Challenges of Verifying Persistent Data Structures

Shengyu Huang

EPFL Semester Project
Advisors: Prof. Clément Pit-Claudel and Prof. Viktor Kunčak

July 3, 2023

Roadmap

- Partially persistent trees
  - Path copying (done in course CS-550)
  - Fat node (with interactive visualization)
  - Node copying (not covered)
- Specification
  - Initial attempt
  - A more SMT-friendly version

Roadmap

- Partially persistent trees
  - Path copying (done in course CS-550)
  - Fat node (with interactive visualization)
  - Node copying (not covered)
- Specification
  - Initial attempt
  - A more SMT-friendly version
- Verifying `Find`
  - E-matching and triggers
  - Unexpected failure due to inactive term
- Verifying `Insert`
  - Verifying a simplified version
  - Heap reasoning in Dafny
- Discussion: debugging proofs

Persistent data structures

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, we cannot retrieve previous versions in **ephemeral data structures**.

We can perform lookup operations on **partially persistent** data structures, while **fully persistent** data structures allow us to **modify** on them (and hence the history of versions will not be linear any more).

Persistent search trees

We focus on tree structures for now. There are three ways to implement a persistent search tree.

- Path copying
- Fat node method
- Node copying (constant amortized time/space complexity for lookup and modification, not covered)

Path copying

The idea is to copy every node it encounters when modification is performed. Essentially, this approach induces an *immutable* data structure.

Path copying

The idea is to copy every node it encounters when modification is performed. Essentially, this approach induces an *immutable* data structure.

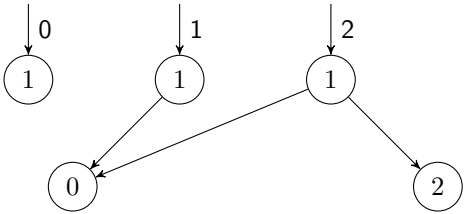


Figure: Inserting 1, 0, 2.

Path copying

Costly when the path we traverse to perform modification is long.

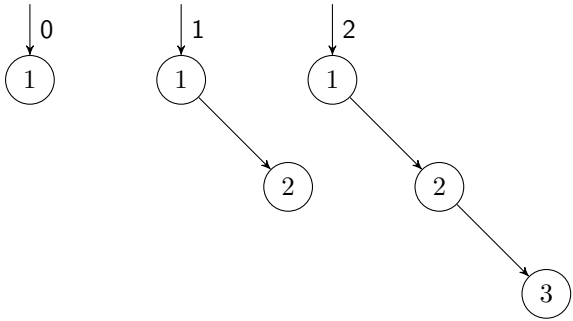


Figure: Inserting 1, 2, 3.

Fat node method

A better idea is to store updates only in the relevant nodes.

Demo: `https://kumom.io/persistent-bst`

High-level specification

We introduce a *ghost variable* `ValueSets` that allows us to infer all the values the tree contains for each version.

E.g., `3 in ValueSets[5]` if and only if value 3 exists in the fifth version of the tree.

High-level specification

We introduce a *ghost variable* `ValueSets` that allows us to infer all the values the tree contains for each version.

E.g., `3 in ValueSets[5]` if and only if value 3 exists in the fifth version of the tree.

However, since `ValueSets` is a field variable, we don't want to update this field for all the nodes every time the data structure gets updated. This means `ValueSets` only contains a subsequence of `[n]`, where `n` is the latest version we are at.

High-level specification

Suppose `ValueSets` only gets updated at versions `[1, 3, 5]`. To query the value at version 2, we perform binary search on the list of versions and return the version that is closest to but smaller than our query version. We call the returned version **maxmin version**.

High-level specification

Suppose `ValueSets` only gets updated at versions `[1, 3, 5]`. To query the value at version 2, we perform binary search on the list of versions and return the version that is closest to but smaller than our query version. We call the returned version **maxmin version**.

Here, 3 is the maxmin version if our query version is 2.

(Partial) specification for `Find`, `Insert`, and `Remove`

By writing a helper function `ValueSetAt(version:int):(res:set<int>)` that returns the correct value set for all possible query versions., we can loosely specify the key postconditions of our interface as follows.

```
function Find(version:int, value:int): (res: bool)
  ...
  ensures res <==> value in ValueSetAt(version)

method Insert(version:int, value:int): (res: Node?)
  ...
  ensures value in ValueSetAt(version)

method Remove(version:int, value:int): (res: Node)
  ...
  ensures value !in ValueSetAt(version)
```

Initial attempt to specify the invariant for `ValueSets`

We initially tried to mimic the specification style of `BinarySearch.dfy` in the Dafny repo.

```
ghost predicate Valid()
  reads this, Repr
  ensures Valid() ==> this in Repr
{
  ...
  (left == null && right == null ==>
    Contents == {data}) &&
  (left != null && right == null ==>
    Contents == left.Contents + {data}) &&
  (left == null && right != null ==>
    Contents == {data} + right.Contents) &&
  (left != null && right != null ==>
    left.Repr !! right.Repr &&
    Contents == left.Contents + {data} + right.Contents)
}
```

Initial attempt to specify the invariant for ValueSets

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]}))
```

Initial attempt to specify the invariant for ValueSets

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 the invariant to specify `ValueSets` leads to 50 lines of formulas, where most of them contain three quantifiers. $\Longrightarrow$ Timeout! (set to be 5 minutes for this project)

Using helper functions to specify

The convoluted specification is mostly due to 1) the "gap" in the version sequences and 2) different cases we need to consider to preserve the well-formedness of formulas.

Using helper functions to specify

The convoluted specification is mostly due to 1) the "gap" in the version sequences and 2) different cases we need to consider to preserve the well-formedness of formulas.

We can instead use helper functions to hide this complexity in our specification.

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

Using helper functions to specify

The convoluted specification is mostly due to 1) the "gap" in the version sequences and 2) different cases we need to consider to preserve the well-formedness of formulas.

We can instead use helper functions to hide this complexity in our specification.

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

Note: we need to rewrite the preconditions of these helper functions in order to avoid infinite loops.

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);
      ...
}
```

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);
      ...
}
```

Omitting the line `assert isBST(version)` makes the solver return `unknown` under 1s.

Quantifier instantiation and E-matching

Z3 handles quantifier instantiation using **E-matching** or **model-based quantifier instantiation** (MBQI). The default setting of Dafny uses E-matching only.

Quantifier instantiation and E-matching

Z3 handles quantifier instantiation using **E-matching** or **model-based quantifier instantiation** (MBQI). The default setting of Dafny uses E-matching only.

A **trigger** is a set of non-ground terms. Terms in a trigger need to cover all quantified variables in the input formula. E-matching looks for ground terms matching the triggers to determine when to perform quantifier instantiation.
A term is said to be **active** if the current partial model gives it an interpretation.

How triggers work - Example 1

Suppose our proof goal is $\forall x.\{f(g(x))\}f(g(x)) \neq x$. Additionally, we have $g(a) = c$, $g(b) = c$, and $a \neq b$.

How triggers work - Example 1

Suppose our proof goal is $\forall x.\{f(g(x))\}f(g(x)) \neq x$. Additionally, we have $g(a) = c$, $g(b) = c$, and $a \neq b$.

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

How triggers work - Example 2

Suppose our proof goal is $\forall x.\{f(x)\}f(x) = f(g(x))$.

How triggers work - Example 2

Suppose our proof goal is $\forall x.\{f(x)\}f(x) = f(g(x))$.

The quantifier gets instantiated whenever a term of the form $f(t)$ is active, and such an instantiation will bring a fresh ground term $f(g(t))$, which causes another instantiation...etc. In this case, if we have a more restrictive trigger $\{f(g(x))\}$, we can avoid the so-called **matching loops**.

How triggers work - Example 3

As the examples have shown, in some cases, a more permissive trigger is preferred, while in other cases, a more restrive trigger is needed.

How triggers work - Example 3

As the examples have shown, in some cases, a more permissive trigger is preferred, while in other cases, a more restrive trigger is needed.

That does not mean we can always resolve problems stemming from triggers. For example, $\forall x. f(x) = g(f(x)) \wedge g(x) = f(g(x))$.

Triggers in Dafny

Heuristics for selecting triggers in Z3 is deemed as unstable in a lot of literature. Dafny implements its own trigger selection algorithm in hope of more predictable behaviors, and users can also specify the triggers explicitly at the Dafny level.

Triggers in Dafny

Heuristics for selecting triggers in Z3 is deemed as unstable in a lot of literature. Dafny implements its own trigger selection algorithm in hope of more predictable behaviors, and users can also specify the triggers explicitly at the Dafny level.

Unfortunately, it does not resolve all possible issues from triggers. New skolem constants may be introduced at the SMT solver level and users cannot specify the triggers explicitly at the verifier level.

The key assertion in `Find`

```
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);
      ...
}
```

The `assert isBST(version)` in our proof activates the trigger to allow quantifier instantiation.

The simplified Insert

```
method Insert(version: int, value: int) returns (res: Node?)
  ...
{
  var x := Value();
  ghost var vs := ValueSet();
  if x < value && 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];
    // our proof
  } else {
    assume false;
  }
}
```

Proof for simplified Insert

Our proof in the Insert contains only four lines.

```
OrderInvariant(old(rightsVersions), old(ValueSetsVersions), version);
assert fresh(res);
assert BasicProp();
InsertRight(res, version, value);
```

The lemma InsertRight(res, version, value) contains about 100 lines and gets verified around 28 seconds.

Proof for simplified `Insert`

Our proof in the `Insert` contains only four lines.

```
OrderInvariant(old(rightsVersions), old(ValueSetsVersions), version);
assert fresh(res);
assert BasicProp();
InsertRight(res, version, value);
```

The lemma `InsertRight(res, version, value)` contains about 100 lines and gets verified around 28 seconds.

Conceptually, this lemma proves all the invariants that hold still hold after calling `Insert`. The proof itself is mostly of the form `old(some data on the heap) == some data on the heap`. In addition, a key precondition for this lemma is

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

The proof itself seems to suggest that the complexity is mainly attributed to the heap reasoning.

Repr for recursive unbounded data structure

In `Find`, `Insert`, and `Remove`, we have recursive calls to the current function/method. To prove termination, we introduce another ghost variable `Repr:set<Node>` that captures the heap we will read in functions or modify in methods.

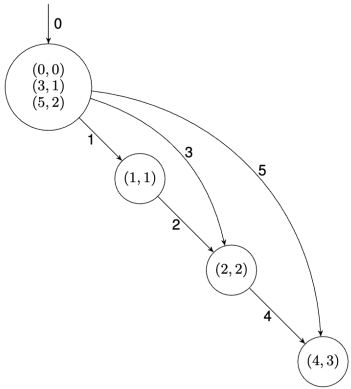
Repr for recursive unbounded data structure

In Find, Insert, and Remove, we have recursive calls to the current function/method. To prove termination, we introduce another ghost variable Repr:set<Node> that captures the heap we will read in functions or modify in methods.

```
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)
```

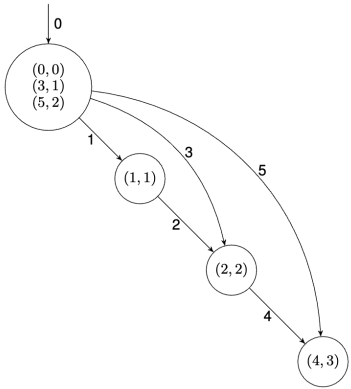
Why heap reasoning is complicated

- The frame captured by `Repr` cannot be inferred simply with pointers on the heap.



Why heap reasoning is complicated

- The frame captured by `Repr` cannot be inferred simply with pointers on the heap.
- The property of binary search is "version-bounded".



Understanding why formal proofs fail

We collected various examples that demonstrate surprising behaviors of the prover/verifier. Unfortunately, we don't know how to explain these behaviors.

Understanding why formal proofs fail

We collected various examples that demonstrate surprising behaviors of the prover/verifier. Unfortunately, we don't know how to explain these behaviors.

For example,

```
assert ValueSetsAtVersion(version).0 >= 0;
assert ValueSetsAtVersion(version).0 >= 0 ==>
  valuesVersions[i] <= ValueSetsAtVersion(version).0;
```

These two assertions are part of our proof for a function and can be verified under 500 ms. However, Dafny times out when we add

```
assert valuesVersions[i]<=ValueSetsAtVersion(version).0;
```

right after these two assertions.

Understanding why formal proofs fail

- We are dealing with undecidable and/or intractable problems.
- There is a lack of rigorous analysis of some important heuristics implemented in Z3 (e.g., triggers selection).
- Proof complexity itself is an active research area (what formulas lead to long refutation proofs?).

Theoretical constraints aside, can we write formal proofs more efficiently with better engineering?

Proofs debugging

No widely agreed-upon definition of "bugs" in SMT solvers, but in the VCC project, they have

- Model viewers for debugging the models
- Axiom profilers for tracing profiles
- Z3 inspector for sampling profiles

Unfortunately, all these tools are no longer maintained, but insights into why we keep getting timeouts are difficult to attain without effective debugging tools.

Q&A and Discussion

- Is there a better way to write the specification for this data structure?
- To what extend can we make formal proofs more or less like pen-and-paper proofs?
- Why is heap reasoning difficult?
- Is there a better way to debug formal proofs?