. This paper not only aligns with our project experience but also introduces several promising tools for debugging proofs at the level of SMT solvers. These tools include model viewers for debugging the models, axiom profilers for tracing profiles, and the Z3 inspector for sampling profiles. Unfortunately, these tools were no longer maintained when we started our project.

6 Conclusion

Verifying a partially persistent binary search tree implemented using the fat node method turns out to be quite challenging. To simplify the specification, we need insights into the data structure itself. Additionally, understanding triggers and heap reasoning is crucial for verifying `Find` and `Insert`.

Ideally, users of program verifiers would be able to write formal proofs in a pen-and-paper style. However, achieving this goal requires addressing some fundamental questions like which formulas lead to lengthy proofs and which types of data structures result in complex heap reasoning. By leveraging more effective proof debugging tools, we hope to gain a deeper understanding of these questions.

References

[1] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," in *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pp. 109–121, 1986.

[2] E. D. Demaine, S. Langerman, and E. Price, "Confluently persistent tries for efficient version control," *Algorithmica*, vol. 57, pp. 462–483, 2010.

[3] B. Farinier, T. Gazagnaire, and A. Madhavapeddy, "Mergeable persistent data structures," in *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, 2015.

[4] J. Engblom, "A review of reverse debugging," in *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pp. 1–6, IEEE, 2012.

[5] R. Hickey, "A history of clojure," *Proceedings of the ACM on programming languages*, vol. 4, no. HOPL, pp. 1–46, 2020.

[6] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers 16*, pp. 348–370, Springer, 2010.

[7] "Dafny reference manual: 13.6.2. verification debugging when verification is slow." https://dafny.org/dafny/DafnyRef/DafnyRef#sec-verification-debugging-slow. Accessed: 2023-05-30.

[8] "Dafny verification optimization." https://dafny.org/dafny/VerificationOptimization/VerificationOptimization. Accessed: 2023-05-30.

[9] V. Kuncak and J. Hamza, "Stainless verification system tutorial," in *2021 Formal Methods in Computer Aided Design (FMCAD)*, pp. 2–7, IEEE, 2021.

[10] "Cs550 - persistent avl tree." https://github.com/kumom/cs550-persistent-avl-tree. Accessed: 2023-06-05.

[11] "An intricate look at the dafny reads and modifies concepts?." https://github.com/dafny-lang/dafny/discussions/1899. Accessed: 2023-06-09.

[12] "Github: dafny/test/dafny1/binarytree.dfy." https://github.com/dafny-lang/dafny/blob/master/Test/dafny1/BinaryTree.dfy. Accessed: 2023-06-07.

[13] M. Moskal, "Programming with triggers," in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pp. 20–29, 2009.

[14] K. R. M. Leino and C. Pit-Claudel, "Trigger selection strategies to stabilize program verifiers," in *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I 28*, pp. 361–381, Springer, 2016.

[15] "Z3 internals (draft)." https://z3prover.github.io/papers/z3internals.html#sec-e-matching-mourab07. Accessed: 2023-06-08.

[16] "Getting started with z3: A guide." https://www.philipzucker.com/z3-rise4fun/guide.html. Accessed: 2023-06-08.

[17] "Project repository." https://github.com/kumom/dafny-fatnode. Accessed: 2023-06-09.

[18] B. C. Pierce, "Bounded quantification is undecidable," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 305–315, 1992.

[19] A. Atserias and M. Müller, "Automating resolution is np-hard," *Journal of the ACM (JACM)*, vol. 67, no. 5, pp. 1–17, 2020.

[20] M. Göös, J. Nordström, T. Pitassi, R. Robere, S. F. de Rezende, and D. Sokolov, "Automating algebraic proof systems is np-hard," in *Electronic colloquium on computational complexity*, vol. 27, 2020.

[21] M. Göös, S. Koroth, I. Mertz, and T. Pitassi, "Automating cutting planes is np-hard," in *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pp. 68–77, 2020.

[22] M. Bringolf, D. Winterer, and Z. Su, "Finding and understanding incompleteness bugs in smt solvers," in *37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–10, 2022.

[23] "Regression with lambdas: trivial formula now unknown." https://github.com/Z3Prover/z3/issues/5516. Accessed: 2023-05-24.

[24] M. Mikša and J. Nordström, "Long proofs of (seemingly) simple formulas," in *Theory and Applications of Satisfiability Testing–SAT 2014: 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings 17*, pp. 121–137, Springer, 2014.

[25] "How does the dafny programming language compile-time check its constraints?." https://github.com/boogie-org/boogie/issues/575. Accessed: 2023-06-09.