TÉCNICO LISBOA

Formal Verification of Pointer-Based Splay Trees in Iris

Ricardo Ciríaco da Graça

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João Fernando Peixoto Ferreira
Prof. Simão Patricio Melo de Sousa

Examination Committee

Chairperson: Prof. Luís Manuel Antunes Veiga
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Prof. Jan Gunnar Cederquist

January 2021

Acknowledgments

I would like to thank my parents, sister and family for their support during this time. This path could not have been possible without them.

I would also like to give a special thanks to my Supervisor, João Ferreira, who helped me a lot during this year. I appreciate all the time that he has spent to support me as his oriented student.

To my Co-Supervisor, Simão Sousa, I also appreciated his help throughout the development of the thesis.

At last, I would also like to thank my friends that have kept me company during the pandemic, they surely made this year a lot better.

Abstract

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.

Keywords

Formal Verification; Coq; Iris Framework; Splay Tree; Heap-lang; GNU Compiler Collection

Resumo

As aplicações, quando suscetiveis a falhas ou a um desempenho decadente, podem afetar significamente os custos daqueles que dependem de tal. Assim sendo, há uma necessidade de garantir que estas aplicações não falham e que tenham o comportamento esperado. Testar é útil na prática, mas só para mitigar certos *bugs* e não para ter esta garantia. No entanto, a verificação formal pode ser usada para garantir que o comportamento de um programa obedece a certa especificação. Contudo, a verificação formal de código usado no mundo industrial, que normalmente lida com estruturas de dados mutáveis e não triviais, é uma tarefa custosa. Nos últimos anos, muitos avanços foram feitos no que toca à verificação formal, mas ainda há muitas oportunidades para verifica programas de alto uso no mercado. Neste projeto exploramos o assistente de provas Coq e o *framework* Iris para verificar a correção de uma implementação de splay trees baseada em apontadores da aplicação GNU Compiler Collection (GCC) que tem uso na biblioteca libgomp (GNU Offloading and Multi Processing Runtime Library). Também provamos uma implementação functional do algoritmo de *splay trees* para tipo de dados generalizado, com o assistente de provas Coq. Com o conhecimento que temos neste momento, fornecemos a implementação baseada em ponteiros formalmente verificada mais completa de Árvores Splay.

Palavras Chave

Formal Verification; Coq; Iris Framework; Splay Tree; Heap-lang; GNU Compiler Collection

Contents

List of Figures

List of Tables

Listings

1

Introduction

Contents

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 [6] and the Iris framework [1] 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.

Below, we present in detail our work objectives and we enumerate the contributions that we have made towards the verification of correctness of algorithms related with tree structures. We finalize this chapter with a description of how this document is organized.

1.1 Work Objectives

Our work objectives are:

1. Verify a functional implementation of the splay tree algorithm for a generalized ordered datatype using the interactive proof assistant Coq. To achieve this objective, we first translate a functional implementation (Nipkow's implementation [7]) of the splay tree algorithm to Gallina (Coq's specification language) and then prove the specification of the behaviour of each of the methods, namely: splay (the most important method), splay_max, insert and remove.

2. Verify a real-world imperative, pointer-based implementation of the splay tree algorithm using the interactive proof assistant Coq. We then proceed to prove a real imperative pointer-based implementation of the splay tree algorithm using the Iris framework, which allows us to reason about pointer-based algorithms with the help of separation logic. We decompose this objective into three sub-objectives:

2.1. Translate GCC's pointer-based implementation written in the C language to a language that allow us to reason in Iris' framework, heap-lang. Although some operations and control structures from the C language are not available in heap-lang (e.g., while loops and address access), we can replace these with other mechanisms that heap-lang provides us. With the heap-lang translated implementation, we can then proceed to its verification.

2.2. 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. For this objective, we take as an example the union find algorithm that was verified with the Iris' framework by Mével, G et al. [4]. Having the splay tree predicate and the implementation modeled in the proof assistant Coq, we can then complete the next sub-objective.

2.3. Specify and verify the correctness of the lemmas related to the splay tree methods. This involves the use of Iris' proof mode that allows us to both specify such lemmas and prove them with provided tactics for Iris' logic.

Figure 1.1 shows a diagram of the sequence of tasks for the work objective **2.**. First we translate the GCC implementation to heap-lang, then we model the splay tree predicate, and then specify and prove the correctness of the heap-lang splay algorithm. The splay tree implementation that we propose to verify is the one used by the well-known compiler GNU's Compiler Collection (GCC).

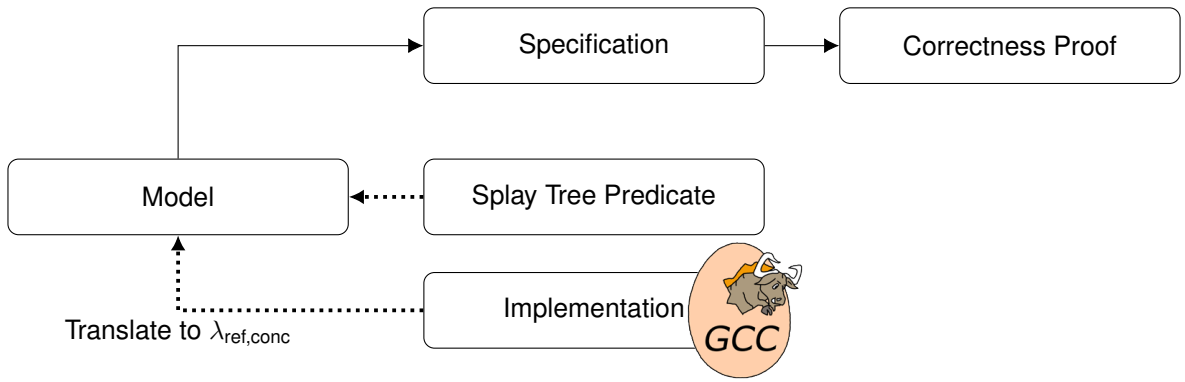

Figure 1.1: Sequence of objectives in order to have a fully verified heap-lang translation of the splay tree algorithm implementation from the well-known GCC compiler

1.2 Contributions

During the execution of the aforementioned objectives, we have managed to successfully verify in Coq the Gallina translation of Nipkow's implementation of the splay tree algorithm [7]. We have also proved additional lemmas and theorems not considered by Nipkow and we successfully extracted the verified code, in which the keys are natural numbers, to the OCaml functional language.

For the verification of the pointer-based implementation, we started by successfully translating 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 (e.g., a node in a binary search tree has at most one parent).

In order to prove some of these lemmas, we had to extend some inductive types to have more information about these types. For example, to prove that a node has at most one parent 4.5.2, we had to make sure that there is an unique path between two nodes in a binary search tree, and for the later we needed to extend the path inductive type so we can have the full information of the path trajectory and not only its endpoints.

Finally, we have successfully proven the correctness of the splay tree method, but, due to time constraints, at the time of writing, we left one of the lemmas, that makes this proof possible, unproven (we have informally proven it, though).

To the best of our knowledge, we provide the most complete formally verified pointer-based implementation of Splay Trees. Moreover, rather than implementing a version that facilitates the verification process, we verify the translation of an existing and widely-used real-world implementation. Finally, it is worth mentioning that our development provides a good starting point to prove time complexity properties of splay trees, using the techniques proposed by Mével, G et al. [4].

1.3 Organization of the Document

This thesis presents first, in **Chapter 2**, the background and related work, referring to splay trees (Section 2.1), the formal verification of tree structures using interactive theorem provers (ITPs) (Section 2.2), the Iris framework (Section 2.3) and finally time credits and time receipts in Iris (Section 2.4).

Subsequently, in **Chapter 3**, we first discuss in Section 3.1, Nipkow's functional implementation and some of the predicates and data structures that were used to specify and prove some of the predicates and lemmas. Afterwards, in Section 3.2, we show some of the lemmas that we have proven in Coq and show how we have proven some of them. Finally, in Section 3.3, we show how we have proven that given a binary search tree, then all possible generated trees, with the splay tree methods, result in a

binary search tree as well. In this later section, we also show how we have extracted the OCaml code for the verified splayed algorithm for natural numbers.

Afterwards, in **Chapter 4**, we show how we have translated the GCC code to heap-lang (Section 4.1) and how we have created the splay tree predicate (Section 4.2) with the needed predicates and invariants. Then we discuss some of the properties of the predicates and inductive types, such as domain properties (Section 4.3), link properties (Section 4.4) and path properties (Section 4.5). In Section 4.6, we talk about how we manipulate the edge set predicate and some relevant proven lemmas that help us to prove the correctness of the rotations of the splay tree algorithm. We then show some of the proofs related to an inductive predicate that models the binary search algorithm on a binary search tree (Section 4.7). Finally, we explain how we have proven the correctness of the translated heap-lang splay method in Section 4.10 after we have explained how we have verified the correctness of the rotate operations performed on the tree (Section 4.8) and how we have created an inductive type that makes part of the proof of the correctness of the splay method (Section 4.9)

In **Chapter 5**, we show some of the metrics about the work that we have done, both for the functional implementation, in Section 5.1, and for the pointer-based implementation, in Section 5.2. Some of the metrics that we show are: the number of total lines, the number of lemmas/theorems that we have specified, the number of tactics that we have used to prove these lemmas/theorems and the number of definitions such as values, predicates, functions and types.

Finally, in **Chapter 6**, we show the main challenges that we have encountered, we briefly discuss the results of the metrics that we have mentioned in Chapter 5, and we present some of the differences between our work and Mével, G et al.'s work. We conclude with a discussion on the current limitations and future work, in Section 6.

2

Background and Related Work

Contents

In this chapter we first discuss splay trees. 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. Finally, we talk about time credits and time receipts that were developed by Mével, G. et al. [4] in the Iris framework.

2.1 Splay Trees

In this section, we 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).

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 [8]. 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 [9].

GCC uses Splay Trees in its Offloading and Multi Processing Runtime Library (libgomp). The library offers a fairly simple Splay Tree pointer-based C implementation [10] used by the OpenACC (an application programming interface that is used to "support offloading of code to accelerator devices" [11]) to manage the memory of devices [12].

In the Offloading implementation in the GCC application, the splay trees are used to do the mapping between the host address and the device target address. They do not justify their use for splay trees, however we suspect that it is because the same host may be accessed frequently. They offer a library with the prototypes and definitions of the splay tree algorithm [1] and also the implementation written in the C language [2].

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

[1] Splay tree library: https://code.woboq.org/gcc/libgomp/splay-tree.h.html
[2] Splay tree implementation in the C language: https://code.woboq.org/gcc/libgomp/splay-tree.c.html

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 2.1).

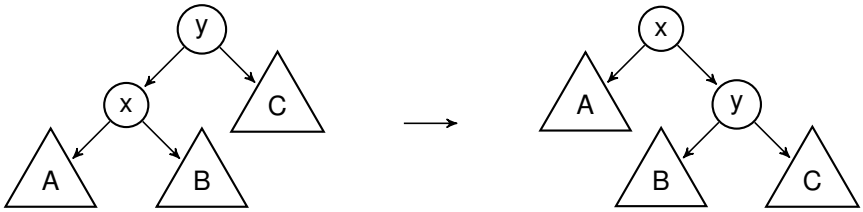

Figure 2.1: Using the splay heuristic method on node x, it is performed the zig case preserving the search property.

This data structure has been and still is a case study when learning about amortized analysis. "By amortized time we mean the average time of an operation in a worst-case sequence of operations" [13]. To do this analysis, we can use a potential function $\mathbf{\Phi}$, a function that assigns a real positive value to each configuration of the data structure D. The amortized time a on the i_{th} operation, a_i, can be less or more than the actual time t_i, depending only on the difference of the potential of the resulted data structure after the i_{th} operation, $\mathbf{\Phi}(\mathsf{D}_i)$, with the potential of the data structure before the operation $\mathbf{\Phi}(\mathsf{D}_{i-1})$, $\mathbf{\Phi}(\mathsf{D}_i) - \mathbf{\Phi}(\mathsf{D}_{i-1})$:

$$a_i = t_i + \mathbf{\Phi}(\mathsf{D}_i) - \mathbf{\Phi}(\mathsf{D}_{i-1})$$

If this difference is positive, then the amortized time represents an overcharge of time in the i_{th} operation, otherwise it represents an undercharge [14]. So, after a sequence of m operations, and considering that the initial data structure has 0 potential, $\mathbf{\Phi}(\mathsf{D}_0) = 0$, with the telescope rule we deduce that the total amortized time A is an upper bound on the total actual time, since $\forall i, \mathbf{\Phi}(\mathsf{D}_i) \geq 0$:

$$A = \sum_{i=1}^{m} t_i + \mathbf{\Phi}(\mathsf{D}_i) - \mathbf{\Phi}(D_{i-1}) = \sum_{i=1}^{m} t_i + \mathbf{\Phi}(\mathsf{D}_m) \geq \sum_{i=1}^{m} t_i$$

To define the potential function of the Splay Tree, Sleator, D.D. et al. [13] have defined a weight function W that maps every node x to an arbitrary, but fixed, positive weight $W(x)$. After that, they have defined a size function S that computes the sum of all the node's weight of the tree t, $S(t)$. They then define the rank of the node as the function R, $R(x) = log_2(S(x))$, and finally they define (as it is in Equation 2.1) the potential function of the Splay Tree $\mathbf{\Phi}$ to be the sum of all the ranks of each node.

$$\mathbf{\Phi}(\mathsf{ST}) = \sum_{\mathsf{x} \in \mathsf{nodes(ST)}} R(x) \tag{2.1}$$

With this potential function, they have proved the bounded logarithmic amortized time of the Splay Tree's splay operation and all the other methods: lookup, insert and delete, which use, at the start, one call to the splay operation. They state and prove several lemmas, including the famous access lemma:

Access Lemma [13] The amortized time to splay a tree with root t at a node x is at most $3(R(t) - R(x)) + 1 = O(log(R(t)/S(x)))$

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 [7]. He proves the following lemmas regarding the splay method:

- $size(splay\ a\ t) = size\ t$: The size of the tree is the same after an application of a splay operation.
- $bst\ t \implies splay\ a\ t = Node\ l\ e\ r \implies x \in$ set-tree $l \implies x < a$: After splaying an element a of tree t, all elements to the left of the root node are lower than the splayed element a even if a does not belong to the tree.
- set-tree$(splay\ a\ t) =$ set-tree t : The set of elements of a tree t after a splay application is the same as before the application.
- $bst\ t \implies bst(splay\ a\ t)$: If t is a binary search tree (bst), then splaying a key a in it results also in a binary search tree.
- $[[bst\ t;\ splay\ a\ t = t']] \implies a \in$ set-tree $t \iff (\exists lr.t' = Node\ l\ a\ r)$: If t is a binary tree and t' is the result of splaying a in tree t, then a is in tree t if and only if it is the root of tree t'.

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 2.2) if not well implemented.

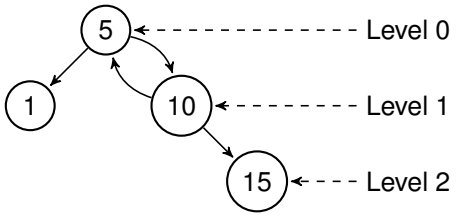

Figure 2.2: A malformed self-adjusting tree. This would lead to a non halting situation if we tried to lookup for a key that was between 6 and 9.

Nipkow also proves the nontrivial amortized logarithmic complexity time of the, splay, insert, delete, tree algorithm methods with a lightweight framework that supports proofs at a high level of abstraction [15]. A small remark of his framework and CFML tool is that only the function calls are counted as opposed to Iris with time credits. This is fine, since they define the functions as recursive and "the asymptotic complexity and the number of recursive calls necessary for the evaluation of the program are of the same order-of-magnitude" [16]. Nipkow proves the access lemma with the use of the potential function, mentioned before, for a tree t and element a [15] :

$$A\ a\ t\ \leq\ 3\ \times\ (\varphi\ t\ -\ \varphi\ \langle\ l,\ a,\ r\rangle)\ +\ 1 \text{ (Access Lemma)}$$

$$|\langle\rangle|\ =\ 0 \text{ and } |\langle\ l,\ a,\ r\rangle|\ =\ |l| + |r| + 1 \text{ (Size of tree)}$$

$$\varphi\ t\ =\ \log_2\ |t| \text{ (Rank of node)}$$

$$\mathbf{\Phi}\langle\rangle = 0 \text{ (Potential of leaf)}$$

$$\mathbf{\Phi}\langle l,\ a,\ r\rangle = \mathbf{\Phi} l\ +\ \mathbf{\Phi} r\ + \varphi\langle l,\ a,\ r\rangle \text{ (Potential of node) [15]}$$

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 [17]. 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 [5].

```
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 2.1: The functional and time specification of the find method [5].

In Listing 2.1, we see the functional and time specification of the find method. The app UnionFind_ml.find[x] is the CF of the correspondent find Caml implementation. The keyword PRE referrers to the precondition and POST to the postcondition in a Hoare triple style. Succinctly, the specification says that the find method does not make changes to the UF predicate (although the graph is modified by the compression function), that the output of the find method is the representative (the root) of x (R x = y) and that the method runs in $2 \times \alpha(|\mathsf{D}|) + 4$ [3] computational steps.

[3] α is the inverse of Ackerman's function.

2.3 The Iris Framework

Iris, is a framework for higher-order concurrent separation logic implemented and verified in the Coq proof assistant [1]. It provides us with Heap-Lang, a deeply embedded higher-order concurrent imperative programming language $\lambda_{\text{ref,conc}}$ in Coq (Figure 2.3). This framework uses separation logic [18] rules on classical mutable shared data structure manipulation atomic commands [1] such as, the ones present in the language: allocation (ref), lookup (!), mutation ($\leftarrow$) and compare-and-set (CAS). Heap-Lang lets us use functions as values, i.e., specify higher-order functions. It also lets us express concurrent programs by using the fork instruction and reason on them with the use of some of Iris' ingredients: *invariants* "to allow different threads to access the same resources" and *ghost states* "to keep track of additional information" [2].

$$
\begin{aligned}
Val \quad v &::= () \mid true \mid false \mid n \mid l \mid (v, v) \mid inj_1 v \mid inj_2 v \mid rec\ f(x) = e \\
Exp \quad e &::= x \mid n \mid e \odot e \mid () \mid true \mid false \mid if\ e\ then\ e\ else\ e \mid l \\
&\quad \mid (e, e) \mid \pi_1\ e \mid \pi_2\ e \mid inj_1\ e \mid inj_2\ e \\
&\quad \mid match\ e\ with\ inj_1\ x \implies e \mid inj_2\ y \implies e\ end \\
&\quad \mid rec\ f(x) = e \mid e\ e \\
&\quad \mid ref(e) \mid !e \mid e \leftarrow e \mid cas(e, e, e) \mid fork\{e\}
\end{aligned}
$$

Figure 2.3: Iris' $\lambda_{\text{ref,conc}}$ programming language syntax [1].

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 ($\hookrightarrow$) [19] (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 [18]. 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 [19]. In Iris, the specification of an Heap-Lang program e is often written as a Hoare triple as seen in figure 2.4.

$$
\frac{\mathbf{\Gamma} \vdash \mathsf{P} : \mathsf{Prop} \qquad \mathbf{\Gamma} \vdash \mathsf{e} : \mathsf{Exp} \qquad \mathbf{\Gamma} \vdash \mathbf{\Phi} : \mathsf{Val} \to \mathsf{Prop}}{\mathbf{\Gamma} \vdash \{\mathsf{P}\} e \{\mathbf{\Phi}\} : \mathsf{Prop}}
$$

Figure 2.4: The Hoare triple proposition defined in Iris, $\mathbf{\Phi}$ has a binder for the return value of expression e [2].

This triple states that if the heap-lang, $\lambda_{\text{ref,conc}}$, program **e** runs on a heap h satisfying the Iris precondition P, then, the program halts and if it terminates with the value v and a heap h', then h' must satisfy the Iris postcondition $\mathbf{\Phi}(v)$. In Iris' lecture notes [2] there is a list of rules that let us verify the correctness

of Heap-Lang programs specified as Hoare triple (Ht). A key feature of this framework, is the addition of *later modality* $\triangleright$ and its main punch, the Löb rule (figure 2.5).

$$\frac{\mathsf{Q} \wedge \triangleright \mathsf{P} \vdash \mathsf{P}}{\mathsf{Q} \vdash \mathsf{P}}$$

Figure 2.5: The Iris Löb rule.

The *later modality* ensures that to prove P, we must first do some work, i.e., a reduction step has to be taken, in order to use P as an assumption (This is well reflected by the Ht-Beta rule in Appendix Figure A.1). When this modality is omitted, the logic becomes inconsistent [3]. In addition, Iris introduces yet another modality, the *persistent modality* $\square$. Informally, $\square$P is like P, but it does not assert any exclusive ownership over resources, hence, it can be duplicated. These modalities are important when discussing Iris' concurrency.

$$\frac{\triangleright\, l \hookrightarrow w \star \triangleright (l \hookrightarrow v \mathrel{-\!\!*} \mathbf{\Phi}())}{wp\ (l \leftarrow v)\{\mathbf{\Phi}\}}$$

Figure 2.6: Iris' weakest precondition for the store operation. [2].

Iris' Hoare triple $\{P\}e\{\mathbf{\Phi}\}$ is equivalent to $\square(P \mathrel{-\!\!*} wp\ e\{\mathbf{\Phi}\})$ (it is persistent $\square$, meaning that we could have nested Hoare triples). The resources represented by $wp\ e\{\mathbf{\Phi}\}$ are the weakest precondition, i.e., the minimum amount of resources that must be owned in order to ensure safety over program e and for the postcondition $\mathbf{\Phi}$ to be true. As an example, consider that we want to make a mutation, with the store operation, i.e., $l \leftarrow v$. We first need to ensure that l is allocated $l \hookrightarrow w$ (it points to some value w, thus it is safe to mutate), and, after one reduction step $\triangleright$, l points to v, $l \hookrightarrow v$, and ensures that the mutation operation returns the unit value $(l \hookrightarrow v \mathrel{-\!\!*} \mathbf{\Phi}())$. This weakest precondition store rule is reflected in figure 2.6.

$$\frac{\mathsf{S} \vdash \{\mathsf{P}_1\}e_1\{v.\mathsf{Q}_1\} \qquad \mathsf{S} \vdash \{\mathsf{P}_2\}e_2\{v.\mathsf{Q}_2\}}{\mathsf{S} \vdash \{\mathsf{P}_1 \star \mathsf{P}_2\}e_1||e_2\{v.\exists v_1\ v_2.v = (v_1, v_2) \star \mathsf{Q}_1[v_1/v] \star \mathsf{Q}_2[v_2/v]\}}$$

Figure 2.7: Disjointed concurrency rule.

As referred, one of Iris' main ingredients that lets us reason about concurrency logic is *invariants*. To reason on concurrent programs, suppose $e_1 \mid\mid e_2$ is the parallel execution of expression e_1 with e_2 and that this execution must wait until both are finished (the $||$ operation can be defined with the fork expression [2]). If e_1 and e_2 do not share resources, then, we could just apply the disjointed concurrency rule (figure 2.7). However, if these expressions share resources, we cannot simply split them and give them to the two subcomputations (as in figure 2.8)!

$$\{l \hookrightarrow n\}(l \leftarrow !l + 1) \mid\mid (l \leftarrow !l + 1); !l\{v.v \geq n\}$$

Figure 2.8: Cannot split the resource $l \hookrightarrow n$ to the two subcomputations since $l \hookrightarrow n \star l \hookrightarrow n \vdash$ False.

To solve this issue, we need *invariants* which let the same resources to be accessed by different threads, without creating any inconsistencies, i.e., in a control way [2]. Succinctly, the resources to be shared, P, are given away (are lost) to an *invariant*, obtaining thus an *invariant* $\boxed{\mathsf{P}}^{\iota}$ for some namespace ι. Since *invariants* are persistent, we can duplicate them, and then they can be passed to multiple subcomputations.

Type Rule:

$$\frac{\mathbf{\Gamma} \vdash \mathsf{P} : \mathsf{Prop} \qquad \mathbf{\Gamma} \vdash \iota : \mathsf{InvName}}{\mathbf{\Gamma} \vdash \boxed{\mathsf{P}}^{\iota}}$$

Allocation rule:

$$\frac{\varepsilon \text{ Infinite} \qquad \mathsf{S} \wedge \exists \iota \in \varepsilon. \boxed{\mathsf{P}}^{\iota} \vdash \{\mathsf{Q}\}e\{v.\mathsf{R}\}_{\varepsilon}}{\mathsf{S} \vdash \{\triangleright \mathsf{P} \star \mathsf{Q}\}e\{v.\mathsf{R}\}_{\varepsilon}}$$

Opening Rule:

$$\frac{e \text{ is an atomic expression} \qquad \mathsf{S} \wedge \boxed{\mathsf{P}}^{\iota} \vdash \{\triangleright \mathsf{P} \star \mathsf{Q}\}e\{v. \triangleright \mathsf{P} \star \mathsf{R}\}_{\varepsilon}}{\mathsf{S} \wedge \boxed{\mathsf{P}}^{\iota} \vdash \{\mathsf{Q}\}e\{v.\mathsf{R}\}_{\varepsilon \cdot \iota}}$$

Figure 2.9: Invariant typing rule as well as the *invariant* allocation and opening rule [2].

To allocate a new *invariant*, we use the allocation rule of figure 2.9 on resource P, moving it into to the persistent context so it can be used by multiple threads with the opening rule. E.g., assume that we have the resource $l \hookrightarrow n$, as it is in figure 2.8, and the *invariant* $P = \exists m.n \leq m \wedge l \hookrightarrow m$. We can easily see that $l \hookrightarrow n \implies P$ and by the *later modality* weakening rule, $P \implies \triangleright P$. Assuming now that we have the namespace N, by applying the allocation rule on $\triangleright P$ we stay with $\boxed{P}^{N}$, $\boxed{\exists m.n \leq m \wedge l \hookrightarrow m}^{N}$, in our persistent context. We can later use the opening rule stated in figure 2.9 to open the *invariant*, by inserting $\triangleright P$ in both the precondition and postcondition, and use it on atomic expressions. The namespace ι, in the opening rule, is used to make sure that we are not able to open the same *invariant* twice, without closing it first. Allowing the same *invariant* to be opened twice would lead to inconsistencies in the logic.

$(M,\ \bar{V} : M \to Prop, |-| : M \to M^?,\ (\cdot) : M \times M \to M)$ satisfying:
$\forall\ a,\ b,\ c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c)$
$\forall\ a,\ b.\ a \cdot b = b \cdot a$
$\forall\ a.\ |a| \in M \implies |a| \cdot a = a$
$\forall\ a.\ |a| \in M \implies ||a|| = |a|$
$\forall\ a,\ b.\ |a| \in M \land a \preceq b \implies |b| \in M \land |a| \preceq |b|$
$\forall\ a,\ b.\ \bar{V}(a \cdot b) \implies \bar{V}(a)$
where
$M^? \triangleq M \uplus \{\bot\}$ with $a^? \cdot \bot \triangleq \bot \cdot a^? \triangleq a^?$
$a \preceq b \triangleq \exists\, c \in M.\, b = a \cdot c$
$a \rightsquigarrow B \triangleq \forall\, c^? \in M^?.\, \bar{V}(a \cdot c^?) \implies \exists\, b \in B.\, \bar{V}(b \cdot c^?)$
$a \rightsquigarrow b \triangleq a \rightsquigarrow \{b\}$
if *unital* (uRA), has an element ϵ:
$\bar{V}(\epsilon)$; $\forall a \in M. \epsilon \cdot a = a$; $|\epsilon| = \epsilon$

Figure 2.10: Definition of resource algebra as a 4-tuple [3].

"Invariants allow us to make resources available to different threads, but exactly because they are shared by different threads, the resources governed by them need to be preserved, i.e., the invariant has to be reestablished after each step of execution" [2]. With *ghost states*, we can get more expressive power on our *invariants*, e.g., we could prove the Hoare triple in figure 2.8 for $v \geq n+1$, by letting us give state to our *invariants*! Iris allow us to use *ghost states* via the following proposition: $\boxed{a}^{\gamma}$, which asserts ownership of a piece $a \in M$ of a ghost location γ. Ghost states are represented in separation logic as *Partial commutative monoids* (PCM) which are *Resource algebras* (RA) (figure 2.10) [3].

The composition operator $(\cdot)$ allows one to compose ownership of different threads where its crucial rule is that $\boxed{a \cdot b}^{\gamma} \iff \boxed{a}^{\gamma} \star \boxed{b}^{\gamma}$ (GHOST-OP rule) [19]. We can change our *ghost state* with the use of the *frame-preserving updates*, $a \rightsquigarrow b$, i.e., for any resource $c^?$ compatible with a, it is the case that b is compatible with $c^?$ as well (rule mentioned in figure 2.10). Some of the rules that make it possible to manipulate *ghost* assertions use the *view shift* (or *ghost move*) connective: $P \Rrightarrow_{\epsilon} Q$, meaning that if we have resources that satisfy P we can change the *ghost state* and end with resources that satisfy Q! In the next section, we will discuss better a specific example of a RA that is used to define the time credits' predicate $\$$, the authoritative RA mentioned in Appendix Figure A.2.

2.4 Time Credits and Time Receipts in Iris

An exciting recent development which is relevant for the context of this thesis is the extension of the Iris logic, by Mével, G. et al. [4], to support reasoning on time credits and time receipts. Besides this contribution, they have also proven the correctness of the union find algorithm and the amortized time of the algorithm with the use of Iris extended with these concepts. Even though we can use this extension to prove the logarithmic time complexity of the splay tree algorithm, we have not attempted to do so.

Time credits $\$$ and time receipts ⧗ are logic concepts that allow us to reason about the time execution

of programs. Whereas time credits can be useful to get an upper bound of the execution time of an algorithm, time receipts can be used to reason about the lower bound. Mével, G. et al. [4], introduced the tick pseudo-instruction, a Heap-Lang value with no run-time effect, as well as the tick translation (figure 2.11), a function that transforms a Heap-Lang expression e into $\langle\langle e \rangle\rangle$.

$$\langle\langle e_1(e_2) \rangle\rangle_{tick} = tick(\langle\langle e_1 \rangle\rangle_{tick})(\langle\langle e_2 \rangle\rangle_{tick})$$

Figure 2.11: Tick translation, transforms an Heap-Lang expression by inserting the tick value before any of its expression [4].

$\$: \mathbf{N} \to iProp$	- Time credit predicate
True $\Rrightarrow_\top \$0$	- Zero time credits can be created
$\$(n_1 + n_2) \equiv \$n_1 \star \$n_2$	- Time credits can be combined and split
$tick : Val$	- There is a *tick* pseudo-op
$\{\$1\} tick(v) \{\lambda w.w = v\}$	- *tick* consumes one time credit

Figure 2.12: Time credit $ interface, *TCIntf* [4].

To use Iris with time credits, Iris$^\$$, we first introduce its interface (figure 2.12). This interface comes with the predicate $ that given a natural number n, it returns us n time credits as an Iris proposition, $\$n$, to use as resources. One of the key properties of time credits (TC) is that they can be split and combined $\$(n_1 + n_2) \equiv \$n_1 \star \$n_2$. This property is quite useful whenever, e.g., we have a sequential program $e = e_1; e_2$ and want to give n_1 time credits to e_1 and n_2 time credits to e_2. Finally, the most notorious of the properties is the tick consumption (last) which asserts that the tick pseudo-operation (written in Heap-Lang in Appendix Listing E.1) has the need to consume exactly one TC, $\$1$. It must be noted that time credits, as resources, cannot be duplicated, but they can be thrown away, since Iris is an affine logic, with the weakening rule (figure 2.13).

$$\frac{\mathsf{S} \vdash n_2 \leq n_1 \qquad \mathsf{S} \vdash \$n_2}{\mathsf{S} \vdash \$n1}$$

Figure 2.13: Time credits weakening rule.

In Appendix Figure A.2 it is given the definition of the authoritative monoid that was used to implement the time credit interface. The time credit predicate is defined as, $ = $\lambda n, \boxed{\circ n}^\gamma$, where $\circ n$ is the *fragmental view*. So, if we have $\$n_1 \star \$n_2 \equiv \boxed{\circ n_1}^\gamma \star \boxed{\circ n_2}^\gamma$, by the GHOST-OP rule mentioned in section 2.3 we have, $\boxed{\circ n_1}^\gamma \star \boxed{\circ n_2}^\gamma \equiv \boxed{\circ n_1 \cdot \circ n_2}^\gamma \equiv \boxed{(\bot, n_1) \cdot (\bot, n_2)}^\gamma \equiv \boxed{(\bot, n_1 \cdot n_2)}^\gamma \equiv \boxed{(\bot, n_1 + n_2)}^\gamma$ $\equiv \$(n1 + n2)$, since time credits use the natural RA monoid (Appendix Figure A.2) with the composing operation $(+)$. We have verified the combine and split property of time credits! However, the time credit invariant, which is required to complete the implementation of the time credit interface, is defined as: $\boxed{\exists m, l \hookrightarrow m \star \boxed{\bullet m}^\gamma}^N$, where $\bullet m$ is called the *authoritative view*. We can update them, if

$\text{⧗} : \mathbf{N} \to iProp$	- Exclusive Time receipt (TR) predicate
$\text{⧖} : \mathbf{N} \to iProp$	- Persistent TR predicate
True $\Rrightarrow_\top$ ⧗0	- Zero exclusive TR can be created
$\text{⧗}(n_1 + n_2) \equiv \text{⧗}n_1 \ \star \ \text{⧗}n_2$	- Exclusive TR can be combined and split
$\text{⧖}max(n_1 + n_2) \equiv \text{⧖}n_1 \ \star \ \text{⧖}n_2$	- Persistent TR obey maximum
$\text{⧗}n \Rrightarrow_T \text{⧗}n \ \star \ \text{⧖}n$	- From Exclusive TR we can get Persistent
$\text{⧖}N \Rrightarrow_T$ False	- No machine runs for N time steps
$tick : Val$	- There is a *tick* pseudo-op
$\{\text{⧖}n\}$	- *tick* produces one exclusive receipt and increments an existing persistent receipt
$tick(v)$	
$\{\lambda w.w = v \ \star \ \text{⧗}1 \ \star \ \text{⧖}(n+1)\}$	

Figure 2.14: Exclusive and Persistent Time receipt interface, *TRIntf* [4].

we own both the *fragmental*, $\circ n$, and *authoritative*, $\bullet m$, *view*. The tick consumption rule proceeds to open the mentioned invariant, and updates both *views*! This is done with the *frame preserving update*: $\circ n \cdot \bullet m \rightsquigarrow \circ(n-k) \cdot \bullet(m-k)$, assuming that $k \leq n$, as it is in Appendix Listing A.2 in the auth_nat_update_decr lemma.

Iris' time receipt (TR) interface in figure 2.14 shows some of the properties of exclusive time receipts ⧗ and persistent time receipts ⧖ (also known as duplicable time receipts TRdup). The difference between these is that an exclusive TR can be though as a proof of work, a hard earned receipt, which is additive as shown in the interface, ⧗ $(n_1+n_2) \equiv$ ⧗ $n_1 \ \star$ ⧗ n_2. Whereas persistent TR's are more like the statement that some work has been done, i.e., ⧖ n_1, at least n_1 work as been done, ⧖ n_2, at least n_2 work as been done, concluding that at least $max(n_1, n_2)$ work has been done: $\text{⧖}n_1 \ \star \ \text{⧖}n_2 \equiv \text{⧖}max(n_1 + n_2)$. Since they are persistent, they can be duplicated: $\text{⧖}n \equiv \text{⧖}max(n, n) \equiv \text{⧖}n \ \star \ \text{⧖}n$. The last property shows the tick production rule, which states that tick produces exactly one exclusive TR, ⧗1, and if it has a witness that at least n work has been done in the past, ⧖n, it updates the witness to ⧖(n+1). A small example follows of TC, TR and TRdup tick rules to clarify how they work on a Heap-Lang function.

```
Definition double : val :=
    rec: "double" "n" :=
        if: "n" = #0 then
            #0
        else
            #2 + "double ("n" - #1).
```

Listing 2.2: A Heap-Lang double function as a value. (Functions are values.

```
Lemma double_spec : ∀ n,
    TCTR_invariant nmax -∗
    {{{ TC(3 + 5*n)  ⋆  TR (0)  ⋆  TRdup (n) }}}
    ≪ double #n ≫
    {{{ RET #(2*n) ; TR(3)  ⋆  TRdup((n+1)%nat) }}}.
```

Listing 2.3: The functional and time specification of the double function. TC = Time Credits ; TR = Exclusive Time Receipts ; TRdup = Persistent Time Receipts (duplicable).

Listing 2.3 is a lemma which states the functional and time specification of the Heap-Lang double function written in Listing 2.2. The proof to this lemma is in Appendix Listing B.1, which requires the use of weakest precondition (wp) tactics such as the wp_tick akin to the consumption/production tick rule. The specification is written as a Hoare triple, where in the precondition, we have $3 + 5 \times n$ time credits, $\$(3 + 5 \times n)$, 0 exclusive time receipts, $\blacktriangleright$ 0, and n persistent time receipts, $\boxtimes n$. In the postcondition, we have the functional specification, i.e., the call to double returns $2 \times n$ as a value, RET #(2*n). Briefly, this specification says that double returns the supposed value and does it in 3 to $3 + 5 \times n$ computational steps, thus proving its bounded linear time complexity.

Time receipts, can also be used to prove that certain undesirable events will not occur, if they need an infeasible amount of time to happen. This is done with the assertion mentioned in the interface $\boxtimes N \Rrightarrow_\top$ False (N = number of maximum steps). In the UF algorithm they were used to prove the absence of integer overflows on the machine integers used to represent the rank of the nodes [4], under the assumption that $log_2(log_2(N)) < w - 1$, where w is the size of the machine integer word used to represent the node ranks.

3

A Functional Implementation of Splay Trees

Contents

In this chapter 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 [7] 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 what proofs we added which are not present in Nipkow's proofs and we also show how we have extracted the verified Splay Tree OCaml code for natural numbers as keys.

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 (in Listing 3.1). This binary tree inductive type has two constructors: **L** for leaves, which has no parameters (represented as $< \mid \mid >$), 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 (represented as $< \mid t_1, o.t, t_2 \mid >$). An ordered type $o.t$ must have an equivalence relation (**reflexivity**, **symmetry** and **transitivity**) eq ($=$) and a binary relation lt ($<$) that has the **transitivity** property and if we have $x < y$ then we can not have $x = y$, i.e., $x < y \longrightarrow \neg(x = y)$ [1].

```
Inductive tree :=
  | L
  | T (t1 : tree) (n : o.t) (t2 : tree).
```

Listing 3.1: Binary tree inductive definition

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)[2]. Specifying that a binary tree is searchable required the set data structure XSet (an ordered type set) as shown in Listing 3.2. According to the definition of the predicate, a leaf is a binary search tree and a node is a binary search tree if all the nodes to its left (set_tree l) are of lesser value and all nodes to its right (set_tree r) are of greater value than itself and its children are binary search tree as well. The For_all predicate in the binary search tree predicate, which receives as arguments a predicate P and a set S, is used to state that all elements from set S obey property P.

[1] Coq orderedtype module: https://coq.github.io/doc/v8.9/stdlib/Coq.Structures.OrderedType.html
[2] Coq set module https://coq.inria.fr/library/Coq.MSets.MSetWeakList.html

```
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 3.2: 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 [7] to Galina. This translation consisted in defining the **splay** tree function which we show in Appendix D.5, the **insert** function (shown in Listing 3.3), and the **splay_max** function (shown in Appendix D.4), which is used by the **delete** function (shown in Appendix D.3).

```
Definition insert (a : o.t) (t : tree) : tree :=
  match splay a t with
  |  <| |> => <| <| |>, a, <| |> |>
  |  <| l, a', r |>  =>
     if eq_dec a a' then <| l, a, r |>
     else if lt_dec a a' then <| l, a, <| <|  |> , a', r |> |>
          else <| <| l , a', <|  |> |> , a, r |>
  end.
```

Listing 3.3: Splay tree insert method

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 these lemmas using functional induction, which ``performs case analysis and induction following the definition of a function" [3].

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

3.2.1 Properties Proved

3.2.1.A Set of elements is invariant over the splay function

One of the main lemmas that we have first proven is: **After applying the splay function, the nodes of a binary tree are the same as the ones before the application.** This lemma is stated in Listing 3.4. Initially we were using the syntactic equality (=) in our lemma specification, instead of the semantic equality XSet.Equal (mentioned in Listing 3.4), which we later found to be impossible to prove such lemma. This impossibility is due to the fact that two sets may be the same, but their internal data structures (such as the list that implements the set data structure) may differ.

```
Lemma set_splay : ∀ a t,
      XSet.Equal (set_tree (splay a t)) (set_tree t).
```

Listing 3.4: Set splay lemma

By using functional induction on the splay function in the set_splay lemma mentioned in Listing 3.4, we spawn fourteen (**14**) sub-cases which are the fourteen possible outputs for the splay tree algorithm in Listing D.5. The base cases, i.e., the cases where the output does not call the function itself, are pretty simple. In these cases we only need to apply several set properties such as union commutativity and associativity. For the induction step, i.e., where the result is a call to the splay method, we have as induction hypothesis that the tree that is called by the splay function has the same elements before applying the splay method. With this hypothesis we can easily prove that the set of elements is equal, i.e., we prove the lemma stated in Listing 3.4.

3.2.1.B Binary search tree invariant over the splay function

We have also proven that: **Applying the splay tree function on a binary search tree preserves the binary search tree predicate**. This is stated in Listing 3.5. In order to prove such lemma, we have used once again functional induction and applied a set of tactics to automate the proofs. However, we needed the set_splay lemma shown in Listing 3.4 in order to prove the bst_splay lemma in Listing 3.5. One of the induction step cases, from the bst_splay lemma (Listing 3.5) that we have proven is shown in Equation 3.1.

```
Theorem bst_splay : ∀ a t,
    bst t ⟶ bst (splay a t).
```

Listing 3.5: Binary search tree invariant over the splay method with tree t and key a

$$\begin{aligned}
&\text{splay } a\ br\ = <\mid al,\ a',\ ar\mid > \longrightarrow \\
&(\text{bst } br\ \longrightarrow\ \text{bst } (\text{splay } a\ br)) \longrightarrow \\
&\text{bst } (<\mid cl,\ c,\ <\mid bl,\ b,\ br\mid >\ \mid >) \longrightarrow \text{bst } <\mid\ <\mid\ <\mid cl,\ c,\ bl\mid >,\ b,\ al\mid >,\ a',\ ar\mid >
\end{aligned} \tag{3.1}$$

In order to prove the induction step in Equation 3.1, we have first proven bst $(\text{splay } a\ br)$ with hypothesis $(\text{bst } br\ \longrightarrow\ \text{bst } (\text{splay } a\ br))$ consequently by proving bst br using the hypothesis $(\text{bst } <\mid cl,\ c,\ <\mid bl,\ b,\ br\mid >\ \mid >)$, with the definition of bst in Listing 3.2. Therefore, with bst $(\text{splay } a\ br)$ we have bst $(<\mid al,\ a',\ ar\mid >)$ thanks to hypothesis $(\text{splay } a\ br\ = <\mid al,\ a',\ ar\mid >)$. To prove the conclusion, we needed to prove that all elements from $(<\mid al,\ a',\ ar\mid >)$ are the same as the elements of br which we can do with the lemma set_splay shown in Listing 3.4. With these two mentioned hypotheses, with the definition of binary search tree (bst) (in Listing 3.2) and with some properties over sets, we have proved the case successfully.

3.2.1.C Splay tree correctness of splayed key to root

One of the important properties of the splay tree method is: **Assuming that t is a binary search tree, a key a belongs to tree t if and only if after applying the splay function with key a on tree t it results in a tree with key a as root.** We have proven successfully the statement as the theorem shown in Listing 3.6. We used functional induction over the splay function which, just like for the bst_splay theorem, requires us to prove all fourteen (**14**) possible sub-cases. Some of these cases required the use of the lemma set_splay shown in Listing 3.4 and some properties of the set data structure.

```
Theorem splay_to_root : ∀ t a t',
    bst t ⟶ eq_tree (splay a t) t' ->
    (XSet.In a (set_tree t) ⟺  (∃ l r, eq_tree t' <| l, a, r |>)).
```

Listing 3.6: A key a belongs to a binary search tree t if and only if after splaying key a on tree t, the resulting tree has key a in its root

3.2.1.D Binary search tree invariant over the insert method

In Nipkow's splay tree implementation [7], and in most implementations, the insert method calls the splay method and then adds the node to be inserted as the root (see Nipkow's splay tree insert function in Listing 3.3). We have proven, the theorem shown in Listing 3.7, i.e., that **the binary search tree invariant is preserved after an insert operation**. Using functional induction over the insert method in

the proof of the bst_insert theorem (Listing 3.7) gives us only **4** sub-cases to be proven (instead of **14** in the splay tree method).

```
Theorem bst_insert : ∀ a t,
    bst t ⟶ bst (insert a t).
```

Listing 3.7: Binary search tree invariant over the insert method

Nevertheless, before proving the bst_insert theorem, we have proven two other important lemmas listed in Equation 3.2 (splay_bstL) and Equation 3.3 (splay_bstR). The lemma splay_bstL in Equation 3.2 states that after a splay with key a on tree t, every node from the left branch l has value lower than key a and the lemma splay_bstR in Equation 3.3 is the same, but for the right branch r. Notice that in these equations (3.6, 3.2, and 3.3) we use a semantic predicate called eq_tree (which is an equivalence relation) instead of the syntactic equality (=), due to the fact that two elements may be the same but differ in syntax (similar to what we already discussed for the sets data structure that we have used to specify these lemmas). The **4** sub-cases for the bst_insert theorem are listed and proven informally below.

$$\text{bst } t \longrightarrow \text{eq\_tree}(\text{splay } a\ t)\ (< |\ l,\ e,\ r|>) \longrightarrow \text{XSet.In } x\ (\text{set\_tree } l) \longrightarrow x < a \tag{3.2}$$

$$\text{bst } t \longrightarrow \text{eq\_tree}(\text{splay } a\ t)\ (< |\ l,\ e,\ r|>) \longrightarrow \text{XSet.In } x\ (\text{set\_tree } r) \longrightarrow a < x \tag{3.3}$$

informal proofs of the 4 sub-cases that arise in the proof of the lemma shown in Listing 3.7 on the binary search tree invariant over the insert function (shown in Listing 3.3)

1. bst $< |\ < | \ | >,\ a, < | \ | > |>$: trivial with the binary search tree predicate definition shown in Listing 3.2.

2. splay $a\ t = < |\ l,\ a',\ r| > \longrightarrow$ bst $t \longrightarrow$ eq $a\ a' \longrightarrow$ bst $< |\ l,\ a,\ r| >$: we use the bst_splay lemma in Listing 3.4 over bst t to conclude bst (splay $a\ t$) and consequently with hypothesis splay $a\ t = < |\ l,\ a',\ r| >$ we conclude bst $(< |\ l,\ a',\ r| >)$; because we have eq $a\ a'$ as hypothesis we can conclude bst $< |\ l,\ a,\ r| >$, i.e., the conclusion.

3. splay $a\ t = < |\ l,\ a',\ r| > \longrightarrow$ bst $t \longrightarrow$ lt $a\ a' \longrightarrow$ bst $< |\ l,\ a,\ < |\ < | \ | >,\ a',\ r| >\ | >$: we can conclude bst $(< |\ l,\ a',\ r| >)$ the same way as we did in the case **2.**, then we just use hypothesis lt $a\ a'$, set properties, and the binary search definition to prove the conclusion bst $< |\ l,\ a,\ < |\ < | \ | >,\ a',\ r| >\ | >$. To prove that all nodes from tree l have a value lower than key a, in the conclusion, we used the splay_bstL lemma in Equation 3.2.

4. splay $a\ t = < |\ l,\ a',\ r| > \longrightarrow$ bst $t \longrightarrow$ lt $a'\ a \longrightarrow$ bst $< |\ < |\ l,\ a',\ < | \ | >\ | >,\ a,\ r\ | >$: we used the same proving process as in case **3.**, but had to use splay_bstR (Equation 3.3), instead of

splay_bstL (Equation 3.2), to prove that all nodes from tree r have a value greater than key a.

3.2.1.E Binary search tree invariant over the delete function

The delete function for splay trees uses both the splay tree method and the splay_max method. The splay_max method is equivalent to using the splay method on the node which has the maximum value, as it is reflected in the proven lemma splay_max_eq_splay (shown in Equation 3.5). By splaying the node with maximum value m, the resulting tree (which is a binary search tree, thanks to the proven lemma in Equation 3.4) has no nodes on the right sub-tree as we would expect (proven lemma splay_max_leaf in Equation 3.6) — if we did have such node with value x, then $m < x$ which contradicts the fact that m is the maximum value. We have also proven, in Equation 3.7, that the elements of the tree are preserved after applying the splay_max function.

$$\text{bst } t \longrightarrow \text{bst } (\text{splay\_max } t) \tag{3.4}$$

$$\text{bst } t \longrightarrow (\forall\, x,\ \text{XSet.In } x\ (\text{set\_tree } t) \longrightarrow x < a \vee x == a) \longrightarrow \text{splay\_max } t = \text{splay } a\ t \tag{3.5}$$

$$\text{splay\_max } t = < \mid l,\ a,\ r \mid > \longrightarrow r = < \mid \mid > \tag{3.6}$$

$$\text{XSet.Equal } (\text{set\_tree } (\text{splay\_max } t))\ (\text{set\_tree } t) \tag{3.7}$$

We then proceeded to prove the binary search tree invariant over the splay tree delete function, which corresponds to the theorem shown in Listing 3.8. The theorem states that **applying the delete function in a binary search tree results in a binary search tree**. Just like the insert function, the delete function has **4** possible outputs and therefore, **4** possible sub-cases to prove. Below, we prove informally these sub-cases for the theorem bst_delete shown in Listing 3.8.

```
Theorem bst_delete : ∀ t a,
    bst t ⟶ bst (delete a t).
```

Listing 3.8: Binary search tree invariant over the delete method

informal proof of the 4 sub-cases that arise in the proof of the theorem shown in Listing 3.8 on the binary search tree invariant over the delete function (shown in Appendix D.3)

1. bst $< \mid \mid >$: trivial with the definition of binary search tree

2. splay $a\ t = < \mid l,\ a',\ r \mid > \longrightarrow$ bst $t \longrightarrow$ bst r : with the use of the bst_splay theorem shown in Listing 3.5 over hypothesis bst t we can get bst $(\text{splay } a\ t)$ and with hypothesis splay $a\ t =$

$< \mid l,\ a',\ r \mid >$ we conclude bst $(< \mid l,\ a',\ r \mid >)$; by definition of the binary search tree predicate shown in Listing 3.2, we can conclude bst r.

3. splay $a\ t\ = < \mid l,\ a',\ r \mid > \longrightarrow$ bst $t\ \longrightarrow$ splay_max $l\ = < \mid l',\ m,\ \_x_0 \mid > \longrightarrow$ bst $< \mid l',\ m,\ r \mid >$: with the binary search invariant over the splay method (Listing 3.5) and the splay_max function (Equation 3.4), and the splay_max set invariant in Equation 3.7 we have easily proven this subcase.

4. splay $a\ t\ = < \mid l,\ a',\ r \mid > \longrightarrow$ bst $t\ \longrightarrow$ bst $< \mid l,\ a',\ r \mid >$: follows the same proving process of case **2.**.

3.3 Discussion

We have proven all the lemmas/theorems that Nipkow's has proven related to the splay tree functions in the Coq proof assistant environment. 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 3.9: Function that creates a list of all possible trees generated by list l, where (++) is the append list operation

The splay_insert_delete_star function shown in Listing 3.9 (also written as (splay | insert | delete)* t l) creates a list of all possible trees that could be generated with list of keys l and with the three mentioned splay tree functions. We then constructed a predicate called all_tree_in_list_are_bst which takes a list of trees and states that all the trees in the list are binary search trees. After the splay_insert_delete_star

function and all_tree_in_list_are_bst predicate definition, we have proven the theorem shown in Listing 3.10, which specifies the property mentioned abvoe. It is easy to see that if we start from a tree with no elements (a leaf **L**: ($< | | >$), Listing 3.1) then we do not need the precondition of it being a binary search tree, because by definition it already is (binary search tree predicate shown in Listing 3.2). Therefore we have: (all_tree_in_list_are_bst (splay_insert_delete_star t ($< | | >$))) for any list l.

```
Lemma bst_splay_insert_delete_star : forall t l,
    bst t ⟶ all_tree_in_list_are_bst (splay_insert_delete_star t l).
```

Listing 3.10: Assuming that t is a binary search tree, then any sequence of application of splay tree methods result in a binary search tree

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. In order to extract the code, we have executed the lines shown in Listing 3.11. We have first created a module with our splay tree library that receives an ordered type as input (OrderedTypeEx.Nat_as_OT), in this case the natural numbers, then we export it and use the Coq keyword Extraction as seen in Listing 3.11 to extract the OCaml code to a module named ”SplayTree” with ml extension (SplayTree.ml).

```
Module Nat_Tree := SplayTree(OrderedTypeEx.Nat_as_OT).
Export Nat_Tree.
Recursive Extraction splay.
Extraction "SplayTree" splay.
```

Listing 3.11: Code extraction from Gallina to OCaml for the natural numbers

4

A Pointer-Based Implementation of Splay Trees

Contents

In this chapter we describe how we have modeled the GCC splay tree pointer-based implementation from libgomp [10]. 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 this chapter, we explain how we have proved 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 [10] to heap-lang, $\lambda_{\text{ref,conc}}$.

In general, the translation of the C++ code to heap-lang is straightforward. This can be shown with the example in Listing 4.1 and its respective heap-lang code function in Listing 4.2 for the rotate left operation for splay trees.

```
static inline void rotate_left
    (splay_tree_node *pp, splay_tree_node p, splay_tree_node n)
{
    splay_tree_node tmp;
    tmp = n->right;
    n->right = p;
    p->left = tmp;
    *pp = n;
}
```

Listing 4.1: Rotate left function in the C++ language

```
Definition rotate_left : val :=
     λ  : "pp" "p" "n",
    let: "tmp" := (right_child "n") in
    "n" <- (SOME (value "n", (left_child "n", "p"))) ;;
    "p" <- (SOME (value "p", ("tmp", right_child "p")));;
    "pp" <- "n".
```

Listing 4.2: Rotate left function in heap-lang, $\lambda_{\text{ref,conc}}$

A problem that occurs 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.

Access and mutation of node fields. A splay tree node, in the C++ code, has three fields, a key value k and two pointers for other splay tree nodes, $n1$ and $n2$. In heap-lang, since there is no way to create a type structure in any way, we have translated this structure to tuples. In heap-lang syntax: $\text{Inj}_2(k, (n_1, n_2)))$. The constructor SOMEV in Listing 4.2 is equivalent to Inj_{RV} (Inj_2 for values) and NONEV, that simulates the C++ NULL keyword, is equivalent to $\text{Inj}_{LV}()$ (value Inj_1 () for values).
However, since such value, $\text{Inj}_2(k, (n_1, n_2)))$, does not contain any field names, we can only access these fields with the projection functions that heap-lang offers us: π_1 (projection on first element of tuple, to access the left node) and π_2 (projection on second element of tuple, to access the right node) with semantics: $\pi_i(v1, v2) \rightsquigarrow v_i$. (Recall that the syntax of heap-lang is shown in Figure 2.3). In Listing 4.2, the heap-lang functions value, left_child and right_child use these projection functions to access the respective key values and children. Notice that we also can not change a specific field of the node without changing its full value.

Loops. Although heap-lang allow us to express rich imperative programs, the only way to perform loops is through recursive functions. GCC's implementation uses a "do while" loop that we translated into a recursive function. In this recursive function, we need to have, as parameters, the splay tree pointer and the key which we are searching for. Since the C++ function is of return type void, we need to pass the arguments that suffer mutation by their reference as opposed to their value. This is similar to what happens in the GCC splay algorithm, where we pass as argument a pointer pp that points to the splay tree root p. The output, i.e., the transformed tree, is then pointed by pointer pp. Finally, to create a cycle we just need the function to call itself in order to have the same behaviour as the while loop. The translation of the splay tree "do while" loop of GCC's implementation can be seen in Appendix E.2.

Access addresses. Another problem that occurred during the translation was the fact that we can not access the address of heap-lang variables, unlike in C++ where we can easily access such addresses with the $\&$ operator. Therefore, to solve these problems, we have created variables that hold references to such variables with the operation ref, as seen in the Appendix E.2.

Generic types. Unlike the actual GCC splay tree implementation, we do not use generic types. We simplify our key values to integers, **int**, since we know for sure that integers are an ordered type set with binary relation $<$ and $=$ (explained in section 3, subsection 3.1). In the GCC algorithm, the splay_compare function, the function that is used in the splay tree algorithm to compare node values with the input key, is provided by the user. Therefore, the behaviour of the search algorithm is at the hands of the user, meaning that it might be wrong for some type set, i.e., given a key, the key might be present in the search structure, but due to the bad comparing function splay_compare, the search

algorithm does not find the key. To solve this problem, we would need to assume some properties, as lemmas to be proven by the user, over the key data type just like in the functional implementation. Since our focus is on properties of the data structure operations (and not properties of the tree's content), this simplification does not affect our goals.

4.2 Splay Tree Predicate

In this section, we specify the splay tree predicate which contains the predicates and invariants for a binary search tree as well as the memory model. The chosen predicate for the splay tree data structure was inspired by the work of Mével, G et al. [4] for the union find algorithm.

4.2.1 Domain

The domain predicate D is the set of all the nodes that are present in the binary search tree. In the Iris framework the datatype for pointers (node) is called loc. This domain predicate D has type $loc \longrightarrow$ Prop, which means that for every node x such that $x \in D$, we have $D\ x$. The TLC library, has a module for sets[1] that already includes this predicate datatype and some operations and lemmas over it.

One of the properties that we must guarantee, so that the splay tree algorithm will terminate for every input, is for the domain D to be finite. Therefore we add the finite property to the binary search tree predicate: finite D. In the TLC library, the finite predicate, described in Equation 4.1, states that there exists a list l (a finite object) that for every node $x \in D$ we have that x is contained by that list (mem $x\ l$).

$$\text{finite } D \equiv \exists l, (\forall x, x \in D \longrightarrow \text{mem } x\ l) \tag{4.1}$$

For every tree, we have considered that there exists a node $p \in D$ that is the root of the tree. Note that if there are no nodes in D then there is no root node and the splay tree algorithm is trivial for this case. We call this property rootisindomain and we define it as shown in Equation 4.2.

$$\text{rootisindomain } p\ D \equiv p \in D \tag{4.2}$$

4.2.2 Edges

The edge predicate F', of type $loc \longrightarrow loc \longrightarrow$ Prop, has the information of all the edges that connect each of the nodes. The predicate itself means that if we have $F'\ x\ y$, then we have an edge connection from x to y. However, we also need information related to the orientation ($O = \{LEFT, RIGHT\}$) of the edge since we are dealing with (binary) trees. Therefore we need a predicate F, of type $loc \longrightarrow$

[1] TLC set module: https://github.com/charguer/tlc/blob/master/src/LibSet.v

$loc \longrightarrow O \longrightarrow$ Prop, that given an edge from x to y with orientation $o \in O$, $F\ x\ y\ o$, states that we have a connection from x to y with orientation o.
Nevertheless it seems redundant to have both F' and F since F already has the information of F', $\forall\ x\ y\ o, F\ x\ y\ o \longrightarrow F'\ x\ y$, therefore we only include F in the binary search tree predicate. The tree example shown in Figure 4.1 satisfies both $F\ p\ x\ RIGHT$ and $F\ x\ y\ LEFT$.

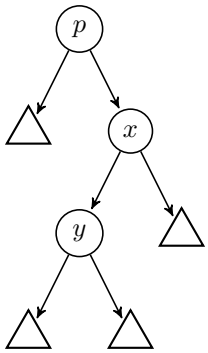

Figure 4.1: A tree where $F\ p\ x\ RIGHT$ and $F\ x\ y\ LEFT$ hold

To make sure that F only holds information related to nodes from domain D we add the confined property to our binary search tree predicate: confined $D\ F$. It is important for our edge set F to only contain information related to nodes from domain D because otherwise the edge set could be infinite and consequently the splay tree algorithm may not terminate. Confining the edge set F to domain D, which is finite, implies that we have a finite maximum of $|D|^2$ number of edges. However, since we are dealing with trees, we have a maximum number of edges of $|D| - 1$. The confined property stated in Equation 4.3, says that for any edge from x to y with orientation o, we have that x and y are in domain D.

$$\text{confined}\ D\ F \equiv \forall\ x\ y\ o,\ F\ x\ y\ o \longrightarrow (x \in D \wedge y \in D) \tag{4.3}$$

We also have to make sure that for all nodes x, we have a maximum of two edges coming from it with different orientations (to make sure that it is a binary tree). For that reason we add the binarytree property, shown in Equation 4.4, to the binary search tree predicate: binarytree F:

$$\text{binarytree}\ F \equiv \forall\ x\ y\ z\ u\ o, y \neq z \longrightarrow\ F\ x\ y\ u \longrightarrow F\ x\ z\ o \longrightarrow u \neq o \tag{4.4}$$

Suppose that we now have three different edges coming from a node x, $F\ x\ y_1\ o_1$, $F\ x\ y_2\ o_2$, $F\ x\ y_3\ o_3$, i.e., $y_1 \neq y_2$, $y_1 \neq y_3$ and $y_2 \neq y_3$. With the mentioned property in Equation 4.4, we can conclude that all o_1, o_2 and o_3 are different from each other, but thanks to the pigeonhole principle we see that two of them must be equal, since for each o_i we have $o_i \in \{LEFT, RIGHT\}$, therefore we cannot have more

than two edges coming from a node x.

The root node p is defined as a node that does not have a parent and that has a path to every node x such that $x \in D$, i.e., it obeys the isroot property: isroot $D\ F\ p$, presented in Equation 4.5. This property states that for all node x and orientation o, if $x \in D$ then we cannot have an edge from x to p (a parent x) with orientation o and we must have a path from p to x. This property guarantees that all nodes from domain D can be accessed by the root node, therefore we exclude the hypothesis that we are dealing with forests.

$$\text{isroot}\ D\ F\ p \equiv \forall\ x\ o, x \in D \longrightarrow \neg(F\ x\ p\ o) \wedge \text{path}\ F\ p\ x \tag{4.5}$$

The path predicate mentioned on Equation 4.5 is an inductive predicate for the edge predicate F that obeys the reflexive and transitive closure over edge set F. The properties of this closure are explicitly mentioned in Equations 4.6, 4.7 and 4.8.

$$\textbf{once} \equiv \forall\ F\ x\ y\ o,\ F\ x\ y\ o \longrightarrow \text{path}\ F\ x\ y \tag{4.6}$$

$$\textbf{reflexivity} \equiv \forall\ F\ x, \text{path}\ F\ x\ x \tag{4.7}$$

$$\textbf{transitivity} \equiv \forall\ Fx\ y\ z, \text{path}\ F\ x\ y \longrightarrow \text{path}\ F\ y\ z \longrightarrow \text{path}\ F\ x\ z \tag{4.8}$$

4.2.3 Value function

In our binary search tree predicate, we also have a value function V, of type $loc \longrightarrow$ int, that holds information about the value of each node in the tree. Although the GCC's splay tree algorithm is generic, i.e., the values that are in the nodes can be of any type, we simplify by only using integers (as explained in Section 4.1).

One of the basic properties of binary search trees is the searchtree property which states that for a specific node, all nodes to its left are of a lesser value and all nodes to its right are of a greater value. We can see the defined property in Equation 4.9. The function op returns the binary relation lower than (lt) when o is equal to $RIGHT$ and greater than (gt) when o is equal to $LEFT$ (e.g., if $F\ x\ y\ RIGHT \wedge$ path $F\ y\ z$ then $Vx < Vy \wedge Vx < Vz$). See Appendix D.1 for op's definition.

$$\text{searchtree}\ F\ V \equiv \forall\ x\ y\ z\ o,\ (F\ x\ y\ o \wedge \text{path}\ F\ y\ z) \longrightarrow op\ o\ (Vx)\ (Vy) \wedge op\ o\ (Vx)\ (Vz) \tag{4.9}$$

The searchtree property defined in Equation 4.9 implies that we cannot have a cycle from a node to itself, i.e., $\forall\ x\ o, \neg(F\ x\ x\ o)$. To prove this, suppose we do have an edge from x to itself with orientation o, $F\ x\ x\ o$. Thanks to the reflexivity property for path defined in Equation 4.7, we have path $F\ x\ x$.

Applying the searchtree property on the conjunction of these two former propositions, we end up with $op\ o\ (V x)\ (V x)$ which is false. Therefore $\forall\ x\ o, \neg(F\ x\ x\ o)$ holds.

4.2.4 Weight function

In our binary search tree predicate, we also have a weight function W that is used to prove lemmas about the potential function of the splay tree algorithm. This function, of type $loc \longrightarrow$ int, maps a node to an arbitrary positive value. Therefore, we include the property mentioned in Equation 4.10 in our binary search tree invariant which says that every output of the weight function is positive.

$$\text{positivefunction}\ W \equiv \forall\ x,\ W\ x\ >\ 0 \tag{4.10}$$

This function W, referred by Sleator, D.D. et al [13] (mentioned in Chapter 2.1, Section 2.1.2), would be used to calculate the potential function of the splay tree and consequently to prove the logarithmic amortized time of the operations performed in this data structure.

4.2.5 Binary search tree invariant

Given the predicates described above: $p\ D$ (section 4.2.1), F (section 4.2.2), V (section 4.2.3), W (section 4.2.4) and invariants: finite (Equation 4.1), rootisindomain (Equation 4.2), confined (Equation 4.3), binarytree (Equation 4.4), isroot (Equation 4.5), searchtree (Equation 4.9), positivefunction (Equation 4.10), we can bring them all together to form the binary search tree invariant mentioned in Equation 4.11.

$$\begin{aligned}
&\text{Inv}\ p\ D\ F\ V\ W \equiv \\
&\text{finite}\ D\ \wedge \\
&\text{rootisindomain}\ p\ D\ \wedge \\
&\text{confined}\ D\ F \wedge \\
&\text{binarytree}\ F \wedge \\
&\text{isroot}\ D\ F\ p \wedge \\
&\text{searchtree}\ F\ V \wedge \\
&\text{positivefunction}\ W
\end{aligned} \tag{4.11}$$

4.2.6 Memory model

The memory of the machine that will run the algorithm is modeled with gmap, a generalized map M, of type $loc \longrightarrow content$, that maps a node from the binary search tree to a content value. A node can have one of four possible values (the content datatype is mentioned as an inductive type in Appendix C.1):

1. NodeB v nl nr : A node which holds value v, left child nl and right child nr
2. NodeL v nl : A node which holds value v and left child nl
3. NodeR v nr : A node which holds value v and right child nr
4. NodeN v : A node which holds value v with no children

With the content inductive type (Appendix C.1), we can translate it easily to a node heap-lang value of the form $\text{Inj}_2(k, (n_1, n_2))$, as mentioned in Section 4.1. The NULL keyword in the C++ language is simulated with the value NONEV which is equivalent to $\text{Inj}_{LV}()$ in the heap-lang notation module.[2] Knowing this, we can easily convert a content datatype to a heap-lang value using the definition shown in Listing 4.3.

```
Definition val_of_content (c : content) : option val :=
  match c with
  | NodeB k v1 v2 => Some (SOMEV(#k, (#v1,    #v2)))
  | NodeL k v1    => Some (SOMEV(#k, (#v1,  NONEV)))
  | NodeR k v2    => Some (SOMEV(#k, (NONEV,  #v2)))
  | NodeN k       => Some (SOMEV(#k, (NONEV,NONEV)))
  end.
```

Listing 4.3: Convert a content data type to an heap-lang value

We then create a memory invariant, Mem, described in Equation 4.12, which uses memory M that has information related to what content is stored by the pointers in memory. This invariant assumes that every node in domain D must have one of the four content values mentioned above and must successfully map these content values to binary search tree components (edges and node values). For example, if we have a pointer x in memory M that points to content $NodeL\ v\ p$, i.e., M !! x = NodeL v p, then we can conclude with the Mem invariant, in Equation 4.12, that there exists edge $F\ x\ p\ LEFT$, there is no edge to the right of x ($\neg(\exists\ y,\ F\ x\ y\ RIGHT)$) and the value in node x is v ($Vx = v$).

[2] Implementation of the notation of the Heap-lang syntax https://plv.mpi-sws.org/coqdoc/iris/iris.heap_lang.notation.html

$$
\begin{aligned}
&\text{Mem } D\ F\ V\ M\ \equiv \\
&\forall x, x \in D \longrightarrow \\
&\text{match (M !! } x\text{) with} \\
&\mid \text{Some (NodeB } v\ p_1\ p_2\text{)}\ =>\ F\ x\ p_1\ LEFT\ \wedge\ F\ x\ p2\ RIGHT\ \wedge\ V\ x\ =\ v \\
&\mid \text{Some (NodeL } v\ p_1\text{)}\ =>\ F\ x\ p_1\ LEFT\ \wedge\ \neg(\exists\ y,\ F\ x\ y\ RIGHT)\ \wedge\ V\ x\ =\ v \\
&\mid \text{Some (NodeR } v\ p_2\text{)}\ =>\ F\ x\ p_2\ RIGHT\ \wedge\ \neg(\exists\ y,\ F\ x\ y\ LEFT)\ \wedge\ V\ x\ =\ v \\
&\mid \text{Some (NodeN } v\text{)}\ =>\ \neg(\exists\ y,\ F\ x\ y\ RIGHT)\ \wedge\ \neg(\exists\ y,\ F\ x\ y\ LEFT)\ \wedge\ V\ x\ =\ v \\
&\mid \text{None}\ =>\ False \\
&end
\end{aligned}
\tag{4.12}
$$

With map M we create an invariant that states that we have ownership of all pointers from M's domain and each one of these points to the heap-lang equivalent of their content value. This can be achieved with the mapsto_M invariant shown in Listing 4.4, where given a generalized map M, it gives us ownership over all pointers in M, each one of these pointing to the heap-lang tuple value equivalent of the content held by the map M (with the use of the val_of_content convert function in Listing 4.3).

```
Definition mapsto_M M : iProp Σ :=
    ([⋆ map] l ⟶ c ∈ M, from_option (mapsto l 1) False (val_of_content c))%I.
```

Listing 4.4: Convert a content data type to a heap-lang value

4.2.7 Splay tree predicate

Finally, the splay tree predicate consists of the three mentioned invariants: the binary search tree invariant Inv (Equation 4.11), the binary search tree memory invariant Mem (Equation 4.12) and the ownership invariant mapsto_M (Listing 4.4). The splay tree invariant puts the edge set F and memory M as existential quantifiers. We put these predicates as quantifiers because the output of the splay tree algorithm modifies the edge set in several ways, depending on the tree itself. This makes it easier to write our specifications. Even though there exists an unique edge set, we will see that the same edge set can be obtained with different manipulation techniques over the initial edge set (Section 4.6).

$$
\begin{aligned}
&\text{ST } p\ D\ V\ W \equiv \\
&\exists\ F\ M, \text{Inv } p\ D\ F\ V\ W\ \star\ \text{Mem } D\ F\ V\ M\ \star\ \text{mapsto\_M } M.
\end{aligned}
\tag{4.13}
$$

4.3 Domain Properties

The domain D consists of all nodes (pointers with type loc) that are present in the tree data structure. In this section we present the descendants function and some of the most important properties related to the domain.

4.3.1 Descendants definition

The descendants definition in Equation 4.14, characterizes all nodes x that have a path from root r, considering an edge set F. It is easy to see that if a node r is in D then the set of descendants of r must be contained by D. This is specified in Equation 4.15. This is due to the isroot property (Equation 4.5), i.e., there exists a path from p, the root of the tree, to any node that belongs to D, therefore there must exist path $F\ p\ r$, and by the transitivity property for paths (Equation 4.8), we also have path $F\ p\ x$.

$$\text{descendants}\ F\ r\ \equiv \{x \mid \text{path}\ F\ r\ x\}. \tag{4.14}$$

$$\forall r,\ \text{Inv}\ p\ D\ F\ V\ W\ \longrightarrow r \in D \longrightarrow \text{descendants}\ F\ r\ \subseteq\ D. \tag{4.15}$$

4.3.2 Domain proofs

We present here two important properties related to the domain.

First, **if there is a path from the binary search tree root p to a node x, then node x must be present in domain D.** This lemma, formulated in Coq as shown in Listing 4.5, can be proven by doing induction on the hypothesis path $F\ p\ x$ and by proving each one of the three sub-cases for: **once** 4.6, **reflexivity** 4.7 and **transitivity** 4.8.

```
Lemma descendants_finite_if_bst : ∀ F x,
    Inv p D F V W ⟶
    path F p x ⟶
    x ∈ D.
```

Listing 4.5: If there is a path from root to some node x, then x belongs to the tree domain

Second, **since a binary search tree has a finite domain as mentioned in Section 4.2.1, and the set D' of the descendants of a node x is a subset of D (stated in Equation 4.15), then D' is also finite.** The lemma that corresponds to the statement is shown in Listing 4.6 and can be proven with the

descendant inclusion lemma mentioned in Equation 4.15 and the finite inclusion rule defined in the TLC library and shown in Equation 4.16.

$$\forall\ E\ F, E \subseteq F \longrightarrow \text{finite}\ F \longrightarrow \text{finite}\ E \tag{4.16}$$

```
Lemma descendants_finite_if_bst : ∀ F x,
      Inv p D F V W ⟶
      x \in D ⟶
      let D' := descendants F x in
      finite (D').
```

Listing 4.6: Finiteness of domain D', the descendants of node x

4.4 Link Properties

As mentioned in Section 4.2.2, a link or edge between two elements x and y with orientation o is depicted by the edge set F as $F\ x\ y\ o$. With the binary search tree invariant we can conclude several lemmas that are useful and used throughout our proofs. Two important lemmas are the **uniqueness of orientation for each edge** and **the uniqueness of edge with same orientation**.

4.4.1 Uniqueness of orientation for each edge

For every binary search tree of root p, such that Inv $p\ D\ F\ V\ W$ (as defined in Section 4.2.5), each edge has one and only one orientation. This is formalized in Equation 4.17. This can be proved easily with the searchtree property (Equation 4.9). Suppose that we have orientation o and orientation u for edge (x, y), then with the searchtree property we would have $op\ o\ (Vx)\ (Vy)$ and $op\ u\ (Vx)\ (Vy)$ (op function definition in Appendix D.1). If o is different from u, we have both $Vx < Vy$ and $Vx > Vy$ at the same time which is a contradiction for integers ($\mathbb{Z}$).

$$\forall\ Fx\ y\ u\ o, \text{Inv}\ p\ D\ F\ V\ W \longrightarrow F\ x\ y\ o \longrightarrow F\ x\ y\ u \longrightarrow u = o \tag{4.17}$$

4.4.2 Uniqueness of edge with same orientation

Given two edges (x, y) and (x, z), if they share the same orientation o, then y must be equal to z. This is formalized in Equation 4.18 and can be proved with the binarytree property (Equation 4.4): if

they are different edges coming from the same node, then they must have different orientation. Suppose they are different, using the binarytree property we end up with the contradiction $o \neq o$, therefore they must be the same edge, i.e., $y = z$.

$$\forall\ F x\ y\ u\ o, \mathsf{Inv}\ p\ D\ F\ V\ W \longrightarrow F\ x\ y\ o \longrightarrow F\ x\ z\ o \longrightarrow y = z \tag{4.18}$$

4.5 Path Properties

We have mentioned already the path inductive type that obeys the three properties: **once** (Equation 4.6), **reflexivity** (Equation 4.7) and **transitivity** (Equation 4.8). This section presents some of the most relevant properties of path in the binary search tree data structure. In particular, it presents the properties: **absence of cycles**, **unicity of parent of an node**, **unicity of path between two nodes** and **path finiteness**.

4.5.1 Absence of cycles in a tree

In a binary search tree, there are no cycles. We already proved before (in Section 4.2.3) that a node cannot point to itself (i.e. $\forall\ x\ o, \neg(F\ x\ x\ o)$). We now want to prove that if a path from x to itself exists, then the size of this path must be 0 (this is formalized in Equation 4.19). Therefore, we have extended our path inductive type so that it can memorize the size of the path between two nodes. For that reason, we have defined a new data type, shown in Appendix C.3, with name path_count. This new inductive type says that for path_count $F\ x\ y\ c$, we have a path from x to y in edge set F of trajectory size c. It has been easily proven that if we have a path from x to y with size c (i.e. path_count $F\ x\ y\ c$) then we have a path from x to y: path $F\ x\ y$.

$$\forall\ F\ x\ c, \mathsf{Inv}\ p\ D\ F\ V\ W \longrightarrow \text{path\_count}\ F\ x\ x\ c \longrightarrow c = 0 \tag{4.19}$$

In order to prove the lemma in Equation 4.19 we can do inversion on the path_count inductive predicate. Afterwards we are required to prove the property for the following three cases: **path_c_refl** (1.), **path_c_step** (2.) and **path_c_trans** (3.) (as defined in Listing C.3):

1. $\mathsf{Inv}\ p\ D\ F\ V\ W \longrightarrow$ path_count $F\ x\ x\ 0 \longrightarrow 0 = 0$: which is trivial.
2. $\mathsf{Inv}\ p\ D\ F\ V\ W \longrightarrow\ F\ x\ x\ o \longrightarrow 1 = 0$: with $F\ x\ x\ o$ in our hypothesis, we can easily prove this by contradiction, since $\forall\ x\ o, \neg(F\ x\ x\ o)$ as mentioned before.

3. Inv $p\ D\ F\ V\ W \longrightarrow$ path_count $F\ x\ y\ n \longrightarrow F\ y\ x\ o \longrightarrow S\ n = 0$: with hypothesis path_count $F\ x\ y\ n$ we conclude path $F\ x\ y$; with the use of the searchtree property (Equation 4.9) on hypothesis $F\ y\ x\ o$ and path $F\ x\ y$ we conclude $op\ o\ (Vy)\ (Vy)$ which is false.

4.5.2 Unicity of parent of a node

In a binary search tree, an element has one and only one parent, except for the root that has none. Therefore, if we have two parents p_1 and p_2 for child x, then p_1 must be equal to p_2 as stated in the lemma shown in Listing 4.7.

```
Lemma only_one_parent_if_bst : ∀ F p1 p2 x o u,
    Inv p D F V W ⟶
    F p1 x o ⟶
    F p2 x u ⟶
    p1 = p2.
```

Listing 4.7: Lemma proving that an element x has only one parent

If an element x had two different parents, $p1$ and $p2$, then we would have link $F\ p1\ x\ o1$ and $F\ p2\ x\ o2$. Using the properties rootisindomain (Equation 4.2), confined (Equation 4.3) and isroot (Equation 4.5), we have both path $F\ p\ p1$ and path $F\ p\ p2$. Since $p1$ and $p2$ are distinct, we can confirm that there are two different paths from p to x. However, the inductive data type path only allow us to compare paths from the endpoint perspective, meaning that it cannot distinguish between the path that passes through element $p1$ and another that passes through element $p2$.

This problem can be solved by storing the path trajectory in some data structure. We opted for lists because they are simple to manipulate and, more importantly, they store the data in an ordered manner which is important when we want to capture the exact trajectory of the path. Furthermore, two trajectories are equal if and only if the lists are equal. Let us call this new data type, that stores the path trajectory, path_memory with the following properties: **once** (Equation 4.20), **reflexivity** (Equation 4.21) and **transitivity** (Equation 4.22). The notation (::) and $(++)$ are respectively the classic cons and append operations for lists[3].

$$\textbf{once} \equiv \forall\ F\ x\ y\ o,\ F\ x\ y\ o \longrightarrow \text{path\_memory}\ F\ x\ y\ [x] \quad (4.20)$$

$$\textbf{reflexivity} \equiv \forall\ F\ x, \text{path\_memory}\ F\ x\ x\ [\,] \quad (4.21)$$

$$\textbf{transitivity} \equiv \forall\ F\ x\ y\ z\ l, F\ x\ y\ o \longrightarrow \text{path\_memory}\ F\ y\ z\ l \longrightarrow \text{path\_memory}\ F\ x\ z\ (x :: l) \quad (4.22)$$

[3]List module coq: https://coq.inria.fr/library/Coq.Lists.List.html

From this new inductive data type we can easily prove that by having a witness l for the path from x to y in edge set F, we also have a path from x to y, i.e., path_memory $F\ x\ y\ l \longrightarrow$ path $F\ x\ y$. Nevertheless, the converse is not true, but we know for sure that if a path exists then there exists a witness, therefore we have the following equivalence: $(\exists\ l,$ path_memory $F\ x\ y\ l) \equiv$ path $F\ x\ y$.

Now suppose we have path_memory $F\ p\ p1\ l1$ for parent $p1$ and path_memory $F\ p\ p2\ l2$ for parent $p2$. With the mentioned properties for path_memory we can conclude path_memory $F\ p\ x\ (l1{+}{+}[p1])$ and path_memory $F\ p\ x\ (l1{+}{+}[p2])$ if p_1 and p_2 are both parents of x. Nevertheless, as we will see in Subsection 4.5.3, in a binary search tree, a path from an element to another has an unique path trajectory, meaning that for x to have two parents, $(l1{+}{+}[p1])$ and $(l2{+}{+}[p2])$ would have to be equal, consequently $p1$ would have to be equal to $p2$, i.e., they would have to be the same parent which contradicts our initial hypothesis.

4.5.3 Unicity of path between two nodes in a binary search tree

In a binary search tree, any path between two nodes has an unique trajectory. The lemma that we have proven, as stated in Listing 4.8, shows that if we have a list l_1 that represents some trajectory between x and y and if we have another list l_2 that also represents some trajectory between x and y, then l_1 and l_2 must be equal.

```
Lemma path_must_be_equal_if_bst : ∀ F x y l1 l2,
    Inv p D F V W ->
    path_memory F x y l1 ->
    path_memory F x y l2 ->
    l1 = l2.
```

Listing 4.8: The path between to nodes in a tree has a unique trajectory

To prove the lemma shown in Listing 4.8, we have done induction on the hypothesis path_memory $F\ x\ y\ l_1$. This leads us to prove three sub-cases (assuming that Inv $p\ D\ F\ V\ W$ is in the hypothesis context) and for each one of these another three sub cases, as shown next.

proof branching for property stating that paths must be equal if binary search tree (Listing 4.8)

1. path_memory $F\ x\ x\ l \longrightarrow [\,] = l$.
   (a) path_memory $F\ x\ x\ [\,] \longrightarrow [\,] = [\,]$ (**reflexivity**, Equation 4.21)
   (b) $F\ x\ x\ o \longrightarrow [\,] = [x]$ (**once**, Equation 4.20)
   (c) $F\ x\ y\ o \longrightarrow$ path_memory $F\ y\ x\ l \longrightarrow [\,] = x :: l$ (**transitivity**, Equation 4.22)
2. $F\ x\ y\ o \longrightarrow$ path_memory $F\ x\ y\ l \longrightarrow [x] = l$.

(a) $F\ x\ x\ o \longrightarrow$ path_memory $F\ x\ x\ [\,] \longrightarrow [x] = [\,]$ (**reflexivity**, Equation 4.21)

(b) $F\ x\ y\ o \longrightarrow$ path_memory $F\ x\ y\ [x] \longrightarrow F\ x\ y\ o' \longrightarrow [x] = [x]$ (**once**, Equation 4.20)

(c) $F\ x\ y\ o \longrightarrow F\ x\ z\ o' \longrightarrow$ path_memory $F\ z\ y\ l \longrightarrow [x] = x :: l$ (**transitivity**, Equation 4.22)

3. $F\ x\ z\ o \longrightarrow$ path_memory $F\ z\ y\ l \longrightarrow (\forall l', \text{path\_memory}\ F\ z\ y\ l' \longrightarrow l = l') \longrightarrow$ path_memory $F\ x\ y\ l' \longrightarrow x :: l = l'$.

   (a) $F\ x\ z\ o \longrightarrow$ path_memory $F\ z\ x\ l \longrightarrow (\forall l', \text{path\_memory}\ F\ z\ x\ l' \longrightarrow l = l') \longrightarrow$ path_memory $F\ x\ x\ [\,] \longrightarrow x :: l = [\,]$. (**reflexivity**, Equation 4.21)

   (b) $F\ x\ z\ o \longrightarrow$ path_memory $F\ z\ y\ l \longrightarrow (\forall l', \text{path\_memory}\ F\ z\ y\ l' \longrightarrow l = l') \longrightarrow$ path_memory $F\ x\ y\ [x] \longrightarrow F\ x\ y\ o' \longrightarrow x :: l = [x]$. (**once**, Equation 4.20)

   (c) $F\ x\ z\ o \longrightarrow$ path_memory $F\ z\ y\ l \longrightarrow (\forall l', \text{path\_memory}\ F\ z\ y\ l' \longrightarrow l = l') \longrightarrow F\ x\ z'\ o' \longrightarrow$ path_memory $F\ z'\ y\ l' \longrightarrow x :: l = x :: l'$. (**transitivity**, Equation 4.22)

To prove sub-case **1.** we did inversion on hypothesis path_memory $F\ x\ x\ l$ which results in three sub-cases (**1.a**), (**1.b**) and (**1.c**): (**1.a**) is trivial with the use of reflexivity on conclusion ; (**1.b**) by contradiction on hypothesis $F\ x\ x\ o$, since $\forall\ x\ o, \neg(F\ x\ x\ o)$ (section 4.2.3) ; and (**1.c**) with hypothesis path_memory $F\ y\ x\ l$ we conclude path $F\ y\ x$ and consequently by applying the searchtree property (Equation 4.9) on $F\ x\ y\ o$ and path $F\ y\ x$, we have $op\ o\ (Vx)\ (Vx)$ which is false, i.e., contradiction.

For sub-case **2.** we follow the same approach by doing inversion on hypothesis path_memory $F\ x\ y\ l$ resulting in three sub-cases (**2.a**), (**2.b**) and (**2.c**): (**2.a**) similar to (**1.b**) ; (**2.b**) trivial with reflexivity on conclusion ; And (**2.c**) if $y = z$, then l must be empty, $[\,]$ (proven in case (**1.**)) and we solve the case by reflexivity on conclusion, just like in case (**2.b**). If $y \neq z$ then we have that o must be different from o' using the binarytree property (Equation 4.4), i.e., y and z branch to different sides. If they branch to different sides, we have with the searchtree property (Equation 4.9): $op\ o\ (Vx)\ (Vy)$ and $op\ o'\ (Vx)\ (Vy)$ which is impossible for $o \neq o'$ since we can not have $a < b \wedge a > b$ for integers (op function in Listing D.1).

Finally, for sub-case **3.** we have our inductive hypothesis in our context,

$$\textit{IH} : (\forall l', \text{path\_memory}\ F\ z\ y\ l' \longrightarrow l = l'),$$

and we do inversion on hypothesis path_memory $F\ x\ y\ l'$. This results in sub-cases (**3.a**), (**3.b**) and (**3.c**): (**3.a**) is similar to case (**1.c**) ; (**3.b**) is similar to case (**2.c**) ; and finally, (**3.c**) if $z' = z$, then we can use the induction hypothesis *IH* on hypothesis path_memory $F\ z'\ y\ l'$ which gives that $l = l'$ and therefore $x :: l = x :: l'$ (conclusion). At last, if $z' \neq z$, then the proof is similar to (**2.c**), i.e., we can conclude that $op\ o\ (Vx)\ (Vy)$ and $op\ o'\ (Vx)\ (Vy)$ for $o \neq o'$ which is a contradiction.

4.5.4 Path finiteness

A path is finite if there exists a natural number that matches the size of its trajectory. **In a binary search tree with a finite domain of nodes D, there can not exist a path with trajectory size bigger than the cardinality of domain D, $|D|$.** If a path would exist with trajectory size bigger than the cardinality of the domain, $|D|$, then in our trajectory we would have at least $|D| + 1$ nodes, which means that some node, in our trajectory, must be repeated. If a node is repeated in our trajectory, then it means that there exists a loop, which contradicts the fact that it is a tree.

```
Lemma path_memory_size_le_card_domain : ∀ p F x y l,
      Inv p D F V W  ⟶
      path_count F x y c  ⟶
      c <= card D.
```

Listing 4.9: Path trajectory size is less or equal than the size of domain of elements

The lemma that we have proven, shown in Listing 4.9, guarantees that **the size of the path trajectory between two nodes, in a binary search tree is at most the cardinality of domain D.** To prove it, we first had to prove that **if we have a binary search tree with edge set F, such that Inv $p\ D\ F\ V\ W$, then there are no duplicated nodes stored in a path trajectory l between two nodes**, i.e. noduplicates l (Equation 4.23). Afterwards, we had to prove that **if a node is in trajectory l, between two nodes of the binary search tree, then all nodes in l are in domain D**, i.e. $\forall\ x, \text{mem}\ x\ l\ \longrightarrow\ x \in\ D$ (Equation 4.24). We then used lemma finite_inv_list_covers_and_card (Equation 4.25) and noduplicates_length_le (Equation 4.26) from the TLC library which enabled us to prove the lemma shown in Listing 4.9. (Remember that mem $x\ l$ means that x is contained by l and length l returns the size of list l as a natural number).

$$\forall F\ x\ y\ l, \text{Inv}\ p\ D\ F\ V\ W\ \longrightarrow \text{path\_memory}\ F\ x\ y\ l \longrightarrow\ \text{noduplicates}\ l. \tag{4.23}$$

$$\forall F\ x\ y\ l, \text{Inv}\ p\ D\ F\ V\ W\ \longrightarrow \text{path\_memory}\ F\ x\ y\ l \longrightarrow\ (\forall\ x, \text{mem}\ x\ l\ \longrightarrow\ x \in\ D). \tag{4.24}$$

$$\forall D, \text{finite}\ D \longrightarrow \exists\ l, ((\forall x,\ x\ \in\ D \longrightarrow \text{mem}\ x\ l) \wedge |D| = \text{length}\ l) \tag{4.25}$$

$$\forall l_1\ l_2, \text{noduplicates}\ l_1 \longrightarrow (\forall x, \text{mem}\ x\ l_1\ \longrightarrow \text{mem}\ x\ l_2)\ \longrightarrow\ \text{length}\ l_1\ \leq\ \text{length}\ l_2 \tag{4.26}$$

Suppose we have path_memory $F\ x\ y\ l$ in our hypothesis context and F is an edge set from a binary search tree, i.e. Inv $p\ D\ F\ V\ W$. With Equation 4.23 we conclude *H1*:(noduplicates l) and with Equation 4.24 we conclude *H2*:$(\forall\ x, \text{mem}\ x\ l\ \longrightarrow\ x \in\ D)$. Then, we can prove $(\forall x, \text{mem}\ x\ l\ \longrightarrow \text{mem}\ x\ l')$ such that l' is the list of all nodes from domain D, with the use of Equation 4.25 and hypothesis *H2*, i.e., if $x \in D$ then x is also in the domain's representative list l'. Finally, with Equation 4.26 we have proven that

length $l \leq$ length l' where length $l' = |D|$, therefore, length $l \leq |D|$. To finalize the proof, we also have proven that if there exists a path from x to y with size c, path_count $F\ x\ y\ c$, then there exists a trajectory l that witnesses such path, path_memory $F\ x\ y\ l$, i.e., path_count $F\ x\ y\ c\ \longrightarrow \exists\ l,$ path_memory $F\ x\ y\ l$.

4.6 Edge Set Manipulation

The edge set F is the predicate mentioned in Section 4.2.2 which defines whether an edge exist in our graph. In this subsection we show how to manipulate such predicate in a way that will be useful to model the behaviour of the rotations that are involved in the splay tree method. We then show some important invariants that will be useful to prove the correctness of the rotate operations.

4.6.1 Operations on edge set

In this section we introduce the main operations for manipulating the edge set F. The operations are the following: add edge (subsection 4.6.1.A), remove edge (subsection 4.6.1.B), update edge (subsection 4.6.1.C), union edge (subsection 4.6.1.D) and elimination of a set of edges (subsection 4.6.1.E).

4.6.1.A Add edge

To add an edge to the edge set predicate F, we simply create a new edge set that joins the edges in predicate F with the new added edge. Remember that an edge must also have an orientation component. Figure 4.2 and Listing 4.10 show, respectively, an example of how an edge is added to the predicate F and the definition of the operation add edge in Gallina.

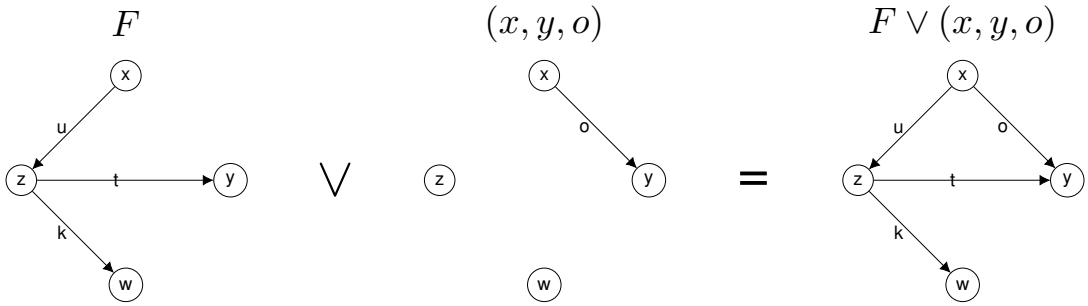

Figure 4.2: Adding edge with connection from x to y with orientation o to predicate F.

```
Definition add_edge F x  y  o :=
    fun x' y' o' => F x' y' o' ∨ (x' = x ∧ y' = y ∧ o' = o).
```

Listing 4.10: Add edge definition written in Gallina

Some properties can be deduced from the add edge operation, such as: **if any path exists in F, then it will for sure exist after adding an arbitrary edge** (Equation 4.27: add edge introduction rule for paths). However, the converse is only true under some hypothesis: **If we add an edge (a, b) and there is no path from the beginning of the path x to a, then the trajectory itself is a witness in the initial edge set F** (Equation 4.28: add edge elimination rule for paths).

$$\forall\ F\ x\ y\ l,\ \text{path\_memory}\ F\ x\ y\ l\ \longrightarrow (\forall\ a\ b\ c,\ \text{path\_memory}(\text{add\_edge}\ F\ a\ b\ c)\ x\ y\ l) \tag{4.27}$$

$$\forall\ Fx\ y\ l\ a\ b\ c, \neg(\text{path}\ F\ x\ a) \longrightarrow \text{path\_memory}(\text{add\_edge}\ F\ a\ b\ c)\ x\ y\ l \longrightarrow \text{path\_memory}\ F\ x\ y\ l \tag{4.28}$$

4.6.1.B Remove edge

In order to remove an edge from our edge set predicate, we need to negate the fact that the edge is in edge set F. For that, we use a conjunction on edge set F with the negated edge that we wish to eliminate. Notice that our remove operation for edges does not need an orientation component, since we assume that for all edge $F\ x\ y\ o$, we can not have o being $RIGHT$ and $LEFT$ at the same time, since it will spawn a contradiction with the searchtree property defined in Equation 4.9. In Figure 4.3 and Listing 4.11, we show, respectively, an example of how an edge is removed from predicate F and the definition of the operation remove edge in Gallina.

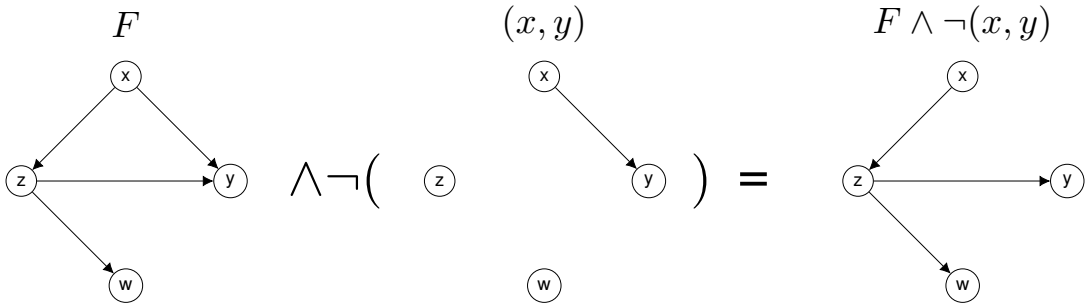

Figure 4.3: Removing edge from x to y in predicate F.

```
Definition remove_edge F x y :=
    fun x' y' o' => F x' y' o' ∧ ¬(x' = x ∧ y' = y).
```

Listing 4.11: Remove edge definition written in Gallina

Some properties for path can also be deduced from the remove edge operation. One of which is: **If any path exists in F after removing an edge, then it will also exist before removing that same edge** (Equation 4.29: remove edge elimination rule for paths). The converse, just like for the add edge

elimination, is not true without some assumptions: **If we remove an edge** (a, b) **and there is no path from some node** x **to** a**, then all paths which start on node** x **still exist** (Equation 4.30: remove edge introduction rule for paths).

$$\forall\ F x\ y\ l\ a\ b, \text{path\_memory}(\text{remove\_edge}\ F\ a\ b)\ x\ y\ l \longrightarrow \text{path\_memory}\ F\ x\ y\ l \tag{4.29}$$

$$\forall\ F x\ y\ l\ a\ b, \neg(\text{path}\ F\ x\ a) \longrightarrow \text{path\_memory}\ F\ x\ y\ l \longrightarrow \text{path\_memory}\ (\text{remove\_edge}\ F\ a\ b)\ x\ y\ l \tag{4.30}$$

4.6.1.C Update edge

In order to modify an element to point to another, in our binary search tree, we need to add one edge and remove up to two edges. Therefore, we have defined another operation to reduce the number of applied add and remove operations (definition in Listing 4.12).

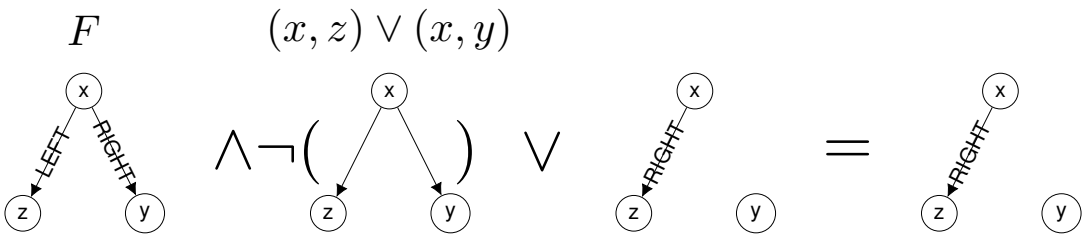

Figure 4.4: Redirecting element x from pointing to y with orientation $RIGHT$, to pointing to z with orientation $RIGHT$, in predicate F.

```
Definition update_edge F x y y'' o'':=
    fun x' y' o' => ¬((x' = x ∧ y' = y) ∨ (x' = x ∧ y' = y'')) ∧ F x' y' o' ∨
                    (x' = x ∧ y' = y'' ∧ o' = o'').
```

Listing 4.12: Update edge definition written in Gallina

As shown in Figure 4.4 we also need to remove edge (x, z) when adding it; if not, then our edge set predicate will have both $F\ x\ z\ LEFT$ and $F\ x\ z\ RIGHT$ which, as we seen with the searchtree property (Equation 4.9), it is a contradiction. We then proved the introduction (Equation 4.31) and elimination (Equation 4.32) rules for paths with the use of the add and remove edge introduction and elimination rule (Equations: 4.27, 4.28, 4.30 and 4.29)

$\forall\ F x\ y\ l\ a\ b\ c\ d, \neg$(path $F\ x\ a$) $\longrightarrow$ path_memory (update_edge $F\ a\ b\ c\ d$) $x\ y\ l \longrightarrow$ path_memory $F\ x\ y\ l$ (4.31)

$\forall\ F x\ y\ l\ a\ b\ c\ d, \neg$(path $F\ x\ a$) $\longrightarrow$ path_memory $F\ x\ y\ l \longrightarrow$ path_memory (update_edge $F\ a\ b\ c\ d$) $x\ y\ l$ (4.32)

4.6.1.D Union edge

The definition of the union of two edge sets is shown in Listing 4.13. It is similar to the add edge operation. From the union edge operation we can deduce easily an introduction rule for paths by induction on trajectory l: **If we have a path from** x **to** y **with trajectory** l **on edge set** $F1$ **or in** $F2$ **then we will have the same path from** x **to** y **with trajectory** l **on the union of both edge sets** (Equation 4.33). In case we want to eliminate such operation: **Assume that** $F1$ **is disjointed from** $F2$**,** $F1 \cap F2 = \varnothing$**, and we have** path_memory (union_edge $F_1\ F_2$) $x\ y\ l$, **therefore we can deduce that either** path_memory $F_1\ x\ y\ l$ **or** path_memory $F_2\ x\ y\ l$ (Equation 4.34). Figure 4.5 shows an example of the union of two edge sets with the use of the logical connective $\vee$ (or).

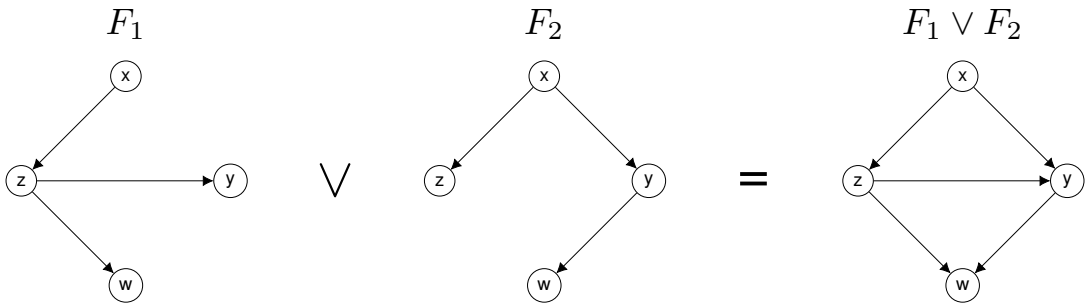

Figure 4.5: Union of edge set F_1 and F_2.

```
Definition union_edge F1 F2 :=
    fun x' y' o' => F1 x' y' o' ∨ F2 x' y' o'.
```

Listing 4.13: Union of two edge sets written in Gallina

$\forall\ F_1\ F_2\ x\ y\ l,$ path_memory $F_1\ x\ y\ l\ \vee$ path_memory $F_2\ x\ y\ l \longrightarrow$ path_memory (union_edge $F_1\ F_2$) $x\ y\ l$ (4.33)

$\forall\ F_1\ F_2\ x\ y\ l, F_1 \cap F_2 = \varnothing \longrightarrow$ path_memory (union_edge $F_1\ F_2$) $x\ y\ l$

$\longrightarrow$ path_memory $F_1\ x\ y\ l \vee$ path_memory $F_2\ x\ y\ l$ (4.34)

4.6.1.E Elimination of a set of edges

While modeling the splay tree rotation operation, mainly the cases with double rotation, we needed to focus on the sub-trees that have as root the child of the root. For that reason, we have created an operation, called remove_edge_that_are_not_in_D, shown in Listing 4.14, that deletes all edges whose endpoints do not belong to a given domain.

With the descendants and remove_edge_that_are_not_in_D definition we can easily retrieve the edge set of some sub-tree F'. Firstly, we get the domain of elements of the sub-tree with root x with descendants, (descendants $Fx) = D'$. Then, we can eliminate all edges that do not belong to that domain with remove_edge_that_are_not_in_D $F\ D' = F'$.

```
Definition remove_edge_that_are_not_in_D F D :=
    fun x' y' o' => (x' ∈ D) ∧ (y' ∈ D) ∧ F x' y' o'.
```

Listing 4.14: Remove edges not in domain definition written in Gallina

In Equation 4.35, we see the rule for elimination on paths for operation remove_edge_that_are_not_in_D. The proof behind this rule is the same as the one for elimination of remove edge in Equation 4.29. For the introduction rule in Equation 4.36, we know that if there exists a path from some node x to another y in F, then it will also exist on edge set (descendants $F\ x$) by definition of remove_edge_that_are_not_in_D.

$$\forall\ Fx\ y\ l\ D, \text{path\_memory}\ (\text{remove\_edge\_that\_are\_not\_in\_D}\ F\ D)\ x\ y\ l \longrightarrow \text{path\_memory}\ F\ x\ y\ l \tag{4.35}$$

$$\forall\ Fx\ y\ l, \text{path\_memory}\ F\ x\ y\ l \longrightarrow \text{path\_memory}\ (\text{remove\_edge\_that\_are\_not\_in\_D}\ F\ (\text{descendants}\ F\ x))\ x\ y\ l \tag{4.36}$$

It is also important to get the edges that do not belong to certain domain. For this reason we have defined the remove_edge_that_are_in_D, in Listing 4.15, which does the task. This operation works as a complement to operation remove_edge_that_are_not_in_D. Nevertheless, not an exact complement, since their union does not complete the edge set F. The proven elimination rule for paths on operation remove_edge_that_are_in_D is similar to the one for remove_edge_that_are_not_in_D (Equation 4.37).

```
Definition remove_edge_that_are_in_D F D :=
  fun x' y' o' =>
    (¬(x' ∈ D) ∧ ¬(y' ∈ D)) ∧ F x' y' o'.
```

Listing 4.15: Remove edges in domain definition written in Gallina

$$\forall\ Fx\ y\ l\ D, \text{path\_memory}\ (\text{remove\_edge\_that\_are\_in\_D}\ F\ D)\ x\ y\ l \longrightarrow \text{path\_memory}\ F\ x\ y\ l \tag{4.37}$$

4.6.2 Child of root is a binary search tree

Considering that p is a root of a binary search tree and if there exists a link from root p to some element x, then x is a root for a binary search tree, that has as its domain elements its descendants. The lemma that has been proven, that reflects the mentioned statement, is written in Listing 4.16. For this proof, we had to prove each property that we have mentioned in Subsection 4.2.5 for this newly created tree.

```
Theorem child_if_inv : ∀ F x o,
    Inv p D F V W ⟶
    F p x o ⟶
    let D' := (descendants F x) in
    let F' := (remove_edge_that_are_not_in_D F D') in
    Inv x D' F' V W.
```

Listing 4.16: The child of the root of a binary search tree, is a binary search tree as well

The proof of lemma shown in Listing 4.16 was divided into **7 sub-proofs** for each property of the binary search tree:

1. finite D' : with the use of the lemma in Listing 4.6 (discussed in Section 4.3.2), we prove that subset D' of D is finite.

2. rootisindomain $x\ D'$: by definition of descendants in Equation 4.14 and rootisindomain in Equation 4.2, this is equivalent to proving path $F\ x\ x$, which is true by reflexivity on path (Equation 4.7).

3. confined $D'\ F'$: with the definition of remove_edge_that_are_not_in_D (Equation 4.14), we have that $F'\ x\ y\ o\ \equiv (x \in D' \wedge y \in D' \wedge F\ x\ y\ o)$. Therefore, we can easily see that $F'\ x\ y\ o\ \longrightarrow\ (x \in D' \wedge y \in D')$ which means that F' is confined by D'.

4. binarytree F' : with binarytree F as hypothesis and its definition in Equation 4.4, to prove binarytree $F'\ \equiv\ \forall\ x\ y\ z\ u\ o, y \neq z \longrightarrow\ F'\ x\ y\ u \longrightarrow F'\ x\ z\ o \longrightarrow u \neq o$, we just need to prove that $\forall x\ y\ o,\ F'\ x\ y\ o\ \longrightarrow\ F\ x\ y\ o$, which is trivial with remove_edge_that_are_not_in_D definition shown in Listing 4.14.

5. isroot $D'\ F'\ x$: using the isroot definition written in Equation 4.5, we have to prove first that there is **no edge from some node in D' to root** x, i.e., $\neg(F'\ e\ x\ o)$. If such edge would exist, then e would belong to D' by the confined property. If e belongs to D' then there exists a path from x to e in F (according to the descendants definition in Listing 4.14), path $F\ x\ e$, however, as seen before, $\forall x\ y\ o,\ F'\ x\ y\ o\ \longrightarrow\ F\ x\ y\ o$. Therefore, we have $F\ e\ x$, which creates a cycle in edge set F, contradicting the fact that we are dealing with a binary search tree. To prove that **there exists a**

path from root x to all other nodes of D' in F', consider some $y \in D'$ as hypothesis, which is equivalent to path $F\ x\ y$. However we can easily prove that path $F\ x\ y \longrightarrow$ path $F'\ x\ y$ with the proven lemma in Equation 4.36, therefore there exists such path in F' to every element y of D'.

6. searchtree $F'\ V \equiv \forall\ x\ y\ z\ o,\ (F'\ x\ y\ o \wedge \text{path}\ F'\ y\ z) \longrightarrow op\ o\ (V x)\ (V y) \wedge op\ o\ (V x)\ (V z)$: to prove it, we only need in our hypothesis $F\ x\ y\ o$ and path $F\ x\ y$. With $F'\ x\ y\ o$ we can deduce $F\ x\ y\ o$ as previously explained, and with path $F'\ x\ y$, we can deduce path $F\ x\ y$ with lemma for elimination of remove_edge_that_are_not_in_D for paths in Equation 4.35.

7. positivefunction W : trivial, by hypothesis, since we do not modify the weight function W.

4.6.3 Join on mutation of sub-tree

Suppose that we have a binary search tree with root p and we have a link from p to some element x with some orientation o. If we modify the sub-tree with root x to another binary search tree with root z, then we can join p with z with orientation o and it will still be a binary search tree. This is due to the fact that all elements that are to orientation o of p are preserved.

```
Theorem join_if_inv : ∀ F F' x z o,
    Inv p D F V W ⟶
    F p x o ⟶
    let D' := (descendants F x) in
    let FC' := (remove_edge_that_are_in_D F D') in
    Inv z D' F' V W ⟶
    let F'' := (add_edge (union_edge F' FC') p z o) in
    Inv p D F'' V W.
```

Listing 4.17: Modifying the roots child sub-tree to another binary search tree preserves the binary search tree properties

We then proceed to prove all properties for this new modified tree:

- finite D : true, by hypothesis.
- rootisindomain $p\ D$: true, by hypothesis.
- confined $D\ F''$: this is equivalent to proving $F''\ x'\ y'\ o' \longrightarrow x' \in D \wedge y' \in D$. To prove this, we unfold F'' until its core F or F'. We start by unfolding add_edge and prove for the case where $x' = p \wedge y' = z \wedge o' = o$, i.e., for $p \in D \wedge z \in D$ which is trivial. Then we are left with the proof obligation (union_edge $F'\ FC'$): For $FC'\ x'\ y'\ o'$ we can conclude $F\ x'\ y'\ o'$ (using the remove_edge_that_are_in_D definition in Listing 4.15) and with the confined property (Equation 4.3),

$x' \in D \wedge y' \in D$; For F', we have, thanks to the confined property (Equation 4.3) that belongs to the invariant Inv $z\ D'\ F'\ V\ W$, $x' \in D' \ \wedge \ y' \in D'$. However, since $D' \subseteq D$, we have, by the inclusion property, also $x' \in D \wedge y' \in D$, therefore we have confined $D\ F''$ as we wanted to prove.

- binarytree F'' : To prove binarytree $F'' \equiv \forall\ x\ y\ z\ u\ o, y \neq z \longrightarrow\ F''\ x\ y\ u \longrightarrow F''\ x\ z\ o \longrightarrow u \neq o$, we start by destructing hypothesis $F''\ x\ y\ u$ and $F''\ x\ z\ o$ repeatedly until we either get to some contradiction in our hypothesis or we get to the core edge set FC' or F'. Notice that FC' and F' are disjointed, therefore it is a contradiction for them to share the same nodes in edges. After getting to the core F' or FC' ($FC' \subseteq F$) we simply use the binarytree property (Equation 4.4) for their respective edge set.

- isroot $D\ F''\ p$: We first prove that **there is no edge from some node x' to root p in F''**: Suppose we have such edge, $F''\ x'\ p\ o'$ for some orientation o'. What we do first is to destruct such hypothesis: The add edge case, $x' = p \wedge p = z \wedge o' = o$, is a contradiction since root p is obviously different than z, $(p \neq z)$; We then are left to prove the union, i.e., for $FC'\ x\ p\ o'$ and $F'\ x\ p\ o'$: For the first case (edge set FC'), we can use the isroot property of edge set F (Equation 4.5) and for edge set F', since there is no edge with node p in F', because of the confined $D'\ F'$ property (Equation 4.3) we conclude that is in fact a contradiction, therefore there is no such edge $F''\ x'\ p\ o'$. Afterwards we are left to prove that **there exists a path from node p to every node x' from domain D in F''**. For this proof we first split it for the case $x' \in D'$ and its complement $\neg(x' \in D')$. For $\neg(x' \in D')$ it is trivial, i.e., we conclude that the respective edge set is FC' and since $FC' \subseteq F$ we only needed to use the isroot property for edge set F (Equation 4.5) to prove it. Finally, for $x \in D'$, that corresponds to the edge set F' and the added edge (p, z, o), we can prove it by induction on the path trajectory.

- searchtree F" V : This proof is fairly simple, since we have both the search property (Equation 4.9) for F' and FC'. However, we had to prove that any path from node p to some other node in D' on edge set F'' obeys the search property which was the difficult part. For this task, we done induction on the path trajectory for F''.

- positivefunction W : trivial, by hypothesis, since we do not modify the weight function.

4.7 Path Find Count Properties

This section describes an inductive data type that models the task of searching for a key in a binary search tree. We call this inductive predicate, path_find_count $F\ V\ p\ x\ z\ n\ s$, and we read it as (on bst): with edge set F, value function V, starting the search for key z on node p, we find x after n steps and with state s. The state s can be either **GOING** if the search algorithm is still in progress or **ENDED** if it

has already terminated. We start off this section by introducing the inductive type in Subsection 4.7.1 and then the termination proof for searching for a key in a binary search tree (Subsection 4.7.2).

4.7.1 Path find count inductive type

The inductive type for path_find_count that we have defined is written in Equation 4.38 with the rules in Equations 4.39, 4.40, 4.41, 4.42 and 4.43. In this subsection, we explain each of the mentioned rules for the path_find_count inductive type.

rules for the inductive type path_find_count

1. If the key value z that we are searching is $z = (V\ x)$ and we are currently in node x, then the search algorithm has ENDED with 0 steps on node x (Equation 4.39).
2. If an edge with orientation o on node x does not exist and we have $(op\ o)\ (Vx)\ (z)$ (Listing D.1) for some key z then the search algorithm stops (ENDED) with 0 steps taken (Equation 4.40).
3. If there exists an edge with orientation o to y on node x and we have $(op\ o)\ (Vx)\ (z)$ we can step into y by taking exactly 1 step and proceed with the search algorithm (GOING) (Equation 4.41).
4. If we have a path_find_count with key z GOING from y to some element t which requires n_1 steps, and if we also have a one step path_find_count with key z GOING (size 1) from x to y, then we have a path_find_count from x to y with size of $1 + n_1$ (Equation 4.42).
5. If we have a path_find_count with key z from some node x to y which requires n_1 steps, and we have a path_find_count with key z that has ENDED from y to itself, then we also have path_find_count which ENDED with key z with key z from x to y (Equation 4.43).

$$\text{Inductive path\_find\_count}: \text{Edgeset} \longrightarrow (\text{loc} \longrightarrow \mathbb{Z}) \longrightarrow \text{loc} \longrightarrow \text{loc} \longrightarrow \mathbb{Z} \longrightarrow \mathbb{N} \longrightarrow \text{state} := \tag{4.38}$$

$$\forall\ F\ V\ x, \text{path\_find\_count}\ F\ V\ x\ x\ (Vx)\ 0\ \text{ENDED} \tag{4.39}$$

$$\forall\ F\ V\ x\ z\ o,\ \neg(\exists\ y,\ F\ x\ y\ o) \longrightarrow (op\ o)\ (V\ x)\ z \longrightarrow \text{path\_find\_count}\ F\ V\ x\ x\ z\ 0\ \text{ENDED} \tag{4.40}$$

$$\forall\ F\ V\ x\ y\ z\ o,\ F\ x\ y\ o \longrightarrow (op\ o)\ (V\ x)\ z \longrightarrow \text{path\_find\_count}\ F\ V\ x\ y\ z\ 1\ \text{GOING} \tag{4.41}$$

$$\forall\ F\ V\ x\ t\ y\ z\ n_1,\ \text{path\_find\_count}\ F\ V\ x\ y\ z\ 1\ \text{GOING} \longrightarrow$$
$$\text{path\_find\_count}\ F\ V\ y\ t\ z\ n_1\ \text{GOING} \longrightarrow \text{path\_find\_count}\ F\ V\ x\ t\ z\ (S\ n_1)\ \text{GOING} \tag{4.42}$$

$$\forall\ F\ V\ x\ y\ z\ n_1,\ \text{path\_find\_count}\ F\ V\ x\ y\ z\ n_1\ \text{GOING} \longrightarrow$$
$$\text{path\_find\_count}\ F\ V\ y\ y\ z\ 0\ \text{ENDED} \longrightarrow \text{path\_find\_count}\ F\ V\ x\ y\ z\ (n_1)\ \text{ENDED} \tag{4.43}$$

4.7.2 Path find count termination proof

In a binary search tree, Inv $p\ D\ F\ V\ W$, searching for key z in edge set F always terminates. To prove the statement, we first have proven that assuming that there is no end to the search algorithm, then we can have for every number of steps a searching path (lemma shown in Listing 4.18).

```
Lemma no_ending_path_then_exists_a_path_with_every_size : ∀ p F z,
      ¬(∃ x n, path_find_count F V p x z n ENDED) ->
      ∀ n, (exists x, path_find_count F V p x z (S n) GOING).
```

Listing 4.18: Assuming that there is no end to the search algorithm, then we have a path for any number of steps

We have easily proven lemma in Listing 4.18 by doing induction on the number of steps. We also have concluded that if we have path_find_count from some node x to other node y in n steps, then we also have path_count from node x to y in n steps. Nevertheless, as we saw in Section 4.5.4 (Listing 4.9), a path can not be longer than the cardinality of the domain of the binary search tree. Therefore, considering that we have a binary search tree such that Inv $p\ D\ F\ V\ W$, and if we consider that there is no end (ENDED) to the search algorithm, then we have a path in our binary search tree longer than the cardinality of the domain by using the lemma in Listing 4.18; this is false for binary search trees, according to the lemma shown in Listing 4.9. Therefore, there must be an end to the binary search algorithm, which we have stated (and proved) as the theorem shown in Listing 4.19.

```
Theorem exists_find_count_path_if_inv : ∀ p F z,
      Inv p D F V W ⟶
      (∃ x n, path_find_count F V p x z n ENDED).
```

Listing 4.19: For every edge set F from a binary search tree, there is an end to the search algorithm

As we will see in the next sections, the theorem shown in Listing 4.19 is important to prove the correctness of the splay tree method algorithm.

4.8 Specification and Correctness of Rotations

The first operations of the splay tree algorithm that we have proven were the rotation operations: rotate_left and rotate_right. In the splay tree algorithm of the GCC implementation, we observe that these rotation operations are used on the root of the tree or on one of the children of the root, therefore we only focus on these. We first have proved that the rotations on the root would preserve the invariants of the binary search tree. Nonetheless, there exist two cases that we must consider: if internal grandchildren exist (Listing 4.20; Figure 4.6) and if internal grandchildren do not exist (Listing 4.21; Figure 4.7), i.e., a node with opposite orientation to the child of the root.

```
Theorem rotate_XI_if_bst : ∀ F x z o,
    Inv p D F V W ⟶
    F p x o ⟶
    F x z (invert_orientation o) ⟶
    let F' := (update_edge F p x z o) in
    let F'' := (update_edge F' x z p (invert_orientation o)) in
    Inv x D F'' V W.
```

Listing 4.20: Rotation on the root with internal grandchildren invariant.

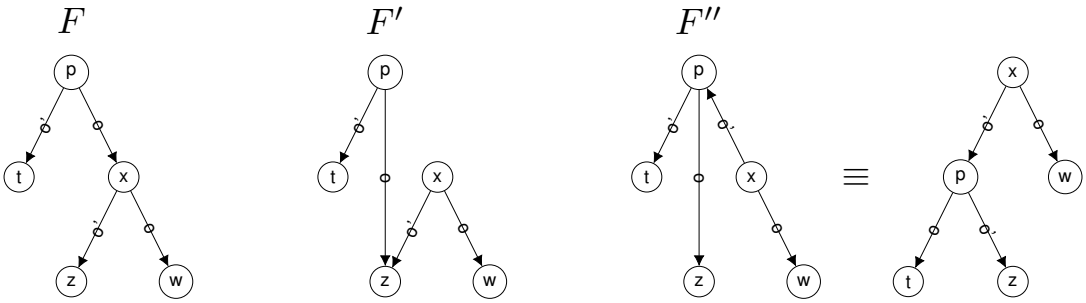

Figure 4.6: Rotate with orientation o on a binary search tree (with root p) with internal grandchild.

```
Theorem rotate_XE_if_bst : ∀ F x o,
      Inv p D F V W ⟶
      F p x o ⟶
      ¬(∃ y, F x y (invert_orientation o)) ⟶
      let F' := (remove_edge F p x) in
      let F'' := (add_edge F' x p (invert_orientation o)) in
      Inv x D F'' V W.
```

Listing 4.21: Rotation on the root with no internal grand children invariant

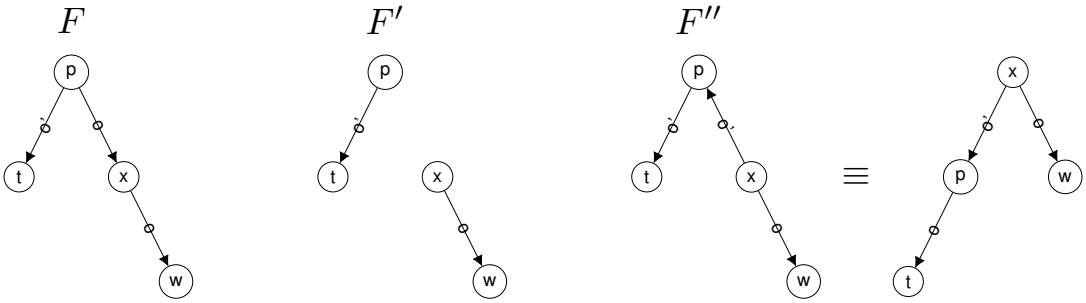

Figure 4.7: Rotate with orientation o on a binary search tree (with root p) with no internal grandchild.

In order to prove the correctness of the rotations performed on the children of the root, we use the child_if_inv theorem shown in Listing 4.16 (Section 4.6.2) to retrieve the sub-tree of the respective child; then we use one of the two mentioned rotation theorems (either rotate_XI_if_bst shown in Listing 4.20 or rotate_XE_if_bst shown in Listing 4.21) to perform the rotation operation, depending on the tree structure. Finally, we use the join_if_inv theorem shown in Listing 4.17 to join the root of the binary search tree with the root of the modified sub-tree (invert_orientation definition in Appendix D Listing D.2).

We then prove the correctness of the rotation operation of the heap-lang code for every possible rotation operation case that can occur in the splay tree method: **8** cases for rotate left on the root node, **8** cases for rotate right on the root node, **32** cases for rotate left on a child node and another **32** cases for rotate right on a child node. All of these cases are the possible memory cases that are allowed during the splay tree method. For example, the rotate right cases on the root are only allowed if there exists a root node with right child, therefore we have two **(2)** possible content values for the root node (*NodeB* and *NodeR* as described in Section 4.2.6) and four **(4)** possible values for the child (*NodeB*, *NodeR*, *NodeL* and *NodeN*), which yields the eight **8** cases.

Proving first the root cases, we can then use these to simplify the children cases on the sub-tree as explained. In Listing 4.22 we show the lemma rotate_left_BB_st, an example of the correctness of the rotate_left operation over the memory where the root pointer p points to content *NodeB* $vp\ p_1\ p_2$ and p_1 points to *NodeB* $vp_1\ p_3\ p_4$ yielding edge set F'' and memory M'. We can then use the same lemma, rotate_left_BB_st, shown in Listing 4.22, and the child_if_inv theorem shown in Listing 4.16 on the child's sub-tree on lemma rotate_left_RBB_st' shown in Listing 4.23. Then, using the join_if_inv theorem shown in Listing 4.17, we can join the root with the new modified sub-tree.

```
Lemma rotate_left_BB_st (pp p p1 p2 p3 p4 : loc) (vp vp1 : Z) :
    let F'' :=  (update_edge (update_edge F p1 p4 p RIGHT) p p1 p4 LEFT) in
    let M' := (<[p:=NodeB (V p) p4 p2]> (<[p1:=NodeB vp1 p3 p]> M)) in
    M !! p  = Some (NodeB vp p1 p2) ⟶
    M !! p1 = Some (NodeB vp1 p3 p4) ⟶
    {{{ pp ⟶ #p ⋆ ⌜Inv p D F V W⌝ ⋆ ⌜Mem D F V M⌝ ⋆ mapsto_M M }}}
      rotate_left #pp #p #p1
    {{{ RET #() ; pp ⟶ #p1 ⋆  ⌜Inv p1 D F'' V W⌝ ⋆
        ⌜Mem D F'' V M'⌝ ⋆
        mapsto_M M' }}}.
```

Listing 4.22: Rotation on the root with internal grandchildren invariant

The lemma that we have proven in Listing 4.23 refers to the part of code from the GCC splay method for the rotation of the children and it proves for the specifica case where the root node p points to value (NodeR $vp\ p_2$), the node p_2 points to value (NodeR $vp_2\ p_3\ p_4$) and p_3 points to (NodeB $vp_4\ p_5\ p_6$).

```
Lemma rotate_left_child_RBB_r_st (pp p p2 p3 p4 p5 p6 : loc) (vp vp2 vp4 : Z) :
    let D' := descendants F p2 in
    let FC' := (remove_edge_that_are_in_D F D') in
    let F' := (remove_edge_that_are_not_in_D F D') in
    let F'' := (update_edge F' p3 p6 p2 RIGHT) in
    let F''' := (update_edge F'' p2 p3 p6 LEFT) in
    let F'''' := (add_edge (union_edge F''' FC') p p3 RIGHT) in
    let M' := (<[p:=NodeR (V p) p3]> (<[p2:=NodeB (V p2) p6 p4]>
    (<[p3:=NodeB (V p3) p5 p2]> M))) in
    M !! p  = Some (NodeR vp p2) ⟶
    M !! p2 = Some (NodeB vp2 p3 p4) ⟶
    M !! p3 = Some (NodeB vp4 p5 p6) ⟶
    {{{ pp ⟶ #p ⋆ ⌜Inv p D F V W⌝ ⋆ ⌜Mem D F V M⌝ ⋆ mapsto_M M }}}
      let: "tmp" := ref #p2 in
      rotate_left ("tmp") #p2 (left_child #p2) ;;
      #p <- SOME (value #p, (left_child #p, !"tmp"))
    {{{ RET #() ; pp ⟶ #p ⋆ ⌜Inv p D F'''' V W⌝ ⋆ ⌜Mem D F'''' V M'⌝ ⋆
        mapsto_M M'}}}.
```

Listing 4.23: Rotation on the child of the root, p_2

4.9 Iterative Rotate Inductive Predicate

To prove the splay tree method, we first modeled the algorithm as an inductive predicate called fw_ir, that we read as forward iterative rotate. The predicate, 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 or ENDED (see C.2). If it is in state ENDED, then the splay tree method terminated successfully, otherwise it is in state GOING which means that it needs to perform another loop of the splay tree iterative method. This inductive predicate, fw_ir, that models the heap-lang GCC's splay tree method present in Listing E.2, holds a total of **15** rules.

```
∀ F V p x y z o,
    (orientation_op o) (V p) z ⟶
    F p x o ⟶
    (orientation_op o) (V x) z ⟶
    F x y o ⟶
    F y t (invert_orientation o) ⟶
    let D' := (descendants F x) in
    let F' := (remove_edge_that_are_not_in_D F D') in
    let FC' := (remove_edge_that_are_in_D F D') in
    let F'' := (update_edge F' x y t o) in
    let F''' := (update_edge F'' y t x (invert_orientation o)) in
    let Fafr := (add_edge (union_edge F''' FC') p y o) in
    let F1 := (update_edge Fafr p y x o) in
    let F2 := (update_edge F1 y x p (invert_orientation o)) in
    fw_ir F V p y z 2 F2 GOING
```

Listing 4.24: Splay tree rule for zigzig

In Listing 4.24 we present one of the rules for the iterative predicate which says that if we have two edges $F\ p\ x\ o$ and $F\ x\ y\ o$ and if by searching for z we end up on node y, then we perform a zig-zig double rotation, starting from the child and then the root yielding the edge set F_2 with tree root y after 2 rotations. However, after performing the double rotation, we go back to the beginning of the cycle, i.e., the state is not yet over, ENDED, therefore we say that the algorithm is still GOING.

```
Lemma fw_ir_inv : ∀ p F x n z F' s,
    Inv p D F V W ⟶
    fw_ir F V p x z n F' s ⟶
    Inv x D F' V W.
```

Listing 4.25: After applying the splay tree method partially or totally (respectively GOING/ENDED) on some binary search tree with root p, we end up with a binary search tree.

The proven lemma shown in Listing 4.25 translates to saying that the iterative rotation algorithm of GCC's splay method preserves the binary search tree invariant, i.e., the updates that occur on edge set F (example shown in 4.24 for F_2) preserve the invariants for the binary search tree invariant (Section 4.2.5).

We also have proven the lemma shown in Listing 4.26, that says that the inductive predicate fw_ir implements the GCC splay tree method loop referred in Listing E.2. As seen in the lemma shown in Listing 4.26, we have fw_ir as hypothesis which says that if we have a tree with root p, edge set F and

value function V, then after n rotations the splay algorithm with search key k will end with edge set F' and root p'. The precondition and post-condition of the lemma describe exactly this behaviour on program splay_tree_while_loop, which is the while loop of the splay tree method. For this proof we have used well-founded recursion [4] and had to use all the rotation cases (two of which are in Listings 4.22 and 4.23) that were mentioned in Section 4.8.

```
Lemma fw_ir_splay_loop_st : ∀ n p D (F:EdgeSet) V W pp p' k F' M,
    fw_ir F V p p' k n F' ENDED ⟶
    Inv p D F V W ⟶
    Mem D F V M ⟶
    {{{ pp ↦ #p ⋆ mapsto_M M }}}
      splay_tree_while_loop #pp #k
    {{{ M', RET #() ; pp ↦ #p' ⋆
        ⌜Inv p' D F' V W⌝ ⋆ ⌜Mem D F' V M'⌝ ⋆
        mapsto_M M'
    }}}.
```

Listing 4.26: The fw_ir inductive predicate mimics the behaviour of the GCC's heap-lang splay tree method.

4.10 Splay Method Specification and Proof

Finally, to conclude the proof of the correctness of the splay tree method, we need to prove that, 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.,

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

To prove the termination lemma, we need to prove that after a constant number of rotations (**4**) the path from the root to the node which we are searching for reduces its level at least by one (represented in Listing 4.27).

[4] https://coq.inria.fr/library/Coq.Init.Wf.html

```
Lemma path_find_count_ir_constant_4 : ∀ p F F' x x' z n,
    Inv p D F V W ->
    path_find_count F V p x z (4+n) ENDED ->
    fw_ir F V p x' z 4 F' GOING ->
    (exists m y, (m < 4+n)%nat /\ path_find_count F' V x' y z m ENDED).
```

Listing 4.27: After four (**4**) rotations, the node for which we are searching for decreases at least one level

We have not succeeded in proving the path_find_count_ir_constant_4 lemma in Listing 4.27, yet we know that it is in fact true. If the node to be splayed is at least four (**4**) steps away from the root, then the splay algorithm needs at least two loops to perform two rotations in each one (either zig-zig or zig-zag).

To informally prove this lemma, we checked how the path size reduces for GCC's zig-zig (Figure 4.8) and zig-zag (Figure 4.9) operation. It can be observed in Figure 4.8 that the zig-zig operation reduces the path if the key is z or the key is in sub-tree z_2, however if it is in sub-tree z_1 the level does not change. In the zig-zag operation in Figure 4.9, we observe that the level of the nodes of sub-tree with root z decrease by at least one (**1**). Therefore, we can exclude the cases where a zig-zag operation is first performed since we know for sure that it will decrease at least one level of the path from the root to the key. This leaves us with the zig-zig cases: As we observe in the zig-zig operation in Figure 4.8, only the sub-tree z_1 does not decrease its level, nevertheless if the key is in sub-tree z_1, then the next operation to be performed will be a zig-zag, reducing the path size to the key by at least one (**1**). We have then informally proven that that after (**4**) rotations the path from the root to the key is reduced.

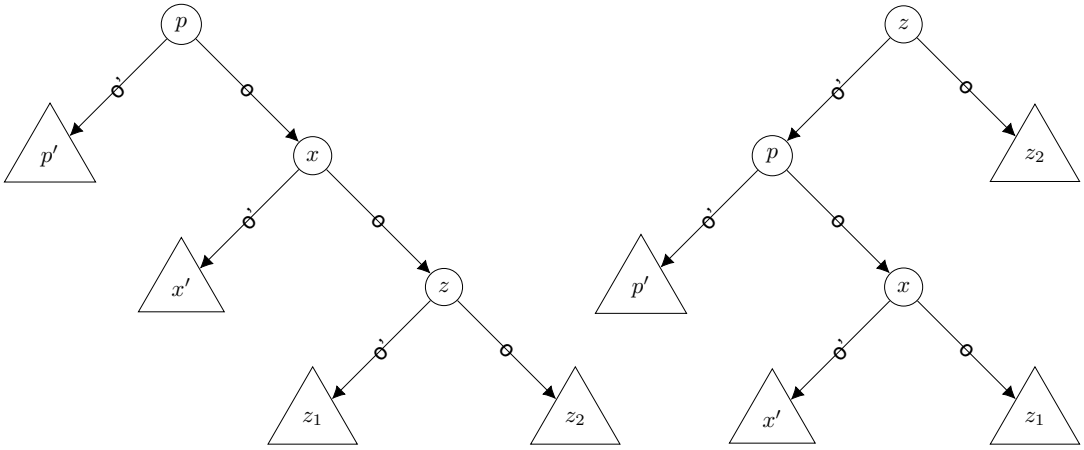

Figure 4.8: Path reduce on GCC's implementation of the zig-zig operation

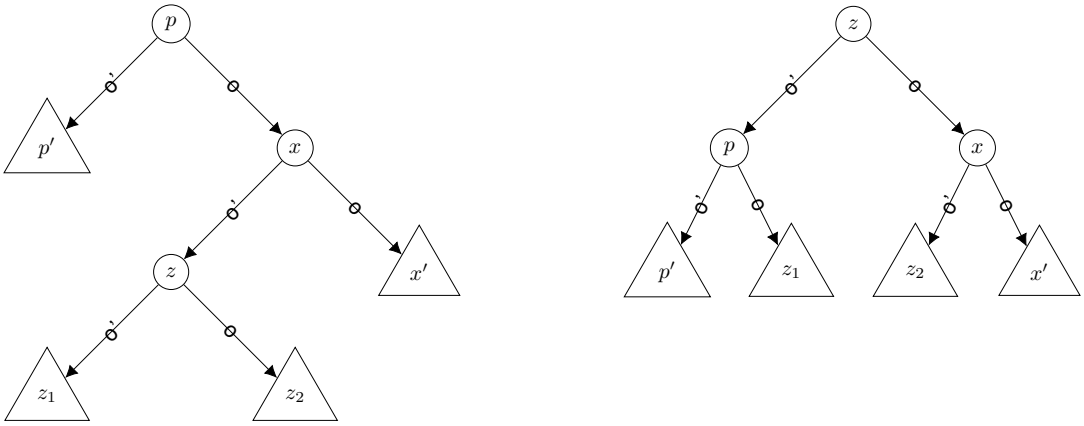

Figure 4.9: Path reduce on GCC's implementation of the zig-zag operation

The proof of lemma path_find_count_ir_constant_4 in Listing 4.27 required the proof of **49** sub-cases, since we have **7** inductive steps for two **2** rotations and therefore $7^9 = 49$ for **4** rotations. Although these sub-cases seem easy to prove as the lemma in itself, the difficulty lays in proving that some paths do not change after the 4 rotations. This difficulty is also due to the operations that we perform in the tree, such as elimination (Section 4.6.1.E) and union (Section 4.6.1.D) of sub-trees, which are not as easy to deal with as other operations such as the add (Section 4.6.1.A) and remove (Section 4.6.1.B).

A way to deal with this problem is to change the manipulation operations that are being performed in the tree to only add and remove operations, nevertheless, by doing this we could not apply the lemmas that have been proven already for the root, meaning that we would have to prove the correctness of the operations over each rotation on the roots children.

Now, assuming that lemma in Listing 4.27 is proven, we have proved that in a binary search tree, if we have an end to a path_find_count for a binary search tree starting from the root of the tree, which we have proven (Listing 4.19), then we can prove that the splay tree method ends, i.e.,

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

Finally, knowing that the splay tree algorithm ends, $\exists\, p'\ n\ F',$ fw_ir $F\ V\ p\ p'\ k\ n\ F'$ ENDED, we can use lemma fw_ir_splay_loop_st to prove the splay tree specification in the lemma shown in Listing 4.28.

```
Lemma splay_st (p pp : elem) (D : set elem) (V W : loc-> Z) (k : Z) :
  {{{ pp ↦ #p ⋆ ST p D V W }}}
    splay_tree_splay #pp #k ;; ! #pp
  {{{ (p' : loc), RET #(p') ; pp ↦ #p' ⋆ ST p' D V W }}}.
```

Listing 4.28: The splay tree method specification lemma

5

Evaluation

Contents

In this chapter, we present some of the metrics for the proofs of correctness of the splay tree algorithm for the functional implementation and for GCC's pointer-based implementation. We also discuss how we have automated some of the proofs and how we could possible have used them better to prove the correctness of the splay tree correctness operations.

5.1 Functional Implementation

The main module, that has all proofs related to the splay tree algorithm methods, is **SplayTree**. The definitions that we encounter in the SplayTree module are the splay tree methods and the predicates mentioned in Section 3.1. The definition of the binary search tree predicate, used to define some of the specifications, uses sets.

The XSet module that we have created is an extension of the MSetWeakList Coq module. We have proved some properties such as union commutativity and associativity over sets, since the XSet module lacks these properties which are important to prove lemmas related to sets in the SplayTree module. Besides this extension, we also have created tactics that automate the proving process by reducing our context hypothesis to simpler hypothesis.

metrics / modules	#lines	#lemmas/theorems	#tactics	#definitions
SplayTree.v **1**	1541	49	2729	20
XSet.v **2**	585	43	955	1
ListTree.v **3**	30	2	40	0
MyTactics.v **4**	24	0	0	0
TOTAL: **4**	2180	94	3724	21

Table 5.1: Metrics of functional implementation of splay trees

	#lines
Code	205
Tactic definition	176
Theorem/Lemmas	1775
TOTAL:	2156

Table 5.2: Number of lines for Code, Tactic definition and Theorem/Lemmas related to the functional splay tree implementation

In table 5.2 we notice that the number of lines for the code of the splay tree algorithm and all theorem/lemmas, that were used to prove the correctness of the code, are respectively roughly 200 and

1800, i.e., we have 9 lines of proofs to 1 line of code.

We notice in Table 5.1 that the module where we used more tactics and where we have written more lines, as expected, was the SplayTree module, where roughly 2700 tactics were used to prove the lemmas/theorems. Nevertheless, these numbers could be bigger if we had not use some automated proving mechanisms that Coq has to offer. We discuss in Section 5.1.1 how we have automated such proofs.

5.1.1 Proof automation

During most of the proofs, we felt the need to automate and simplify some of the proofs. This required the creation and use of several Coq tactics. These tactics greatly reduced the number of lines in our proofs, e.g., to prove the fourteen (**14**) sub-cases of the bst_splay theorem in Listing 3.5 we only used four (**4**) lines of tactics as shown in Listing 5.1.

```
intros a t ; functional induction (splay a t) using splay_ind ; intro ;
simpl in H ; simpl ; auto ; unfold XSet.For_all ; intros ; tree_reduction ;
repeat try (order || logical_reduction || set_reduction ||
            logical_reduction || set_create).
```

Listing 5.1: Sequence of Coq tactics used to prove the bst_splay theorem shown in Listing 3.5

description of the created tactics that we used:

1. **logical_reduction** : simplifies the proving context by recursively eliminating duplicated hypotheses, splitting hypotheses that contain conjunctions, create new hypothesis with the use of *modus ponens*, contradiction whenever false exists as hypothesis and other logical related rules.

2. **set_reduction** : simplifies the proving context or proves the conclusion, related to hypothesis over sets such as: reflexivity over sets (XSet.Equal $S\ S$) ; singleton equivalence ($x\ \in\ \{y\}$ then $x = y$) ; commutativity over union of sets ($S_1\ \cup\ S_2\ \longrightarrow S_2\ \cup\ S_1$).

3. **order** : generated with the functor from the Ordered Type module that we have mentioned in Section 3.1. It tries to automate some of the arithmetic proofs related to the Ordered Type.

4. **tree_reduction** : simplifies some of the tree related proofs such as: a leaf and a singleton node are binary search trees (respectively bst $< \mid \mid >$ and bst $< \mid < \mid \mid >, a, < \mid \mid > \mid >$) ; the **reflexivity**, **symmetry** and **transitivity** properties over the tree equality predicate eq_tree ; after a splay_max application on a tree, the right subtree is a leaf $< \mid \mid >$ (lemma in Listing 3.6).

Besides these tactics, we also used other tactics that are already available with Coq, such as **repeat**, which repeats a tactic until it fails or does not change the proving context, the **try** tactical keyword which allow us to use a certain tactic even if it fails (in case of failure it does not change the proving context). Both of these are present in Listing 5.1.

5.2 Pointer-Based Implementation

In this section we present some of the metrics for GCC's pointer-based splay tree implementation for each of the **32** Coq modules that we have created. We have grouped up these metrics, namely the number of lines, lemmas and theorems, tactics and definitions, into Table 5.3. In this table we observe that we used roughly around 39000 tactics which was reduced due to some created Coq tactics.

metrics / modules	#lines	#lemmas/theorems	#tactics	#definitions
Code.v **1**	200	0	0	14
STorientation.v **2**	170	14	156	4
STpredicate.v **3**	231	0	0	13
STlink.v **4**	564	17	415	0
STdomain.v **5**	346	20	339	3
STpath.v **6**	1305	53	1220	1
STvaluefunction.v **7**	177	4	184	2
STpotential.v **8**	148	6	156	2
STedgeset.v **9**	2520	96	3306	8
STedgesetrotateroot.v **10**	773	21	1101	0
STnumberedges.v **11**	94	3	117	1
STmemory.v **12**	1021	21	1893	1
STproof.v **13**	165	10	177	0
STpathcount.v **14**	1165	47	1871	5
STir.v **15**	980	11	1588	1
STrotaterightcasesroot.v **16**	2033	24	2023	0
STrotateleftcasesroot.v **17**	1983	24	2000	0
STrotaterightcaseschild.v **18**	4112	32	5073	0
STrotateleftcaseschild.v **19**	4033	32	4933	0
STsplaycases_ir.v **20**	64	1	64	0
STsplaycases_irfail.v **21**	233	3	300	0
STsplaycases_ironeleftrotation.v **22**	408	4	580	0
STsplaycases_ironeleftrotationgrandchild.v **23**	283	3	410	0
STsplaycases_ironerightrotation.v **24**	413	4	602	0
STsplaycases_ironerightrotationgrandchild.v **25**	289	3	398	0
STsplaycases_ir_oo_child_nlgc.v **26**	785	3	1277	0
STsplaycases_ir_oo_child_lgc.v **27**	1070	3	2078	0
STsplaycases_ir_ooi_child_nlgc.v **28**	483	3	759	0
STsplaycases_ir_ooi_child_gco.v **29**	652	3	1145	0
STsplaycases_ir_ooi_child_gcoi.v **30**	513	3	778	0
STsplaycases_ir_ooi_child_gcooi.v **31**	675	3	1178	0
STsplay.v **32**	129	3	196	0
TOTAL: **32**	28017	474	39146	55

Table 5.3: Metrics of GCC's splay tree algorithm proofs

The modules for which we have the most part of the definitions (which includes definition of values, predicates, functions and types) are **Code**, which has the translation of GCC's splay tree implementation to heap-lang (mentioned in Section 4.1), and **STpredicate**, which has the definitions of the splay tree

predicate that we have mentioned in Section 4.2. We also define some important inductive types, such as path_find_count and fw_ir in the modules **STpathcount** and **STir**, respectively. In the **STpotential** module we have defined the potential function with the use of the weight function (referred in Section 4.2.4) that is present in the splay tree invariant.

By observing Table 5.3, we notice that the Coq modules that have the proofs related to the correctness of the rotation operation over the children of the root of the tree, **STrotaterightcaseschild** and **STrotateleftcaseschild**, are the modules with more lines and tactics. In each of these two modules, we have **32** lemmas which are the possible combination of content values for the three pointers where a rotation operation on a child can be performed. For example, if we wanted to perform a rotate right in the right child, the root pointer could only hold **2** possible content values, the child pointer could only have **2** possible content values and the grandchild pointer could have **4**, evaluating to **16** cases (example of a case for three pointers is shown in Listing 4.23). We have more **16** more cases for the left child of the root equating in total to **32** cases for the right rotation operation on children. Some automated proving could be done in these proofs, however, since the edge set transformation is different for each memory M configuration, it would probably be harder to try to automate it and therefore we have decided to prove each case individually.

The last mentioned modules in Table 5.3 (which have STsplaycases as suffix) are the proof of the sub-cases of the splay method specification. All of these modules together make almost 6000 lines, and that is the reason why we have initially separated them by each sub-case.

5.2.1 Proof automation

At the end of each module, we usually define a tactic that simplifies further proofs related to lemmas from that module. We consider this a good way to automate our proofs, since we only need to know the name of the tactic and not the name of all lemmas. The proof assistant Coq allows us to create such tactics with the use of the Ltac keyword by doing a match on the proving context. As an example, we show in Listing 5.2 a small part of the created tactic for the link module. This listing shows a rule related to the fact that we can not have an edge from a node to itself (which we have discussed in Section 4.2.3 about the searchtree property).The rule states: If in our proving context we have some hypothesis (h1) Inv $p\ D\ F\ V\ W$, i.e., a binary search tree with root p, domain D, edge set F, value function V and weight function W, and we have also as hypothesis (h2) $F\ x\ x\ o$, a link from a node x to itself, then apply the cant_point_to_itself_if_bst lemma and end the proof.

```
Ltac stlink_tac := match goal with
    (...)
    | h1 : Inv ?p ?D ?F ?V ?W, h2 : ?F ?x ?x ?o |- _ =>
      exfalso ; apply (cant_point_to_itself_if_bst p D V W F x o h1 h2)
    (...)
end.
```

Listing 5.2: Tactic for every link-related lemma

	#lines
Code	200
Tactic definition	223
Theorem/Lemmas	27594
TOTAL:	28017

Table 5.4: Number of lines for Code, Tactic definition and Theorem/Lemmas related to the pointer-based splay tree implementation

Notice in table 5.4 that the number of lines of code **200** to the number of lines of theorem/lemmas **27594** in the pointer-based verification gives us with roughly **138** lines of proofs to **1** line of code. This ratio is way bigger than the **1** to **9** ratio that we have mentioned in Section 5.1 for the functional implementation, showing how much costly it is to prove a pointer-based implementation.

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 [7]. 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 path_find_count_ir_constant_4 lemma (shown in Section 4.10, Listing 4.27) 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. [4] 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 can see, by just looking at the heap-lang function for the splay method in Listing E.2, that we deal with a lot more cases than just **2** (in total, the fw_ir predicate mentioned in Section 4.9 has 15 inductive rules), which substantially increases the verification complexity. Also, in their model, a node in the union find algorithm can only have **2** possible content values. On the other hand, in our case we have that a node can have **4** possible content values, as seen in Section 4.2.6. This means that if we have n nodes in memory, we would have 4^n possible memory content to analyse, instead of 2^n, which still makes a big difference for a small number of nodes to analyse.

Along the proofs we have used some automated proving mechanisms, but they were not enough to automate some of the cases such as the proofs related with the correctness of the double rotations (which involve the rotations on the children). This is a problem when the number of cases is exponential for the number of pointers we are analysing in the memory. When we had to analyse three pointers in the memory for the rotations performed on the children of the root, we had to consider **64** cases (see Table 5.3 for module **STrotaterightcaseschild** and **STrotateleftcaseschild**). If the algorithm would go deeper in the tree in the cycle done in the splay method, then it would be non-viable to do a brute-force approach on each case as we have done.

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 path_find_count_ir_constant_4 lemma, that we have discussed in Section 4.10 Listing 4.27, 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 mentioned in Section 4.28), 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 (in Appendix Listing E.3). However, we felt that the lemma path_find_count_ir_constant_4 in Section 4.10 Listing 4.27 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 seen in Appendix Listing E.4.

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. We have already proven the constant time (**64** computational steps) of the rotation operations for the root (shown in Listing 6.1), which were simple to prove. 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).

```
Lemma rotate_right_RN_st_tc `{!tctrHeapG Σ} (pp p p2 : loc) (vp vp2 : Z) :
    M !! p  = Some (NodeR vp p2) ⟶
    M !! p2 = Some (NodeN vp2) ⟶
    TCTR_invariant nmax -⋆
    {{{ pp ⟶  #p ⋆ ⌜Inv p D F V W⌝ ⋆  ⌜Mem D F V M⌝ ⋆ mapsto_M M ⋆ TC (64) }}}
      ≪rotate_right #pp #p #p2 ≫
    {{{ RET #() ; pp ⟶ #p2 ⋆ ST p2 D V W }}}.
```

Listing 6.1: Rotation constant time complexity proven

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) [20], 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* [2].

Bibliography

[1] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *Journal of Functional Programming*, vol. 28, 2018.

[2] L. Birkedal and A. Bizjak, "Lecture notes on Iris: Higher-order concurrent separation logic," 2018.

[3] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal, "The essence of higher-order concurrent separation logic," in *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*. Berlin, Heidelberg: Springer-Verlag, 2017, p. 696–723. [Online]. Available: https://doi.org/10.1007/978-3-662-54434-1_26

[4] G. Mével, J.-H. Jourdan, and F. Pottier, "Time credits and time receipts in iris," in *European Symposium on Programming*. Springer, 2019, pp. 3–29.

[5] A. Charguéraud and F. Pottier, "Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits," *Journal of Automated Reasoning*, vol. 62, no. 3, pp. 331–365, 2019.

[6] "The Coq proof assistant." [Online]. Available: https://coq.inria.fr/

[7] T. Nipkow, "Splay tree," *Archive of Formal Proofs*, vol. 2014, 2014.

[8] R. M. Stallman *et al.*, "Using the GNU compiler collection," *Free Software Foundation*, vol. 4, no. 02, 2003.

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

[10] ——, "gcc," https://github.com/gcc-mirror/gcc/blob/master/libgomp/splay-tree.c, 2019, [Online; accessed 2019-12-21].

[11] ——, "OpenACC," https://gcc.gnu.org/onlinedocs/gfortran/OpenACC.html, [Online; accessed 2019-12-21].

[12] ——, “gcc,” https://github.com/gcc-mirror/gcc/blob/master/libgomp/oacc-mem.c, 2019, [Online; accessed 2019-12-21].

[13] D. D. Sleator and R. E. Tarjan, “Self-adjusting binary search trees,” *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 652–686, 1985.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[15] T. Nipkow, “Amortized complexity verified,” in *International Conference on Interactive Theorem Proving*. Springer, 2015, pp. 310–324.

[16] D. Le Métayer, “Ace: An automatic complexity evaluator,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 10, no. 2, pp. 248–266, 1988.

[17] A. Charguéraud, “Characteristic formulae for the verification of imperative programs,” in *ACM SIGPLAN Notices*, vol. 46, no. 9. ACM, 2011, pp. 418–430.

[18] J. C. Reynolds, “Separation logic: A logic for shared mutable data structures,” in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 2002, pp. 55–74.

[19] R. Krebbers, A. Timany, and L. Birkedal, “Interactive proofs in higher-order concurrent separation logic,” in *ACM SIGPLAN Notices*, vol. 52, no. 1. ACM, 2017, pp. 205–217.

[20] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan, “The cb tree: a practical concurrent self-adjusting search tree,” *Distributed computing*, vol. 27, no. 6, pp. 393–417, 2014.

A

Iris

$$\frac{S \vdash \{P\} e[v/x] \{u.Q\}}{S \vdash \{\triangleright P\} (\lambda x.e) v \{u.Q\}}$$

Figure A.1: The Hoare triple beta rule reduction. One step later, we can use P as an assertion.

Given a unital RA $(M, \epsilon, V, |\cdot|)$, Auth($M$)
Carrier: $M_{\bot,\top} \times M$
Composition operation:

$$(x, a) \cdot (y, b) = \begin{cases} (y, a \cdot b) & \text{if } x = \bot \\ (x, a \cdot b) & \text{if } y = \bot \\ (\top, a \cdot b) & \text{otherwise} \end{cases}$$

Core:

$$|(x, a)|_{\text{AUTH(M)}} = (\bot, |a|)$$

Valid elements:

$$V_{\text{AUTH(M)}} = \{(x, a) | x = \bot \land a \in V \lor x \in M \land x \in V \land a \preceq x\}$$

and

$$\bullet m = (m, \epsilon) \;\; ; \;\; \circ n = (\bot, n)$$

Figure A.2: Authoritative resource algebra.

```
Section nat.
    Instance nat_valid : Valid nat := λ x, True.
    Instance nat_validN : ValidN nat := λ n x, True.
    Instance nat_pcore : PCore nat := λ x, Some 0.
    Instance nat_op : Op nat := plus.
    (...)
```

Listing A.1: Natural resource algebra components from cmra.v file, using plus (+) as the composition operation.

```
Lemma auth_nat_update_decr (γ : gname) (m n k : nat) :
    (k ≤ n)%nat →
    own γ (●nat m) -∗
    own γ (◯nat n) -∗
    ==> own γ (●nat (m - k)) ∗
        own γ (◯nat (n - k)).
```

Listing A.2: The authoritative Update rule used by the tick pseudo-code instruction to remove one time credit from both *views*.

B

Proofs

```
Lemma double_spec : ∀ n,
TCTR_invariant nmax -∗
{{{ TC(3 + 5*n)  ⋆  TR (0)  ⋆  TRdup (n) }}}
≪ double #n ≫
{{{ RET #(2*n) ; TR(3)  ⋆  TRdup((n+1)%nat) }}}.
Proof.
    intros.
    iIntros "#Htickinv !#" (Phi) "TC Post".
    iInduction n as "IH" forall (Phi).
    + wp_tick_rec. wp_tick_op. wp_tick_if. iApply "Post".
      done.
    + wp_tick_rec. wp_tick_op. wp_tick_if.
      replace (5 * S n')%nat with (5 + 5 * n')%nat. lia.
      wp_tick_op. assert (Haux : S n' - 1 = n'). lia.
      rewrite Haux. iDestruct "TC" as "[T1 TC]".
      wp_apply ("IH" with "TC").
      iIntros. wp_tick_op.
      assert (Hpost : 2 * S n' %nat = 2 + 2 * n' %nat. lia.
      rewrite Hpost. iApply "Post". done.
Qed.
```

Listing B.1: Proof of the double Heap-Lang function specification in the Coq proof assistant.

C

Definitions of inductive types

```
Inductive content :=
| NodeB : Z -> elem -> elem -> content
| NodeL : Z -> elem -> content
| NodeR : Z -> elem -> content
| NodeN : Z -> content.
```

Listing C.1: The content of a node in the splay tree algorithm

```
Inductive state :=
| ENDED
| GOING.
```

Listing C.2: A state of an algorithm can either be $ENDED$ or $GOING$

```
Inductive path_count : EdgeSet -> elem -> elem -> nat -> Prop :=
    path_c_refl  : forall F x, path_count F x x 0
  | path_c_step  : forall F x y o, F x y o -> path_count F x y 1
  | path_c_trans : forall F x y z o n,
                   path_count F x y n ->
                   F y z o ->
                   path_count F x z (S n).
```

Listing C.3: The path_count inductive type written in Gallina

D

Definitions of functions

```
Definition op (o : orientation) :=
    match o with
    | LEFT  => Z.gt
    | RIGHT => Z.lt
    end.
```

Listing D.1: Operation orientation, returns "$<$" if $RIGHT$ and "$>$" if $LEFT$

```
  Definition invert_orientation (o : orientation) :=
    match o with
    | LEFT => RIGHT
    | RIGHT => LEFT
    end.
```

Listing D.2: Invert function for orientation

```
Definition delete (a : o.t) (t : tree) : tree :=
    match splay a t with
    |  <| |> => <| |>
    |  <| l, a', r |>  =>
       if eq_dec a a' then
         match splay_max l with
         | <| |> => r
         | <| l', m, r' |>  => <| l', m, r |>
         end
       else
         <| l, a', r |>
  end.
```

Listing D.3: Splay tree delete method

```
Fixpoint splay_max (t : tree) : tree :=
    match t with
    | <| |> => <| |>
    | <| l, b, <| |> |> => <| l, b, <| |> |>
    | <| l, b, <| rl, c, rr |> |> =>
        match splay_max rr with
        | <| |> => <| <| l, b, rl |> , c, <| |> |>
        | <| rrl, x, xa |> => <| <| <| l, b, rl |> , c,  rrl |>, x , xa |>
        end
    end.
```

Listing D.4: Splay tree splay max method

```
Fixpoint splay (a : o.t) (t : tree) : tree :=
    match t with
    | <| |> => <| |>
    | <| cl , c, cr |> =>
      if eq_dec a c then <| cl , c, cr |>
      else if lt_dec a c then match cl with
                | <| |> => <| cl, c, cr |>
                | <| bl, b, br |> =>
                  if eq_dec a b then <| bl, b, <| br, c, cr |> |>
                  else if lt_dec a b then match splay a bl with
                            | <| |> => <| bl, b, <| br, c, cr |> |>
                            | <| al, a', ar |> =>
                            <| al, a', <| ar, b, <| br, c, cr |> |> |>
                            end
                       else match splay a br with
                            | <| |> =>  <| bl, b, <| br, c, cr |> |>
                            | <| al, a', ar |> =>
                            <| <| bl, b, al |> , a', <| ar, c, cr |> |>
                            end
                end
           else match cr with
                | <| |> => <| cl, c, cr |>
                | <| bl, b, br |> =>
                  if eq_dec a b then <| <| cl, c, bl |>, b, br |>
                  else if lt_dec a b then match splay a bl with
                            | <| |> => <| <| cl, c, bl |>, b, br |>
                            | <| al, a', ar |> =>
                            <| <| cl, c, al |>, a', <| ar, b, br |> |>
                            end
                       else match splay a br with
                            | <| |> => <| <| cl, c, bl |>, b, br |>
                            | <| al, a', ar |> =>
                            <| <| <| cl, c, bl |>, b, al |> , a', ar |>
                            end
                end
    end.
```

Listing D.5: Functional splay tree method implemented in Gallina

E

Heap-lang code

```
tick_c ≜ rec self(x) =
         let k = !c in
         if k = 0 then oops()
         else if CAS(c,k,k-1) then x else self(x)
```

Listing E.1: Tick function in Heap-Lang.

```
Definition splay_tree_while_loop : val :=
    rec: "func" "sp'" "key" :=
    let: "n" := !"sp'" in
    let: "cmp1" := splay_compare "key" (value ("n")) in
    if: "cmp1" = #0 then
      #()
    else (
```

```
let: "c" := (
     if: "cmp1" < #0 then
       ( left_child  ("n") )
     else
       ( right_child ("n") )
   ) in
if: ("c" = NONE) then
  #()
else
  let: "cmp2" := splay_compare "key" (value "c") in
  if: ("cmp2" = #0)
      || (("cmp2" < #0) && ((left_child  "c") = NONE))
      || ((#0 < "cmp2") && ((right_child "c") = NONE)) then
    (
      if: ("cmp1" < #0) then
        rotate_left  "sp'" "n" "c"
      else
        rotate_right "sp'" "n" "c"
    ) ;; #()
  else (
      if: ("cmp1" < #0) && ("cmp2" < #0) then
        (let: "tmp" := (ref "c") in
         rotate_left ("tmp") "c" (left_child "c") ;;
         "n" <- SOME (value "n", (!"tmp", right_child "n"))) ;;
        rotate_left ("sp'") "n" (left_child "n")
      else if: (#0 < "cmp1") && (#0 < "cmp2") then
             (let: "tmp" := (ref "c") in
              rotate_right ("tmp") "c" (right_child "c") ;;
              "n" <- SOME (value "n", (left_child "n", !"tmp"))) ;;
             rotate_right ("sp'") "n" (right_child "n")
           else if: ("cmp1" < #0) && (#0 < "cmp2") then
                  (let: "tmp" := (ref "c") in
                   rotate_right ("tmp") "c" (right_child "c") ;;
                   "n" <- SOME (value "n", (!"tmp", right_child "n"))) ;;
                  rotate_left ("sp'") "n" (left_child "n")
                else if: (#0 < "cmp1") && ("cmp2" < #0) then
                       (let: "tmp" := (ref "c") in
                        rotate_left ("tmp") "c" (left_child "c") ;;
```

```
                                "n" <- SOME (value "n", (left_child "n", !"tmp"))) ;;
                               rotate_right ("sp'") "n" (right_child "n")
                             else
                               #()
            ) ;;
               "func" "sp'" "key"
      ).
```

Listing E.2: Splay tree method in heap-lang.

```
Definition splay_tree_insert : val :=
  λ: "sp" "n",
  splay_tree_splay "sp" (value "n") ;;
  let: "comparison" := (
       if: !"sp" ≠ NONE then
         splay_compare (value (!"sp")) (value "n")
       else
         #0
     ) in
  if: (!"sp" ≠ NONE) && ("comparison" = #0) then
    #()
  else (
    if: (!"sp" = NONE) then
      "n" <- SOME(value "n", (NONE, NONE))
    else if: ("comparison" < #0) then
      "n" <- SOME(value "n", (!"sp",right_child "n")) ;;
      "n" <- SOME(value "n", (left_child "n", right_child (left_child "n"))) ;;
      (left_child "n") <-
      SOME(value (left_child "n"), (left_child (left_child "n"),NONE))
    else
      "n" <- SOME(value "n", (left_child "n", !"sp")) ;;
      "n" <- SOME(value "n", (left_child (right_child "n"), right_child "n")) ;;
      (right_child "n") <-
      SOME(value (right_child "n"), (NONE,right_child (right_child "n")))
  ) ;;
  "sp" <- "n"
```

Listing E.3: Splay insert method in heap-lang.

```
Definition splay_tree_remove : val :=
  λ: "sp" "key",
  splay_tree_splay "sp" "key" ;;
  if: (!"sp" ≠ NONE) && ((splay_compare (value(!"sp")) "key") = #0) then (
    let: "left" := ref (left_child (!"sp")) in
    let: "right" := ref (right_child (!"sp")) in
    if: (!"left" ≠ NONE) then (
      "sp" <- !"left" ;;
      if: (!"right" ≠ NONE) then
        (
          rec: "func" "left'" :=
            if: (right_child (!"left'") ≠ NONE) then
              "left" <- (right_child (!"left")) ;;
              "func" "left'"
            else #()
        ) "left" ;;
        !"left" <- SOME(value !"left", (left_child !"left", !"right"))
      else #()
    )
    else
      "sp" <- !"right"
  )
  else #()
```

Listing E.4: Splay remove method in heap-lang.