Formal Verification of Pointer-Based Splay Trees in Iris

RICARDO GRAÇA, Instituto Superior Técnico, Portugal

When real-world applications crash or start to lack in performance, they can bring tremendous costs to the involving parties. Therefore, it is important to ensure that these applications do not fail. Testing is useful in practice as it can be used to show the presence of bugs. However, it cannot be used to prove their absence. On the other hand, formal verification can be used to prove that a program fully meets a given specification. However, formal verification of real-world code, which normally manipulates mutable and non-trivial data structures, is a difficult task. In the last few years, many advances have been made in formal verification, but there are still many opportunities to verify real-world code. In this project, we explore Coq and the Iris framework to verify the functional correctness of the pointer-based implementation of Splay Trees which is used by the GNU Compiler Collection (GCC) in the Offloading and Multi Processing Runtime Library (libgomp). In the process, we also verify a functional implementation of the splay tree algorithm for a generalized ordered datatype using the interactive proof assistant Coq. To the best of our knowledge, we provide the most complete formally verified pointer-based implementation of Splay Trees.

Additional Key Words and Phrases: Formal Verification; Coq; Iris Framework; Splay tree; Mutable Data Structures; Heap-lang; GNU Compiler Collection; Libgomp

1 INTRODUCTION

Our world is now largely dependent on software systems running as expected. When software fails, even if it is just for a mere few seconds, consequences can bring tremendous costs. Therefore, it is crucial to ensure that software does not fail. Software testing is perhaps the most used technique to prevent software failures. However, even though testing can be used to show the presence of bugs, it cannot be used to prove their absence. On the other hand, formal verification can be used to prove a program fully meets a given specification. However, formal verification of real-world code, which normally manipulates mutable and non-trivial data structures, is a difficult task. In the last few years, many advances have been made in formal verification, but there are still many opportunities to verify real-world code.

In this project, we propose to explore the interactive proof assistant Coq [coq [n. d.]] and the Iris framework [Jung et al. 2018] to verify the functional correctness of the pointer-based implementation of Splay Trees which is used by the GNU Compiler Collection (GCC) in the Offloading and Multi Processing Runtime Library (libgomp).

We chose Splay Trees for two main reasons: i) because they have become a widely-used data structure for being the fastest type of balanced search tree for many applications; ii) and because, due to their self-balancing properties, their formal verification presents an interesting challenge.

1.1 Work Objectives

In this project, we will first verify a functional implementation (Nipkow's) of the splay tree method and then verify the correctness of a pointer-based implementation of the splay tree algorithm from a well-known application, GCC, which uses it in its Offloading and Multi Professing Runtime Library. The four work objectives that we have proposed for this project are well-enumerated:

Author's address: Ricardo Graça, Departamento de Engenharia Informática, Instituto Superior Técnico, Av. Rovisco Pais 1, Lisboa, 1049-001, Portugal, ricardo.ciriaco@tecnico.ulisboa.pt.

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

1. *verify a functional implementation of the splay tree algorithm for a generalized ordered datatype in, the theorem proof assistant, Coq.* ***2.*** *translate GCC's pointer-based implementation written in the C language to a language that allow us to reason in Iris' framework, heap-lang.* ***3.*** *create a predicate for the splay tree that holds the predicates and invariants for a binary search tree and for the memory that is modeled with a generalized map.* ***4.*** *specify and verify the correctness of the lemmas related to the splay tree methods.*

1.2 Contributions

During the execution of the mentioned objectives, we have managed to successfully verify in Coq the Gallina translation of Nipkow's implementation [Nipkow 2014]. We have also proved some other lemmas and theorems that Nipkow has not and also successfully extracted the verified code, for natural numbers, to the OCaml functional language.

For the verification of the pointer-based implementation, we started by successfully translate the GCC implementation of splay trees to heap-lang. Then, we have also succeeded in modeling the splay tree predicate which contains all the needed invariants for a binary search tree, as well as the modeled memory and the ownership of all the pointers of the binary search tree. Afterwards, we have proven important lemmas related to each of the predicates that make the splay tree predicate.

At last, we have successfully proven the correctness of the splay tree method, but leaved one of the lemmas, that makes this proof possible, unproven, which is not desirable, however, we have informally proven it and are pretty confident that in the future we will be able to prove it.

2 BACKGROUND AND RELATED WORK

In this section we first present the practical uses of the splay tree algorithm and an application that uses it. Then we talk briefly on self-adjusting structures and the potential function that is used to prove the amortized complexity time of algorithms (in this case, the splay tree algorithm). Afterwards, we present some variants of tree structures that have been proven with ITP, namely with Isabelle and CFML. We then present the Iris framework which uses separation logic to reason about programs that deal with pointers.

2.1 Splay Trees

2.1.1 Practical Applications. Splay Trees have many practical applications, particularly in contexts where the same data is accessed frequently. Examples include network routing (where IP addresses are accessed frequently) and memory allocation algorithms.

An application particularly relevant to this project is the use of Splay Trees by the GNU Compiler Collection (GCC), a widely known sophisticated free collection of compilers for a wide variety of programming languages: C, C++, Objective-C, Objective-C++, Java, Fortran, Ada, and Go [Stallman et al. 2003]. GCC was originally written for the GNU operating system and is now available on UNIX and Linux operating systems with new version releases every year [GCC Team 2019].

2.1.2 Self-Adjusting Tree Structures. Splay trees are binary search trees (BST) that apply restructuring rules in each operation in order to improve the efficiency of future operations without needing extra space to do so. This restructuring is done by the splay heuristic method which has the responsibility to move more frequently accessed nodes towards the root while adjusting itself with constant time rotation operations along the way (Example of rotation in Figure 1).

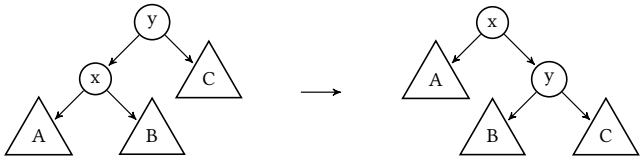

Fig. 1. Using the splay heuristic method on node x, it is performed the zig case preserving the search property.

2.2 Formal Verification of Tree Structures using ITPs

Tobias Nipkow already used an interactive theorem prover (ITP), namely Isabelle/HOL, to prove the functional correctness of the Splay Tree methods [Nipkow 2014]. However, he reasons on a functional Isabelle implementation which does not require memory resource reasoning. A pointer-based imperative implementation of the Splay Tree algorithm is more error prone than the functional implementation, once it may lead to malformations of the tree (e.g. occurrence of a cycle by putting one of the nodes pointing to the root, Figure 2) if not well implemented.

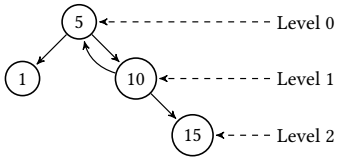

Fig. 2. A malformed self-adjusting tree. This would lead to a non halting situation if we tried to lookup for a key that was between 6 and 9.

CFML is a tool that is used to get a high degree of confidence in the correctness of Caml code with the use of the Coq proof assistant. It comes with a generator that converts Caml code in Characteristic Formulae (CF) Coq source. The CF of a program is a higher order logic formula that gives a sound description of the semantics of the program [Charguéraud 2011]. Arthur Charguéraud and François Pottier extended CFML, which allows reasoning about memory resources, with time credits and proved the correctness and amortized time complexity of their own Union-Find (UF) (root/link) pointer-based algorithm implementation [Charguéraud and Pottier 2019].

```
Theorem find_spec: ∀ D R V x, x ∈ D →
    app UnionFind_ml.find[x]
    PRE(UF D R V (2*|α(card D) + 4))
    POST(fun y => UF D R V * [R x = y])
```

Listing 1. The functional and time specification of the find method [Charguéraud and Pottier 2019].

2.3 The Iris Framework

Iris, is a framework for higher-order concurrent separation logic implemented and verified in the Coq proof assistant [Jung et al. 2018]. It provides us with Heap-Lang, a deeply embedded higher-order concurrent imperative programming language $\lambda_{\text{ref,conc}}$ in Coq. This framework uses separation logic [Reynolds 2002] rules on classical mutable shared data structure manipulation atomic commands [Jung et al. 2018] such as, the ones present in heap-lang syntax: allocation (ref), lookup (!), mutation (←) and compare-and-set (CAS). Heap-Lang lets us express concurrent programs by using the fork instruction and reason on them with the use of: *invariants* "to allow different threads to access

the same resources" and *ghost states* "to keep track of additional information" [Birkedal and Bizjak 2018].

This rich framework provides us with some of the classic separation logic connectives, such as the *separation conjunction* ($\star$), *separation implication* ($-\!\!*$) and the *points-to* predicate ($\hookrightarrow$) [Krebbers et al. 2017] (not to be mistaken by the Reynolds' singleton heap ($\rightarrow$) predicate). The semantic definition of such connectives is defined in Reynolds' paper about separation logic [Reynolds 2002]. A Iris proposition (iProp) has the following type ($State \xrightarrow{mon} Prop$), where a state $\sigma \in State$ is a finite map $\mathbf{N} \xrightarrow{fin} Val$ and $Prop$ is a Coq proposition [Krebbers et al. 2017].

3 A FUNCTIONAL IMPLEMENTATION OF SPLAY TREES

In this section we explain how we have proven the correctness of a functional implementation of splay trees. We start off by showing how we have translated Nipkow's Isabelle implementation and predicates [Nipkow 2014] to Gallina. Then we show how we have proven the correctness of the splay tree methods, mainly the **splay**, **insert**, **splay_max** and **delete** methods. Finally, we present a brief discussion about the differences from our proof of correctness and Nipkow's and how we have automated our proofs.

3.1 Nipkow's Implementation

To verify a functional implementation of the splay tree algorithm, we first defined a simple inductive type for binary trees. This binary tree inductive type has two constructors: **L** for leaves, which has no parameters (with notation $<||>$), and **T** for nodes that has as parameters a left binary tree t_1, an ordered type element $o.t$ and a right binary tree t_2 (with notation $<|\, t_1, o.t, t_2 |>$).

Since some of the proofs and predicate definitions require the use of a set data structure, we have decided to use a Coq module for sets implemented with a simple list data structure (since it already comes with some properties proven)[1]. Specifying that a binary tree is searchable required the set data structure XSet (an ordered type set) as shown in Listing 2.

```
Fixpoint bst (t : tree) : Prop :=
    match t with
    | <| |> => True
    | <| l, a, r |> => (bst l) ∧ (bst r) ∧
    XSet.For_all (fun n => n < a) (set_tree l) ∧
    XSet.For_all (fun n => a < n) (set_tree r)
end.
```

Listing 2. Binary search tree predicate

After the definition of the binary tree inductive type and the binary search tree predicate, we have simply translated the recursive splay tree algorithm from Nipkow's Isabelle implementation [Nipkow 2014] to Galina. This translation consisted in defining the **splay** tree function, the **insert** function, the **splay_max** function and the **delete** function.

3.2 Functional Correctness

In this section we present the most important lemmas that we have proven (some that already have been proven by Nipkow) related to the binary search tree. During the proving task, some difficulties arose, particularly how and where to perform induction. We then successfully proved

[1] Coq set module https://coq.inria.fr/library/Coq.MSets.MSetWeakList.html

these lemmas using functional induction, which performs case analysis and induction following the definition of a function" [2].

One of the main lemmas that we have first proven is the set invariant over splay: **After applying the splay function, the nodes of a binary tree are the same as the ones before the application.** Then, we have proven lemma in Equation 1 which states that: **Applying the splay tree function on a binary search tree preserves the binary search tree predicate**. In order to prove such lemma, we have used functional induction with the set invariant over splay lemma and applied a set of tactics to automate the proofs, resulting in a 4 line proof of Coq tactics.

$$\text{bst } t \longrightarrow \text{bst } (\text{splay } a\ t) \tag{1}$$

With the use of functional induction we have easily proven the binary search tree invariant over the insert method as well, since this method calls the splay function and we have already proven this invariant for the splay method (Equation 1). Nevertheless, the delete method calls the splay_max method which is equivalent to calling the splay method on the node with maximum value (Equation 3). Then before proving the correctness of the delete method, we have proved that the splay max function preserves the binary search tree invariant as mention in Equation 3. With the mentioned proofs above, we have then proved that the all splay tree method preserve the binary search tree invariant, as expected.

$$\text{bst } t \longrightarrow \text{bst } (\text{splay\_max } t) \tag{2}$$

$$\text{bst } t \longrightarrow (\forall\ x,\ \text{XSet.In } x\ (\text{set\_tree } t) \longrightarrow\ x < a \vee\ x = a) \longrightarrow \text{splay\_max } t\ =\ \text{splay } a\ t \tag{3}$$

$$\text{XSet.Equal } (\text{set\_tree } (\text{splay\_max } t))\ (\text{set\_tree } t) \tag{4}$$

3.3 Discussion

We have proven all the lemmas/theorems that Nipkow's has proven related to the splay tree functions. The lemmas that are related to transformations between trees and lists and between trees and maps that Nipkow has proven, were not proven by us.

Beside the theorems that were proven by Nipkow's, we have proven, unlike Nipkow, that **every possible tree constructed by the splay (lookup), insert and delete operations is a binary search tree, assuming that the initial tree is a binary search tree**. To prove this theorem, we assume that there is an arbitrary list l with key values that will be used for the input in the calls to these three operations (e.g., if list is [a;b] then one possible sequence would be (insert b (splay a t))).

```
Fixpoint splay_insert_delete_star t l :=
    match l with
    | [] => [t]
    | hd :: tl =>
      let insert_ := splay_insert_delete_star (insert hd t) tl in
      let delete_ := splay_insert_delete_star (delete hd t) tl in
      let splay_  := splay_insert_delete_star  (splay hd t) tl in
      insert_ ++ delete_ ++ splay_
    end.
```

Listing 3. Function that creates a list of all possible trees generated by list l, where (++) is the append list operation

After we have proved successfully the functional implementation, we have successfully extracted an implementation of the splay algorithm, from Gallina to OCaml, for the natural numbers set since the fact that they are an Ordered Type set is already proven.

[2]Functional induction: https://coq.inria.fr/refman/using/libraries/funind.html

4 A POINTER-BASED IMPLEMENTATION OF SPLAY TREES

In this section we describe how we have modeled the GCC splay tree pointer-based implementation from libgomp [GCC team 2019]. We first show how we have translated the C++ code and then how we modeled the splay tree data structure algorithm in order to prove its correctness. Later on, we explain how we have only proven the correctness of the splay method from the splay tree algorithm.

4.1 GCC's Splay Tree: Heap-Lang Code Translation

To perform the verification of GCC's splay tree algorithm using the Iris framework, we manually translated the C++ code from libgomp [GCC team 2019] to heap-lang, $\lambda_{\text{ref,conc}}$. In general, the translation of the C++ code to heap-lang is straightforward. However, a problem that occurred during this task is that some operations and control structures available in the C++ language are not present in heap-lang. For this reason, we had to translate these operations and control structures with what heap-lang had to offer us. There are four aspects that deserve being explicitly mentioned: **access and mutation of node fields**, **loops**, **access to addresses** and **generic types**.

4.2 Splay Tree Predicate

In the splay tree predicate, we have first created the binary search tree invariant, Inv $p\ D\ F\ V\ W$, which states that we have a binary search tree with root p, domain D (the nodes that belong to the tree structure), edge set F (the edges of the tree, e.g., $F\ x\ y\ \mathit{RIGHT}$ meaning that we have an edge from x to y with orientation *RIGHT*), value function V (which is the mapping of the nodes to the value that they hold) and finally W (the weight function that gives an arbitrary positive value to each of the node of the binary search tree, setup for the proof of the amortized logarithmic time).

Afterwards, we have defined the memory as a generalized map and created the invariant for the memory, Mem $D\ F\ V\ M$, which states that every node in domain D must have one of the four content values: *NodeB* (Node with two children), *NodeL* (Node with only left children), *NodeR* (Node with only right children) or *NodeN* (Node with no children), and must successfully map these content values to binary search tree components (edges and node values). We also define the invariant, mapsto_M M, that asserts ownership of all the pointers that are in map M.

The splay tree predicate consists of the three mentioned invariants: the binary search tree invariant **Inv**, the binary search tree memory invariant **Mem** and the ownership invariant **mapsto_M.** The splay tree invariant abstracts from the edge set F and memory M. The predicate ST $p\ D\ V\ W$ translates to saying that if we have a root node p, domain D, value function V and weight function W then there exists an edge set F and memory M for which F is a binary search tree (**Inv**), all pointers of M match with the edges of F and values of V (**Mem**) and we have ownership of all pointers in M (**mapsto_M**).

$$\begin{aligned}&\text{ST}\ p\ D\ V\ W \equiv \\ &\exists\ F\ M, \text{Inv}\ p\ D\ F\ V\ W\ \star\ \text{Mem}\ D\ F\ V\ M\ \star\ \text{mapsto\_M}\ M.\end{aligned} \tag{5}$$

4.3 Component properties

For the binary search tree invariant, we have proven certain properties for each of the predicates and invariants that it holds, namely for the: domain, edges and path. Three of the main proofs about the binary search tree invariant are 1) **There exists an unique path between two nodes of a binary search tree** (Equation 6) 2) **A node from a tree as at most one parent** (Equation 7) and 3) **A path between two nodes is finite** (Equation 8). For these proofs we had to extend the path inductive type to both path_count and path_memory. The path_count $F\ x\ y\ c$ states that there exists a path from x to y with size c and path_memory $F\ x\ y\ l$ states that there exists a path

from x to y and l is the trajectory witness for such path.

$$\forall\ F\ x\ y, \text{Inv}\ p\ D\ F\ V\ W \longrightarrow \text{path\_memory}\ F\ x\ y\ l_1 \longrightarrow \text{path\_memory}\ F\ x\ y\ l_2 \longrightarrow l_1 = l_2 \quad (6)$$

$$\forall\ F\ x\ p_1\ p_2, \text{Inv}\ p\ D\ F\ V\ W \longrightarrow F\ p_1\ x \longrightarrow F\ p_2\ x \longrightarrow p_1 = p_2 \quad (7)$$

$$\forall\ F\ x\ y\ c, \text{Inv}\ p\ D\ F\ V\ W \longrightarrow \text{path\_count}\ F\ x\ y\ c \longrightarrow c \leq |D| \quad (8)$$

4.4 Edge set manipulation

To model the rotations done by the splay algorithm, we need to be able to manipulate the edge set F. For this reason we define several operations that make this possible, such as: add edge, remove edge, update edge, union edge and elimination of a set of edges. The add edge operation, seen in Equation 9, receives an edge set F and adds edge from x' to y' with orientation o' (orientation in a binary search tree can be either *LEFT*, *RIGHT*) with the use of the or logical connective. In order to remove an edge from x' to y', we use the remove edge operation mentioned in Equation 10.

$$add\_edge\ F\ x'\ y'\ o' \equiv \lambda\ x\ y\ o,\ F\ x\ y\ o \vee (x = x' \wedge y = y' \wedge o = o') \quad (9)$$

$$remove\_edge\ F\ x'\ y' \equiv \lambda\ x\ y\ o,\ F\ x\ y\ o \wedge \neg(x = x' \wedge y = y') \quad (10)$$

Other important manipulation operations that we have defined are the update edge, the union edge and the removal of a set of edges that belongs to certain domain. The update edge operation definition in Equation 11 redirects an edge from a node to another by removing the edge for where he points to where he will point next. Other relevant operation, in definition 12, is the union edge which does the union of two edge sets F_1 and F_2 with the or logical connective (similar to the add edge operation). Finally, we have two more operation, remove edge that are not in D (Equation 13) and remove edge that are in D (Equation 14).

$$update\_edge\ F\ x'\ y' z' o' \equiv \lambda\ x\ y\ o, \neg((x' = x \wedge y' = y) \vee (x' = x \wedge z' = y)) \wedge F\ x\ y\ o \vee (x' = x \wedge z' = y \wedge o' = o) \quad (11)$$

$$union\_edge\ F_1\ F_2 \equiv \lambda\ x\ y\ o,\ F_1\ x\ y\ o \vee F_2\ x\ y\ o \quad (12)$$

$$remove\_edge\_that\_are\_not\_in\_D\ F\ D \equiv \lambda\ x\ y\ o,\ x \in D \wedge y \in D \wedge F\ x\ y\ o \quad (13)$$

$$remove\_edge\_that\_are\_in\_D\ F\ D \equiv \lambda\ x\ y\ o,\ \neg(x \in D) \wedge \neg(y \in D) \wedge F\ x\ y\ o \quad (14)$$

These last two operations in Equation 13 and 14 are important when we want to extract a sub-tree of a binary search tree, perform some operation on this sub-tree and then rejoin it with the original tree. This is possible with the use of the child (Equation 15) and join (Equation 16) lemmas that we have proven. The child lemma in Equation 15 states that if we have a binary search tree with root p and we have some edge from p to x ($F\ p\ x\ o$), then the sub-tree with root x is also a binary search tree. Meanwhile, the join lemma in equation 16 states that if we do some transformation in the child sub-tree, and this transformation guarantees that the sub-tree is still a binary search tree, then we can rejoin the root with this new transformed sub-tree and the overall tree is a search tree.

$$\begin{aligned}&\text{Inv}\ p\ D\ F\ V\ W \longrightarrow F\ p\ x\ o \longrightarrow \text{let}\ D' := (\text{descendants}\ F\ x)\ \text{in}\\ &\text{let}\ F' := (\text{remove\_edge\_that\_are\_not\_in\_D}\ F\ D')\ \text{in}\\ &\text{Inv}\ x\ D'\ F'\ V\ W\end{aligned} \quad (15)$$

$$\begin{aligned}&\text{Inv}\ p\ D\ F\ V\ W \longrightarrow F\ p\ x\ o \longrightarrow \text{let}\ D' := (\text{descendants}\ F\ x)\ \text{in}\\ &\text{let}\ FC' := (\text{remove\_edge\_that\_are\_in\_D}\ F\ D')\ \text{in}\\ &\text{Inv}\ z\ D'\ F'\ V\ W \longrightarrow \text{let}\ F'' := (\text{add\_edge}(\text{union\_edge}\ F'\ FC')\ p\ z\ o)\ \text{in}\\ &\text{Inv}\ p\ D\ F''\ V\ W.\end{aligned} \quad (16)$$

4.5 Specification and Correctness of Rotations

During the proof of correctness of the splay tree method, we needed to prove the correctness of every rotation performed on the root and on the children of the root. Although the correctness of these operations were easy to prove, since we did not managed to find a good way to automate the proofs, we had to proof all the 48 cases of rotations for the root and the 64 for the children of the root which turned out to be costly in terms of lines of proving.

4.6 Iterative Rotate Inductive Predicate

We have modeled the splay tree method as an inductive predicate called fw_ir, that we read as forward iterative rotate. The predicate

$$\text{fw\_ir } F\ V\ p\ x\ k\ n\ F'\ s$$

says that if we have edge set F and value function V, and we perform the splay tree method on root p with key k, then n rotations are required to get to edge set F' with root x and state s. The state of the splay tree method can be either GOING, if the algorithm did not end, or ENDED, if it is over. In total, we needed 15 rules to define this inductive predicate, showing the complexity of GCC's splay method implementation. After defining such predicate, we have proven, in Equation 17, that for any state in between the cycle of the iterative splay tree method, the tree preserves the search property.

$$\text{Inv } p\ D\ F\ V\ W \longrightarrow \text{fw\_ir } F\ V\ p\ x\ z\ n\ F'\ s \longrightarrow \text{Inv } x\ D\ F'\ V\ W. \tag{17}$$

Afterwards we have tried to prove that the splay tree algorithm terminates: considering a binary search tree with root p, edge set F and value function V, there exists a final: root p', edge set F' and a finite number of rotations n for fw_ir $F\ V\ p\ p'\ k\ n\ F'$ ENDED, i.e.,

$$\exists\ p'\ n\ F', \text{fw\_ir } F\ V\ p\ p'\ k\ n\ F' \text{ ENDED}$$

The splay tree termination proof mentioned was reduced to proving that after a constant number of rotations 4, if possible, the path to the key node that is being searched for decreases after these 4 rotations (Equation 18). After analysing the double rotation, i.e., the zig-zig and zig-zag operations, we have informally proven this lemma. Nevertheless, since the fw_ir predicate that models the splay tree algorithm is quite complex, proving it required us to prove 49 cases which were costly, and so we did not managed to prove this lemma and had to assume for know to be true.

$$\begin{aligned}
&\text{Inv } p\ D\ F\ V\ W \longrightarrow \\
&\text{path_find_count } F\ V\ p\ x\ z\ (4+n)\ \mathit{ENDED} \longrightarrow \\
&\text{fw_ir } F\ V\ p\ x'\ z\ 4\ F'\ \mathit{GOING} \longrightarrow \\
&(\exists\ m\ y,\ (m < 4+n) \wedge \text{path_find_count } F'\ V\ x'\ y\ z\ m\ \mathit{ENDED})
\end{aligned} \tag{18}$$

4.7 Splay Method Specification and Proof

After all this effort, we successfully proven that the inductive predicate bw_ir implements the splay tree algorithm from the GCC implementation. And then assuming that there exists an end to the splay tree algorithm applied to a binary search tree, i.e., assuming lemma 18, we have the prove for the correctness of the splay tree method in Equation 19, which states, as a Hoare triple, that the splay tree method ends on a binary search tree and the result is a binary search tree.

$$\begin{aligned}
&\{\{\{pp \mapsto \#p \star \text{ ST } p\ D\ V\ W\ \}\}\} \\
&\text{splay_tree_splay } \#pp\ \#k;;!\#pp \\
&\{\{\{(p' : loc), RET\#(p'); pp \mapsto \#p' \star \text{ST } p'\ D\ V\ W\ \}\}\}.
\end{aligned} \tag{19}$$

5 EVALUATION

During the evaluation phase, we have counted the lines related to the **Code**, **Tactic definition** and **Theorem/Lemmas** for the functional implementation in Table 1, and for the pointer-based implementation in Table 2). From these mentioned tables, we can see that the ratio from number of lines of code to number of lines of theorems/lemmas is roughly 1:9 in the functional implementation and 1:138 in the pointer-based implementation. This ratio difference between these two implementations and the difference between the overall total number of lines of the two implementations, shows the difficulty between proving a functional implementation and a pointer-based implementation, which, as we explained in Section 2.2, the later is more error prone.

Functional	#lines
Code **1**	205
Tactic definition **2**	176
Theorem/Lemmas **3**	1775
TOTAL: **3**	2156

Table 1. Functional number of lines

Pointer-based	#lines
Code **1**	200
Tactic definition **2**	223
Theorem/Lemmas **3**	27594
TOTAL: **3**	28017

Table 2. Pointer based number of lines

6 CONCLUSION

The main challenge addressed by this project is the formal verification of pointer-based splay trees using the Iris framework. We started by using Coq to successfully prove functional correctness of a functional implementation of splay trees, in a development inspired by Nipkow's work [Nipkow 2014]. We then modelled and verified a real pointer-based implementation of splay trees — the one used by GCC, the GNU's Compiler Collection. The verification of this pointer-based implementation proved to be much more challenging than the verification of the functional implementation. For example, we left the lemma in Equation 18 unproven. Nevertheless, we were able to verify key properties of splay trees and we have organized our work in such a way that anyone who wants to prove the correctness of algorithms related with binary search tree structures has a good starting point with some important properties already proven.

Our work for the proof of GCC's splay tree pointer-based implementation, was inspired by the work of Mével, G et al. [Mével et al. 2019] for the union find algorithm. After all, the Iris framework is still a recent tool and their work was extremely important to understand how they have approached the verification problem. In particular, we modeled the tree structure as a graph and the memory as a generalized map in the same way as Mével, G et al. However, since we are working with a completely different structure and algorithms from those considered by Mével, G et al., then we had different challenges. For example, in the find operation of the union find algorithm, they have defined an inductive predicate with only two (**2**) cases (for the root node and for a non-root node). We noticed that the heap-lang function for the splay method deals with a lot more case conditions than just **2**, which substantially increases the verification complexity.

CURRENT LIMITATIONS AND FUTURE WORK

At the moment, we have some prepared setup to start proving intensional aspects of the splay tree algorithm, namely its logarithmic amortized time complexity. Nevertheless, the lemma that we have referred in Section 4.6 Equation 18 was left unproven. We have informally proven it (on paper), but did not prove it in the Coq Proof Assistant. Therefore, since it is not desirable to have lemmas depending on other unproven lemmas (this case the splay method specification), then the proof of this lemma should be top priority.

Insert and delete operations. We are confident that we would have proven the correctness of the insert method for GCC's splay tree pointer-based implementation. However, we felt that the lemma in Section 4.6 Equation 18 was more important to invest our time in proving due to the reasons that we have previously mentioned. Nevertheless, the remove method from the GCC implementation would require a little more effort to prove, due to the fact that we would have to create an inductive predicate that would model the splay_max while loop.

Time complexity properties. After the proof of correctness of these splay tree methods we would, for future work, start proving time complexity properties of the splay tree algorithm. Nevertheless, to prove the amortized algorithm time, we would have to change the splay tree predicate mentioned in Section 4.2 to have stored in itself Φ time credits, which we did not did because we would have to modify every lemma that would use such predicate. And besides this extension of the splay tree predicate, we would have to prove the difference of potential for each rotation operation (single and double).

Concurrency. During this project, we also did not use Iris concurrency reasoning which we would like to further explore on concurrent algorithms related to tree structures. One of the concurrent tree structures that we would wish to explore in the future is the *counting-based* tree (CBTree) [Afek et al. 2014], a concurrent variant of Splay Trees. This would allow us to explore more of what the Iris framework has to offer us, such as: *invariants* and *ghost states* [Birkedal and Bizjak 2018].

REFERENCES

[n. d.]. The Coq Proof Assistant. ([n. d.]). https://coq.inria.fr/

Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E Tarjan. 2014. The CB tree: a practical concurrent self-adjusting search tree. *Distributed computing* 27, 6 (2014), 393–417.

Lars Birkedal and Aleš Bizjak. 2018. Lecture notes on iris: Higher-order concurrent separation logic. (2018).

Arthur Charguéraud. 2011. Characteristic formulae for the verification of imperative programs. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 418–430.

Arthur Charguéraud and François Pottier. 2019. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning* 62, 3 (2019), 331–365.

GCC team. 2019. gcc. https://github.com/gcc-mirror/gcc/blob/master/libgomp/splay-tree.c. (2019). [Online; accessed 2019-12-21].

GCC Team. 2019. GCC Releases. https://gcc.gnu.org/releases.html. (2019). [Online; accessed 2019-12-21].

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018).

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 205–217.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time credits and time receipts in Iris. In *European Symposium on Programming*. Springer, 3–29.

Tobias Nipkow. 2014. Splay tree. *Archive of Formal Proofs* 2014 (2014).

John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 55–74.

Richard M Stallman et al. 2003. Using the GNU compiler collection. *Free Software Foundation* 4, 02 (2003).