TU WIEN Informatics

Verifizierung von Rank-Balanced-Bäumen mit LiquidHaskell

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Alexander Genser, BSc.
Matrikelnummer 01326384

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assoc. Prof. Dipl.-Math. Dr.techn. Florian Zuleger

Wien, 30. Juni 2024

Alexander Genser	Florian Zuleger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Verified Rank-Balanced trees using LiquidHaskell

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Alexander Genser, BSc.
Registration Number 01326384

to the Faculty of Informatics

at the TU Wien

Advisor: Assoc. Prof. Dipl.-Math. Dr.techn. Florian Zuleger

Vienna, June 30, 2024

Alexander Genser	Florian Zuleger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Alexander Genser, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Overview of Tools used“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben Prompts und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 30. Juni 2024

Alexander Genser

Danksagung

Ich möchte meinen tiefsten Dank an zwei wesentliche Unterstützer aussprechen, ohne die diese Arbeit nicht möglich gewesen wäre. Zuerst möchte ich meiner Frau danken, dessen unerschütterliche Unterstützung und Verständnis in schwierigen Zeiten nicht nur Trost, sondern auch die Kraft gab, die herausfordernden Phasen dieser Forschungsarbeit zu überstehen. Deine Geduld und Ermutigung waren für mich von unschätzbarem Wert.

Ebenso bin ich meinem Professor zutiefst dankbar, dessen Anleitung entscheidend war, um meine Beweise und meinen Gesamtansatz zu verfeinern. Dein aufschlussreiches Feedback und die beharrliche Hilfe bei der Reformulierung der kritischen Komponenten meiner Arbeit haben die Qualität meiner Thesis erheblich verbessert.

Acknowledgements

I want to extend my deepest gratitude to two key supporters without whom this thesis would not have been possible. First, I must thank my partner, whose unwavering support and understanding during the darker times provided solace and the strength to persevere through the challenging phases of this research. Your patience and encouragement have been invaluable to me.

I am also profoundly grateful to my professor, whose expert guidance was instrumental in refining my proofs and overall approach. Your insightful feedback and persistent help in reformulating the critical components of my work have significantly enhanced the quality and rigour of my thesis.

Thank you both for your immense contributions to my academic journey.

Kurzfassung

Diese Arbeit präsentiert eine umfassende Überprüfung des WAVL-Baums (weak AVL), einer Variante des klassischen AVL-Baums, der mehr Flexibilität bei den Balancierungsoperationen ermöglicht. Die Studie konzentriert sich auf drei kritische Aspekte des WAVL -Baums: Terminierung, funktionale Korrektheit und amortisierte Komplexitätsanalyse seiner Ressourcennutzung. Die funktionale Korrektheit wird durch den Einsatz von LiquidHaskell bewertet, indem demonstriert wird, dass der WAVL -Baum seine Operationen, Einfügen und Löschen, korrekt implementiert und dabei alle Eigenschaften beibehält. Eine detaillierte amortisierte Komplexitätsanalyse wird durchgeführt, um den Ressourcenverbrauch des WAVL -Baums über eine Reihe von Operationen für die Ausbalancierung des Baums formal zu beweisen und somit seine Effizienz und Leistungsmerkmale zu bestätigen. Diese Überprüfung festigt nicht nur die Position des WAVL -Baums als robuste Datenstruktur, sondern trägt auch erheblich zum Bereich der Datenstrukturen bei, indem sie fortgeschrittene Verifikationstechniken mit LiquidHaskell detailiert darlegt, die Zuverlässigkeit und Effizienz gewährleisten.

Abstract

This thesis presents a comprehensive verification of the weak AVL tree, a variant of the classic AVL tree that allows for more flexibility in balancing operations. The study focuses on three critical aspects of the WAVL tree: termination, functional correctness, and amortized complexity analysis of its resource usage. Functional correctness is assessed by demonstrating, through LiquidHaskell, that the WAVL tree accurately implements its specified operations—insertions, deletions, and searches—while maintaining all weak AVL properties. An in-depth amortized complexity analysis is conducted to formally verify the WAVL tree's resource usage over sequences of operations, thereby confirming its efficiency and performance characteristics. This verification solidifies the WAVL tree as a robust data structure and contributes significantly to the field of data structures by detailing advanced verification techniques with LiquidHaskell that ensure reliability and efficiency.

Contents

CHAPTER 1

Introduction

Formal verification of data structures and proving algorithms functionally correct gained popularity because of some sensational bugs in real-world applications.

Besides functional correctness, good performance is an algorithm's next most important quality. Showing worst-case performance for algorithms via an amortised complexity analysis has proven to be a good tool for showing average performance for algorithms and data structures. However, formalising them and proving actual code was hard for a long time. Because most algorithms and, in that sense, programming languages could be influenced by implicit contexts, finding a system for theorem proving inside an existing programming language proved to be rather challenging.

LiquidHaskell is one of the younger contenders in the theorem provers space. It shows a good overall tool set for proving lemmas about data structures while still being written in the Haskell programming language.

In this work, we successfully used LiquidHaskell and verified an amortised complexity analysis by Häupler et al. [HST15] and even improved upon their original approach by supplying a unified analysis for both the insert and deletion method. Weak AVL Trees are relatively unknown and not yet in use in mainstream applications, but their general structural behaviour and familiarity with both AVL trees and Red-Black-Trees give developers an alternative implementation for specific use cases.

1.1 Motivation and Problem Statement

Our motivation for this work was to perform an amortised complexity analysis on binary search trees and find a suitable framework or improve one s.t. the automation degree is, for our purposes, sufficient. A common problem of automatically proving statements about binary search trees is invariants. It is relatively easy to prove some form of explicit change in state in a function by using input and output and calculating it. In contrast,

invariants are implicit and tied to some variables or their role in the structure. Thus, finding a suitable framework which lets us describe an invariant and helps us prove it was the first hurdle to overcome.

So we chose LiquidHaskell plus the RTick library as a framework and the weak AVL Trees (WAVL) tree as a suitable, not yet fully proven data structure. It is not fully proven in that the white paper shows only a pen-and-paper proof, and the structure is relatively recently described in 2015 by Häupler et al. [HST15].

So, our problem was to prove the amortised runtime complexity of the rebalancing process, which was not yet shown to be doable with LiquidHaskell.

Others did similar work, i.e. Nipkow showed some amortised complexity proofs [Nip17], Niki Vazou et al. provided some small examples on how to use the RTick library to prove simple examples in [HVH19], while in [Hoc24] the RTick Library is used for a more complex use case. More on it is described in section 1.2.

1.1.1 Binary Search Trees and Invariants

Binary Search Trees (BST) have a long history being researched but, as a subject, are still not fully figured out, and there is still active research. The main topics of interest are concurrency, better worst-case bounds in certain operations and verification of these bounds. Starting from the most famous ones, i.e. AVL Trees and Red-Black Trees, many minor adjustments to their algorithmic behaviour were investigated. With good reason, since BST are used in databases as indexes and in Linux in multiple areas.

Formalising an automated amortised complexity bound for a BST is difficult because these data structures rely heavily on structural invariants. These are implicit constraints over a data structure which are not explicitly expressed via variables. To prove these structural constraints and use them in the following for proofs of their respective runtime complexity, one needs to show that they hold at all times or use repairing actions to correct the structure.

Only by proving that an algorithm adheres to such constraints can one show that an algorithm exhibits a certain low worst-case bound. For BST, such structural constraints are usually *order* and *balance*.

Balancedness is the main invariant we look at since it is needed for all proofs on worst-case runtime bounds.

1.2 State of the Art and Related Work

Research on binary search trees and especially on WAVL was done by Häupler et al. in [HST15] and [HOSS97], while the latter extends the WAVL definition by using a different deletion method. The amortised complexity analysis using potentials in [Tar85] is the current standard in the complexity analysis and is extended by using different bases of potentials, e.g. binomials or logarithmic ones in [CFG19].

Verification of Data structures in LiquidHaskell becomes more common and is actively researched in [JSV15, Hoc24, Pal20] while only in [Hoc24] we find another example of formalisation of a runtime complexity analysis. To the best of our knowledge, there is no other functional implementation of WAVL trees or a verification of its functional correctness. The same goes for the verification of its complexity bounds for its functions.

On a further note, we have to appreciate the fact that LiquidHaskell, while mature enough to be part of many works of late, is still under active development and general issues which affect the whole field of term rewriting and formal verification are addressed (i.e. see [Vaz23, VG22]). This shows a similar mentality to how the whole Haskell Community is developing rapidly, and changes are quickly incorporated into production code.

1.3 Methodological Approach

In the proceedings of this work, we analysed multiple BST to find suitable candidates for thorough research on their amortised runtime complexity and did a rigorous analysis of their existing pen-and-paper-proofs.

By doing so, we gained extensive knowledge on how a possible implementation could be achieved. We fixated our research on LiquidHaskell as our theorem-proving tool, which suggested some support for the BST invariant problem.

In the next step, we show a functional implementation of the WAVL tree in Haskell; for this, we used existing implementations of AVL and Red-Black trees to guide us.

This prototype is then annotated with the refinement types, and we proved with it balancedness and termination of the WAVL tree.

In the last step, we add the RTick framework to it, using it as a resource analysis framework [HVH19]. The RTick library uses a monadic approach and is wrapped over our initial prototype. This extends our functional correctness proofs, and with it, we can prove the amortised bounds of the *insert* and *delete* methods.

1.4 Contributions

Our contributions in this work are thus three-fold:

- functional implementation of WAVL trees in Haskell
- verification of functional correctness and termination of its methods
- verification of an amortised runtime complexity analysis on WAVL trees

The sections in this work are mapped out to understand the final proofs for the WAVL trees. In chapter 2, we lay out the amortised complexity analysis and the mathematical

fundamentals we employ. In chapter 3, we are taking an overview on LiquidHaskell as the theorem prover of our choice and go into some details which are essential for the chapter 4 where we go first over the details of the WAVL Tree and then go into our proofs. Last, in the final chapter 5, we take a look at some future directions and give our conclusion on this work.

CHAPTER 2

Amortised Complexity Analysis

In computer science, we often want to know the *worst case* runtime an algorithm can have under the most unfavourable conditions. Thus, the worst case refers to the maximal amount of computation steps an algorithm can take or how much memory it will occupy. Such analysis gives developers predictability and guarantees in their application designs.

Amortised complexity analysis is a method used to calculate the time required to perform a sequence of data structure operations, averaged over all the operations performed. This approach provides a more nuanced understanding of performance than a worst-case analysis, which only considers the most time-consuming operation. Such an analysis can also be done on other resources like storage, but we focus on runtime operations in this work.

2.1 Different Analyses Methods

there are three main key concepts of amortised Analyses: Aggregation Analysis, the Accounting Method and the Potential Method.

For the Aggregation Analysis, we take the total time for a sequence of operations and divide it by the number of operations to find the average time per operation. This is useful when we can show that a costly operation decreases the possibility or cost of future operations.

In the Accounting Method approach, we assign a hypothetical "charge" or "token" to different operations. Some operations may have a charge higher than their actual cost, and this extra charge can pay for other operations with a higher actual cost than their assigned charge. The idea is to ensure that the total charge assigned to all operations is enough to cover their total cost.

2.2 The Potential Method

In the Potential Method, which we use in our analysis for WAVL trees, we assign a "potential" Φ to the structure depending on its state. Potential, in this sense, is just another way of saying that we store the cost of work for a later time to be consumed by overpaying on the cost of some functions, thus *increasing* the overall potential stored.

Each function call on the structure can change the stored potential. While certain operations increase the potential, other operations will decrease it. With this technique, we want to show that all changes to a structure's potential sum up to a positive number. The potential of a structure is denoted as Φ in the following. This method works well for us since we deal with different runtime costs for operations, and it allows us to perform fine-grained accounting on different parts of a function or variations of the input.

The mathematical formula is then written like this:

$$T_{amortized} = T_{actual} + c * (\Phi_{after} - \Phi_{before}) \tag{2.1}$$

Further, if we analyze over some actions n executed over a data structure T, we find that we can summarize the actions as in 2.2

$$\begin{aligned} c &= 1 \\ T_{amortized} &= \hat{c} \\ T_{actual} &= \bar{c} \\ \hat{c} &= \bar{c} + c * (\Phi_{after} - \Phi_{before}) \\ \sum_{i=1}^{n} \hat{c} &= \sum_{i=1}^{n} (\bar{c}_i + (\Phi_i - \Phi_{i-1})) \\ &\sum_{i=1}^{n} (\bar{c}_i + (\Phi_i - \Phi_{i-1})) \\ \sum_{i=1}^{n} \bar{c}_i &+ \sum_{i=1}^{n} (\Phi_i - \Phi_{i-1}) \\ &\sum_{i=1}^{n} \bar{c}_i + \Phi_n - \Phi_0 \end{aligned} \tag{2.2}$$

Thus, in our analysis, we need to show

1. a potential of a structure is never decreased below 0 by a given operation, i.e. $\Phi_n - \Phi_0 \geq 0$ (s. 2.4)

2. potential decreases can cover $\bar{c}$ costs

The second point shows that our functions (rules) for calculating potential over a given data structure correspond to the change introduced by an operation on the structure.

Next, we need to account for the potential increasing operations for which we can extend the amortised costs overall operations in 2.3 and find a bound for all operations such that we can prove that the inequality in 2.3 holds. Thus, the *amortised* costs for all operations become a bound for the actual costs, and our analysis is concluded.

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} \bar{c}_i \tag{2.3}$$

$$\begin{aligned} \sum_{i=1}^{n} \hat{c}_i &\geq \sum_{i=1}^{n} \bar{c}_i \\ \sum_{i=1}^{n} \bar{c}_i + \Phi_n - \Phi_0 &\geq \sum_{i=1}^{n} \bar{c}_i \\ \Phi_n - \Phi_0 &\geq 0 \end{aligned} \tag{2.4}$$

2.3 Example: Multi-Delete in Arrays

A practical example of a potential analysis showcasing its application is the analysis of an array's insert, delete and multi-delete operation. The following ruleset applies:

- We define potential as the number of elements in the array.

insert adds an element to the array and *increases* the potential by 1.

delete removes an element from the array and *decreases* the potential by 1.

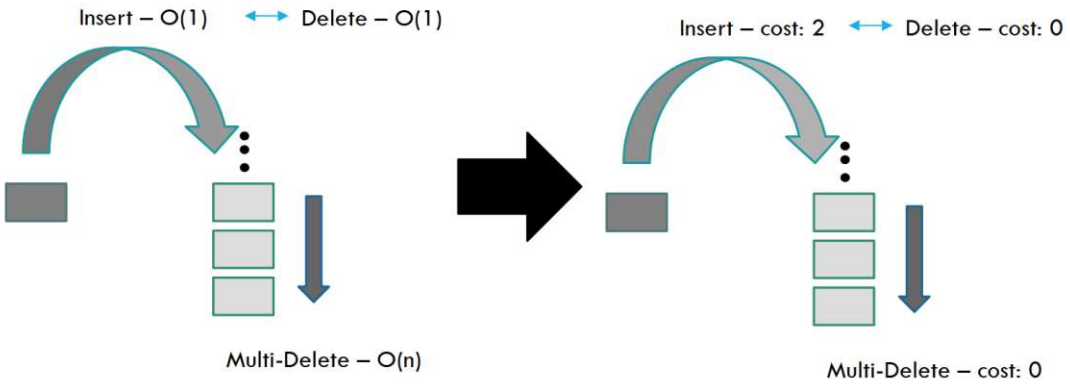
multi-delete removes up to n elements from the array and thus decreases the potential of up to n.

Theorem 1. *All three operations, insert, delete and multi-delete, can operate on a given array in amortised time of* $O(1)$

Without an amortised analysis, we would have to categorise these three operations such that insert and delete would have worst-case bounds of $O(1)$ while the multi-delete would have a worst-case bound of $O(n)$. To prove that over all operations we have a bound in $O(1)$, we apply our amortised analysis and argue that the potential is changed as depicted in listing 2.3.

Then, we find that the situation is as depicted in 2.1.

Figure 2.1: Multi-Delete in Arrays



Proof. We define the actual cost of inserting/deleting an array element to be 1. Potential changes are as in listing 2.3.

$$\begin{aligned} \hat{c}_{\text{insert}} &= 1 + 1 \\ \hat{c}_{\text{delete}} &= 1 - 1 \\ \hat{c}_{\text{multi-delete}} &= n - n \quad \text{for n elements} \\ \forall \hat{c} = \sum_{i}^{n} max(\hat{c}_i) &= 2 * n \end{aligned}$$

A delete operation can only be applied to a non-empty array. Similarly, a multi-delete can only delete as many elements in an array as it contains, which is incidentally the same as the potential. Then we find that for n operations, we get amortised costs of $2 * n$, which for one operation is $2 \ll O(1)$; thus, Theorem 1 holds. □

CHAPTER 3

LiquidHaskell

3.1 Introduction

LiquidHaskell is a static verifier for Haskell, an extension of the Haskell programming language. LiquidHaskell uses a refinement logic to add logical predicates to Haskell's type system [VSJ14].

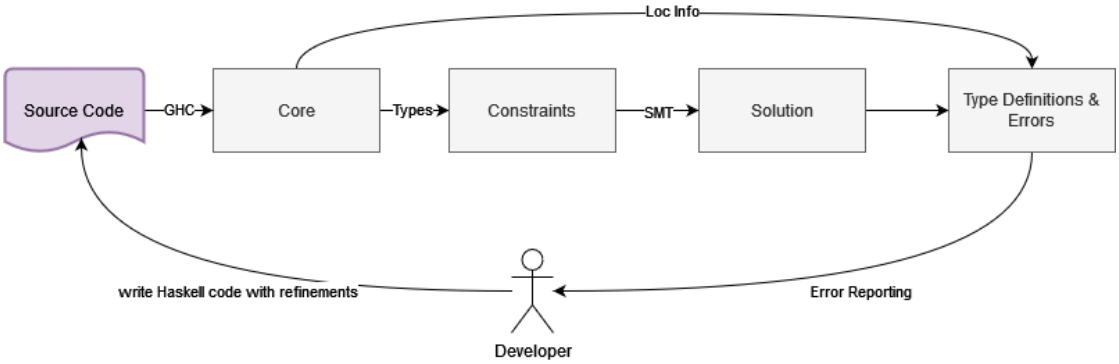
These boolean predicates can express properties like the range of an integer or the shape of a data structure, enabling the verification of properties like array bound checks or division safety at compile time.

Simple enough, these functions and type refinements apply to various industries and code bases. The ability to check logical constraints on functions at compile time enables developers to write safer and more resilient code.

For the verification, LiquidHaskell reduces the refinement annotation to a logical equivalent, which is then given to a Satisfiability Modulo Theories (SMT) solver like Z3 from Microsoft Research or MathSAT [Hø14], thus dipping into the growing power of SAT-solvers. In that context, LiquidHaskell (LH) employs the Quantifier free logic of Equality and Uninterpreted Functions and Linear Arithmethic (QF-EUFLA) as the general theorem to build upon, while the verifier itself is implemented as a GHC plugin to enable verification at compile time.

As a theorem prover LH can be compared to Dafny or F* or on a more general basis in comparison to its powers as a theorem prover, we can look at Agda or Coq. Its most significant advantage over the others is the integration into a programming language, which is actively used by a huge community for something other than only proofs. The ease of writing enables developers to make a smoother entry into correct annotations with relatively few extra steps in work to get started.

Figure 3.1: LiquidHaskell Workflow



3.2 Workflow

The basic setup for working with LiquidHaskell requires installing a SMT solver in the development setting. Our setup is explained in more detail in the appendix at 5.1 for reproducible results.

The workflow in 3.1 consists of multiple stages, which annotations go through until an Error or a Safe-to-use is reported to the developer. Since this happens behind the scenes, developers don't necessarily need to concern themselves with the specifics and can view LHas a black box or an oracle.

In short, the logical predicates are taken, plus the location info is saved for later usage. The code is simplified and enriched via the LHcore. By enriched, we mean annotations like measure, inline and reflect are expanded to constraints to match the functional implementation. In the same way, clauses are pruned where applicable to minimize the overall Satisfiability Modulo Theories Library (SMT-LIB) data sent to the SMT solver. At the reporting stage, LHcombines the location info with the reported errors given by the SMT solver, thus allowing an IDE to mark faulty lines in the source code.

Remark. *We used Visual Studio Code with the simple GHCi Integration extension to use the error markup in our development.*

3.3 Examples

In the following, we will present some syntax examples of LiquidHaskell to get a better handle for the proofs and refinements in chapter 4. Since a case study on the whole functionality of Haskell would be too much for this work, we expect a base knowledge of the programming language Haskell and its syntax. We do not use any special *dialects* of Haskell, and only the LH specifics need to be understood in a bit more depth.

3.3.1 function annotation

While in Haskell, we already write for each function-specific input/output relation using normal Haskell syntax, by using LH, we can take the concept a step further:

Listing 3.1: absolute function

```
1 {-@ absolute :: x:Int -> {v:Int | v >= 0} @-}
2 absolute :: Int -> Int
3 absolute x
4     | x < 0     = -x
5     | otherwise = x
```

The function *absolute* operates on an integer and returns an integer as defined in the first line in 3.1. We know from math what the absolute *should* return, namely, always a non-negative integer. Thus, a *refinement* could be written as in line 1 of listing 3.1 and checks that the function always returns an integer greater or equal to 0.

LH annotations are written as code comments, enclosed by {−@...@−} and extracted during compile time.

3.3.2 Type annotation

LH refinements are not limited to functions. Similarly to them, we can define type refinements, which allow you to constrain values of a given type and reuse such definitions.

The basic syntax for a type refinement in LiquidHaskell is:

```
1 {-@ type Alias = OriginalType {variable | constraint} @-}
```

In 3.3.2, *Alias* is the new type name that can be used instead of the *OriginalType* with the added *constraint*. The *constraint* is a predicate expressed in terms of a *variable* that must hold true for all values of this type.

For example, we define that integers greater than or equal to 0 are of the type *Pos* as defined in 3.2. With this type, we can simplify our example 3.1 and write it more naturally as in 3.3. For brevity, we will leave out the enclosing brackets if not explicitly needed as a distinction between the Haskell code and the refinement.

Listing 3.2: Pos type annotation

```
1 type Pos = {v:Int | v >= 0}
```

Listing 3.3: absolute function, v2

```
1 absolute :: x:Int -> Pos
```

Furthermore, we can extend existing types with other constraints and create additional sub-types, i.e. something like example 3.4 is possible.

Listing 3.4: type with constraints

```
type limitedBy10 = {v:Pos | v <= 10}
```

3.4 LiquidHaskell keywords

In this section, we want to present some of the LHkeywords used in our work because their inner workings influence the extracted constraints. These keywords are also explained in more detail at `https://ucsd-progsys.github.io/liquidhaskell/`, but the docs are prone to change. In this work, we explicitly refer to LHversion 0.9.4.7.

3.4.1 data type

Similar to what we have seen with type refinements, we can annotate a Haskell data class with a LHannotation. In our example listing 3.5, we define a heap tree and refine the Tree by constraining the left and right children in listing 3.6. The refinement states that only values greater than or equal to the value of the root node are allowed as input.

Listing 3.5: Haskell data class example

```
    data Tree a = Leaf
    | Node { value :: a
           , left  :: Tree a
           , right :: Tree a
           }
```

Listing 3.6: Haskell data class refinement

```
    {-@ data Tree a = Leaf
    | Node { value :: a
           , left  :: Tree {v:a | v >= value}
           , right :: Tree {v:a | v >= value}
           }
@-}
```

3.4.2 Termination

In Software verification, an often-asked question concerns the termination of a function. LiquidHaskell also faced that question and introduced a syntax for it. The so-called *termination metric* can be assigned to functions and data types to give the program a function through which the program can evaluate if an execution will reach an end. If you ask yourself if that solves the Halting problem, it doesn't. A heuristic is used, which essentially states that a type, if used in a loop, has a numeric value that must decrease and go towards 0. The general syntax is like in Listing 3.7.

Listing 3.7: Haskell data refinement with termination metric

```
    {-@ data T [termination_metric] a = A | B @-}
```

We use termination for our proofs to show that functions will, at most, take n steps to complete.

3.4.3 measures

In the LHlogic, measures have a special place. A function annotated with the keyword *measure* needs to conform to the following rules:

- it can be recursive but only on one argument
- it may only take one argument

LHchecks a measure for termination, usually via structural induction, on the same argument. To prove this, all parts of the function must be present in the LHlogic. Put another way, LHwill complain if parts of the function are not annotated with *measure* or *reflect* since normal Haskell functions are only proven to be correct and then only their functional refinements are reused in the LHlogic, but the proofs get pruned.

Listing 3.8: measure example

```
1
2 {-@ measure len @-}
3 {-@ length :: s:SafeList -> {v:Int | v >= 0} @-}
4 length :: SafeList -> Int
5 length Nil = 0
6 length a = 1 + length (tail a)
```

The example 3.8 shows how we can define a measure to get the length of a dynamic array from the previous example 3.6.

3.4.4 inline

The *inline* keyword is used to lift a Haskell function into the LHlogic. Such a function is restricted to be non-recursive, and dependent sub-functions must also be in the LHlogic. An advantage of an inline-defined function over the LH predicate keyword is that we get a concrete Haskell-esque function definition for an inline function, which LHuses to cross-check input parameters when used in annotations. An example of an inline is used in our proofs at code **??**.

3.4.5 refinement reflections, PLE and Proofs

In our proofs, we make use of Proof by Logical Evaluation (PLE), which automatically unfolds function definitions and evaluates expressions to assist in proving properties of Haskell programs. This automatic evaluation helps establish whether predicates or properties hold for functions across their possible inputs. It simplifies the process of verifying function behaviour by reasoning about their logical consequences without manual proof steps.

For example, when LiquidHaskell encounters a function with a known output for specific inputs, PLE allows the system to use these concrete outputs in proofs directly. This automatic unfolding of function definitions to instantiate proofs is a powerful feature that leverages the Haskell type system and the underlying SMT -solver.

The general outline of a proof looks like this:

Listing 3.9: Haskell Proof Structure

```
1  {-@ proof_1 :: a -> { bool_statement } @-}
2  proof_1 :: a -> Proof
3  proof_1 v = ()
```

In listing 3.9, we find the same syntax we use as in normal Haskell functions. Also, the function refinement stays the same. Note that the unit type *Proof* is the output, which follows other Haskell implementations of Proofs. The empty clause () represents a *trivially* produced proof and is the short form of the keyword *trivial*.

The *bool_statement* in the refinement is our hypothesis, which we want to prove via proof steps. Proof steps are similar to traditional pen-and-paper proofs derived from previous truth statements or axioms, which we can express as Haskell terms. The chain operator `===` is used to show equality between terms. Further, since many of our proofs operate with the logic QF-EUFLA in mind, we use the chain operators `=>=` and `=<=` for showing inequality directions between terms.

An example is shown in listing 3.10.

Listing 3.10: Proof Structure with Chaining proof steps

```
1  {-@ proof_2 :: a:Nat -> {b:Int | b == div a 2 } -> { a >= b } @-}
2  proof_2 :: Int -> Int -> Proof
3  proof_2 0 _ = ()
4  proof_2 a b = b
5              ==== div a 2
6              =<= a
7              *** QED
```

The proof can also use other functions to deduce the correctness of steps via the ? operator, which effectively drops the second argument but uses it to prove the output Proof type by combining the first argument with the second. This makes for an excellent concise proving scheme and lets us split sub-proofs into other functions. LHwill use the second argument as a guide for PLE to prove the step is sound.

QED finalizes a proof and is derived from mathematics where it stands for the Latin *Quod erat demonstrandum*, i.e. *shown what was to be proven*. It is not necessary but makes for a nice mathematically looking proof.

In some of our proofs, we use multiple chained boolean sub-proofs in parallel with the inequality chain operators `=<=` and `=>=`. This needed a separate function to tie it together, namely the prove function in listing 3.11. A concrete usage for it can be found in the proof 4.6.1 for the logarithmic height.

Listing 3.11: *prove* function

```
{-@
inline prove
prove :: {v:Bool | v} -> {v}
@-}
prove :: Bool -> ()
prove _ = ()
```

3.4.6 RTick

For our amortized complexity analysis, we use the RTick module of LiquidHaskell [HVH19]. Acting as a wrapper over our tree data types, we can count the actual execution costs of statements and compare them to our amortized cost sums. In doing so, and when LHshows that our code is sound, we have proof of the amortized runtime cost.

Listing 3.12: RTick data type

```
data Tick a = {
    tval :: a,
    tcost :: Int
}
```

The *tval* member function of *Tick* returns the wrapped type, which in our case will be a tree type. The *tcost* function returns the accrued costs.

The amortized cost function can be found in appendix **??** and 5.1.

For other monads, there are several commonly used verbs that work on monads, such as *pure* and $liftM$. In our context, we only use the *pure* function, which wraps a type in the Tick monad and assigns a cost of 0.

CHAPTER 4

weak AVL Trees

A WAVL tree is classified as a rank-balanced tree by Häupler et al. in [HST15]. This height-balanced data structure is named after the AVL Tree, which was invented in 1962 [GAV62] and was the first structure in the domain of binary search trees to tackle large search spaces and having worst-case runtime bounds in $O(log\ n)$for all of their operations, in particular search, insert and deletion of nodes due to their self-balancing nature.

Both structures are binary search trees, meaning that a tree has a single *root* node, and each node in the tree has at most two children, respectively called the *left* and *right* child of a node, which is then called the *parent node* respectively. A root node has no parent. Further, a node is a *sibling* to another node if they are both children of a common node.

A node without children is called a *leaf*.

By chaining multiple nodes as children, we get our binary tree. These connections between the nodes build up the *paths* in the tree.

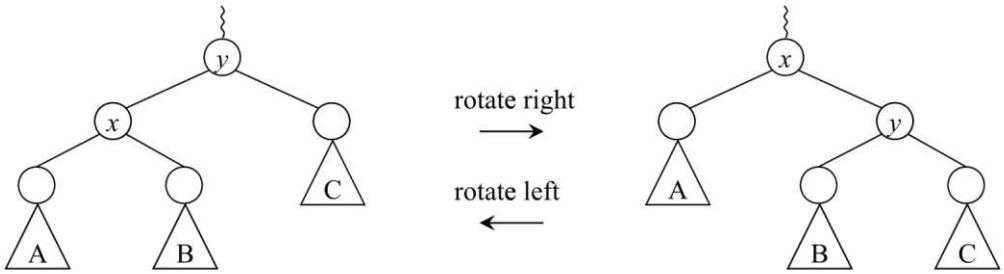
A *path* starts at the top at the root node and ends at a leaf node. The longest path in a tree defines the *height* of a tree.

In this analogy, we define that relative to a node, all nodes between it and the root are *ancestors* to it, including the root itself.

Characteristics of Binary search trees are:

- *Balancedness* in that sense describes structures' characteristic to keep a relatively similar amount of nodes in both its children.
- *Orderedness* nodes in a tree are inserted lexicographically so that the left node is respectively either always (strictly) smaller or greater than its parent node, and the right node is always greater than its parent.

Figure 4.1: Right rotation at node x. the triangles A, B and C denote subtrees. For the inverse operation, the same applies, [HST15]



- it's *height* is logarithmic in relation to the number of nodes in the tree.

All these characteristics are needed for worst-case runtime in $O(log\ n)$, i.e. searching for an entry or node in such a structure starts at the root, comparing an entry against the node and then following either along the left or right-hand side of the tree.

A key operational feature of such structures is the *self-balancing nature* of their functions, i.e. after an insertion or deletion of a node, the structure checks for if it is still balanced. Imagine this would not be the case. Then, a tree could insert a list of nodes already sorted, starting from its smallest element. This would lead to a tree having only ever inserted new nodes on its right-hand side, essentially becoming a list. This would degrade the search runtime to a linear-bound (i.e. $O(n)$).

Rebalancing is accomplished via *rotations*, which means moving nodes so that the Orderedness is preserved and balancedness is acquired again.

The main building blocks of such rotations are the left and right rotations, as seen in figure 4.1.

4.0.1 Ranks and rank differences

A rank is an integer, and every parent node in a tree has a rank that is strictly greater than its children's. A tree's rank is the rank of its root node. We define the rank of a missing node (i.e. NIL node) to be -1. This definition is needed since we compare ranks and classify nodes according to their *rank difference* with their children, even if they are missing. A node with rank differences to both of its children of 1 is called a 1,1-node. A node with a rank difference r to its parent is called a r-child.

A node rank is increased or *promoted* during the rebalancing process. Similarly, a rank can be decreased or *demoted* if the rebalancing requires it.

Häupler et al. made a framework around the idea of overcoming the weak points of AVL, namely the rebalancing after a node deletion, and created the classification of

rank-balanced trees to describe trees which rely on information in nodes which are not directly height-related like AVL trees or weight-related (i.e. sum of nodes).

4.1 Definition of different rank-balanced trees

- **AVL trees** named after its inventors, Adelson-Velsky and Landis, are trees where the difference in heights between the left and right subtrees (balance factor) of any node is at most 1 and the difference to the parent is at most 2.

  In terms of rank difference, we allow 1,1, 1,2, and 2,1-nodes in AVL trees but not 2,2-nodes. **Remark:** The rank at the root node is the same as the tree's height.

- **Red-Black trees** are so-called because nodes in the tree are coloured either red or black. With this, the following rules apply:

  – The root node is coloured black
  – all NIL nodes are coloured black
  – red nodes cannot have red children
  – Every path of root to a leaf must have the same number of black nodes (*Black Property*)

  In Red-Black trees, we count NIL nodes as part of the tree to fulfil the *Black Property*. By disallowing red children of red nodes plus the Black Property, we get a tree with a height of at most $2log(\ n)$ with n being the number of nodes in the tree.

  Regarding ranks, we allow 1,1-, 2,2-, 1,2- and 2,1-nodes.

- **WAVL trees**

  – rank differences are either 1 or 2
  – leaves are of rank 0

  This rule is a bit special, but during a deletion, a 2,1-node can become a 2,2-node of rank 1. Such nodes are prohibited from keeping a bound on the number of rotations needed per insertion or deletion. This rule was added since the deletion of such a node can result in a 2,4-node, which can not be solved by two rotations but three. More on that is explained in the proof section of this chapter.

This classification was meant to directly link AVL trees and Red-Black-Trees, which both fit this description, even though they have different *original* recipes of their own when to rebalance. By doing so, Häupler et al. found a related structure, which was somewhere in between and this structure was then called the weak AVL tree.

On this remark, in an initial working paper, the authors still called it rank balanced tree, but this was such a commonly used name that it was dropped, and the name weak AVL tree was chosen instead [HST09].

4.2 insert and delete

An insertion in a WAVL tree follows the typical process as in an AVL tree:

- start at the root
- find a node with a missing child where the new node can be inserted into
- insert the new node
- starting from the new node, follow along the path back to the root
- at each ancestor, check if the balancing condition is violated
- if it is, rebalance the tree

Remark. *In an imperative programming language, we could exit the function because to manipulate the tree, we would use pointers and rebalancing a tree is a rotation of pointers which are exchanged between nodes to get a balanced configuration. In our functional approach, we cannot exit the function but have to return to the root node since we don't deal with pointers but pure data structures, which are only defined by the functions working on them.*

At the same time, the purely functional approach gives us confidence in our proofs that the *code context* is not changed by any side conditions. In an imperative language, global variables are often assumed or at least have to be accounted for during execution. In a functional context, this can be ignored.

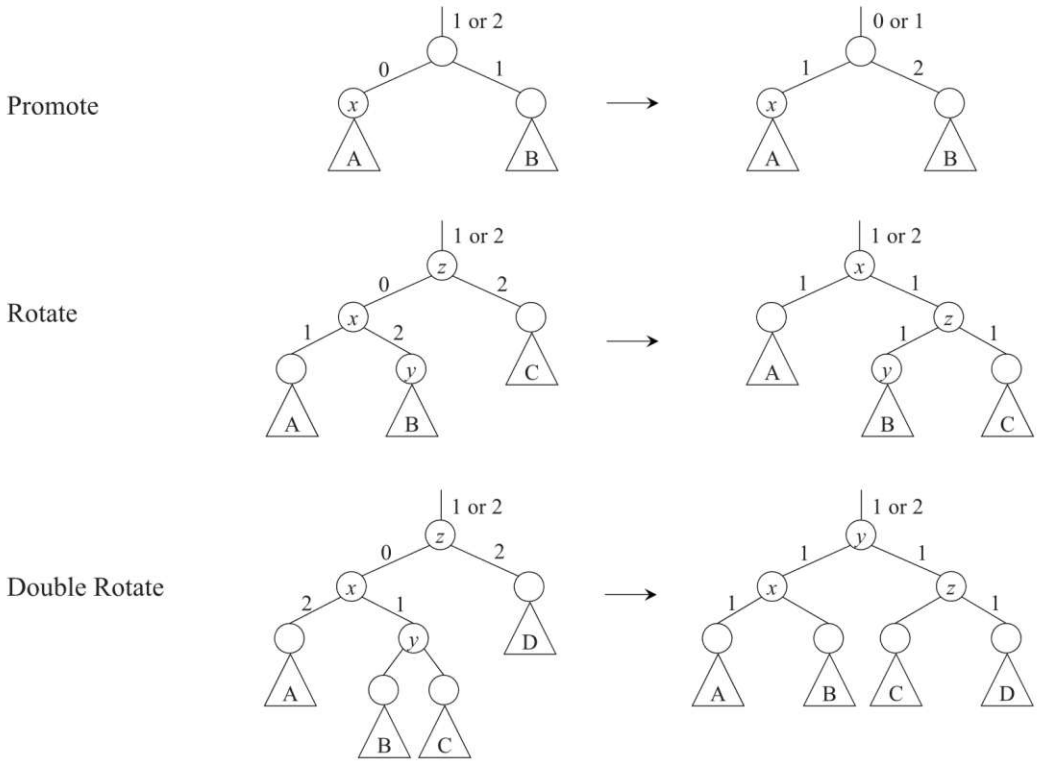
4.2.1 rebalancing in insert

The rebalancing is conducted as follows:

- after the insertion of node x in y, we check for the rank difference to x.
- if the rank difference is 0, then one of three cases can be. For now, we assume that x is the left child of y:
  1. *promote*: node y is a 0,1-node; then y gets promoted, i.e. its rank is increased by 1. The rebalancing will continue; y gets to set x, and the parent of y is set to y.
  2. *rotate*: node y is a 0,2-node, and x is a 1,2-node; then y is rotated once to the left, and y is demoted by 1. The rebalancing stops;
  3. *double rotate*: node y is a 0,2-node and x is a 2,1-node; then we rotate first x to the right and then y to the left. x and y are both demoted by 1.

We have mirror cases for the rotation cases. Rebalancing is continued only after a promote case. This is possible because a rotation case fixes the imbalance. The functional correctness proof for that is done in LiquidHaskell for all cases.

Figure 4.2: Insert - Rebalancing steps taken after an insertion, Numbers next to edges are rank differences. All cases have mirror images, [HST15]



4.2.2 rebalancing in delete

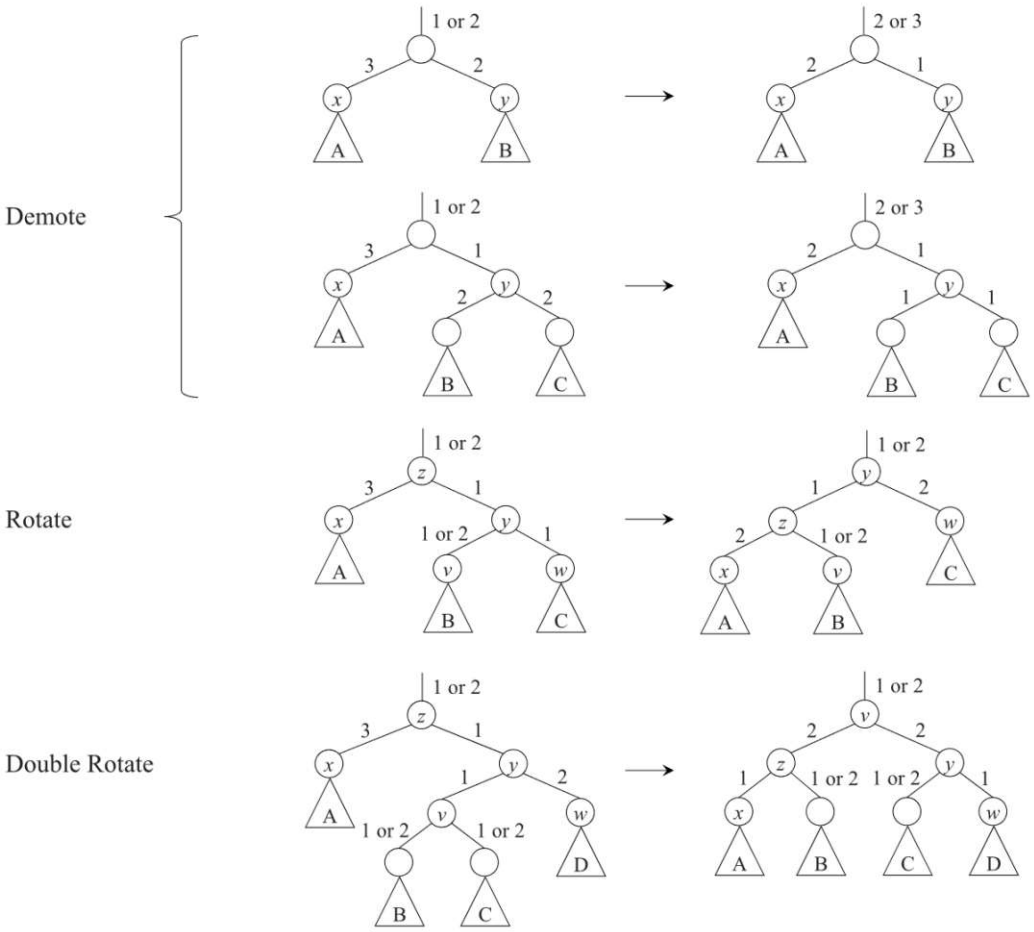
Similar to the insert case, we conduct the rebalancing in the delete case. The deletion takes place in the following:

- find the node to delete
- if the node is not a leaf, find the immediate successor of the node which is a leaf (s. the *getMin* function down below)
- starting from the position of the successor (or the node itself, if it is a node), we remove the node and start the rebalancing process beginning with its parent y while the deleted node is x

Then the rebalancing can have the following cases, assuming x is the left child and its sibling node is z:

- *demote*: y is a 3,2-node: demote y by 1 and set the parent of y to y and the y to x. The rebalancing continues.

Figure 4.3: rebalancing after a deletion. The same syntax as in Fig. 4.2 applies. the demote steps may repeat, [HST15]



- *double demote*: y is a 3,1-node, with z being a 2,2-node itself: demote both y and z by 1, then continue analogue to the demote case.

- *rotate*: y is a 3,1-node with z having a rank difference of 1 to its right child: rotate right at z, promote z and demote y. if z is a leaf, demote it again. This restores the rank rule, i.e., leaves must be of rank 0.

- *double rotate*: y is a 3,1-node, with z being a 1,2-node (you only need to check for the right child being a 2-child): rotate left on z, then rotate right on y. promote the previous left child of z twice, demote z by 1 and y by 2.

Both rotation cases end the rebalancing process since the tree is balanced again.

Implementation, Version 1

In our attempts to prove the WAVL trees invariants, we started with the classical definition of a tree:

Listing 4.1: WAVL tree definition

```
data Tree a = Nil | Tree { val :: a,
                           rd :: Int,
                           left :: (Tree a),
                           right :: (Tree a)
                           }
    deriving Show
```

This definition is now refined with some constraints in the LiquidHaskell style:

Listing 4.2: WAVL data type refinement

```
    {-@ data Tree [rk] a = Nil | Tree { val :: a,
                                        rd :: {v:Int | v >= 0 },
                                        left :: ChildT a rd,
                                        right :: ChildT a rd } @-}
```

In this annotation we stated, that the tree has a Termination metric *rk* and a rank rd, which is essentially the same. But because we want to compare ranks of empty trees (NIL) and non-empty trees, we define a separate function *rk* in which we also define the rank of NIL trees:

Listing 4.3: rank function

```
{-@ measure rk @-}
{-@ rk :: t:Tree a -> {v:Rank | (empty t || v >= 0)
                              && (notEmptyTree t || v == (-1))}
@-}
rk :: Tree a -> Int
rk Nil = -1
rk t@(Tree _ n _ _) = n
```

The *rk* function is defined as a measure to reuse it in other LH refinements. The annotation for this function also describes that every tree is either empty or not empty and combines that information via logical implication with the numerical information of the rank. This seems intuitive initially, but LH needs this constraint to infer the connection between the rank and empty and non-empty trees in the other refinements.

To complete the definition of the WAVL tree, we still need the recursive definition:

Listing 4.4: additional WAVL type refinements

```
type ChildT a K = {v:Tree a | rk v <= K && K <= rk v + 3}
type Wavl = {v: Tree a | balanced v }
type NEWavl = {v:Wavl | not (empty v)}
```

With the balancing function implemented as a measure:

Listing 4.5: balanced function

```
{-@ measure balanced @-}
balanced :: Tree a -> Bool
balanced Nil = True
balanced t@(Tree _ n l r) =
       rk r < n && n <= rk r + 2
    && rk l < n && n <= rk l + 2
    && ((notEmptyTree l) || (notEmptyTree r) || (n == 0))
    && (balanced l)
    && (balanced r)
```

In 4.4, we define the relationship between the child node and its parent via the rank. This is done via a structural recursion, compare 4.2.

In this first version of our tree, we defined it in a *relaxed* variant, i.e. we allow the rank difference to be between 0 and 3. Why did we choose this range instead of the $1-2$? This approach becomes more natural if we look at how trees are usually changed in a functional setting:

Listing 4.6: insert function, v1

```
{-@ insert :: (Ord a) => a -> s:Wavl
    -> {t:NEWavl | ((RkDiff t s 1) || (RkDiff t s 0)) } @-}
insert :: (Ord a) => a -> Tree a -> Tree a
insert x Nil = leaf x
insert x t@(Tree v n l r) = case compare x v of
    LT -> insL
    GT -> insR -- symmetric to insL
    EQ -> t
    where
        l' = insert x l
        r' = insert x r
        lt' = Tree v n l' r
        rt' = tree v n l r'
        insL | rk l' < n = lt'
             | rk l' == n && rk l' == rk r + 1 = promoteL lt'
             | rk l' == n && rk l' == rk r + 2
               && rk (left l') + 1  == rk l'
               && rk (right l') + 2 == rk l' = rotateRight lt'
             | rk l' == n && rk l' == rk r + 2
               && rk (right l') + 1 == rk l'
               && rk (left l') + 2 == rk l'
                    = rotateDoubleRight lt'
             | otherwise = t
```

The LH annotation can be read like it was taken from the paper, i.e.

- the input is formed of an *ordered* Element a and a WAVL tree

- and the output is then an un-empty WAVL tree (i.e. $NEWavl$, or not empty WAVL tree) which has at most a rank of 1 greater than the input tree.

The rest is relatively straightforward, i.e. we define the leaf node as in 5.1. In this version of the insert function, we use the output of the *insert* function to define a new Tree object lt'. This approach follows the exact definition of the WAVL paper [HST15]. First, do the insertion in the child node, then check the *resulting* tree is imbalanced. With that in mind, our initial choice of the rank range makes sense since this makes our code more readable in the following case matching of $insL$.

the case distinctions in $insL$ follows the definition in

The rotation cases are written like this:

Listing 4.7: rotation case for Rebalancing

```
{-@ rotateRight :: {v:Node0_2 | isNode1_2 (left v) }
    -> {t:NEWavl | rkDiff t v 0 } @-}
rotateRight :: Tree a -> Tree a
rotateRight (Tree x n (Tree y m a b) c) =
            Tree y m a (Tree x (n-1) b c)
```

Note that the annotation of *rotateRight* matches the output of the insert's refinement. This is necessary since all LH uses in proving a function are the logical refinements of sub-functions but not their actual definitions. This application of Hoare Logic makes the logic decidable since recursive calls like the one in the insert function are reduced to Hoare Logic and made solvable. At the same time, we lose some information about the object's transformation in the sub-call, e.g., the relationship between the nodes or which node was moved to which exact position. LH only *bubbles up* the refinement, and in our case, we are left with `{t: NEWavl | rkDiff t v 0}` with v being the input tree.

Remark. *Our work shows that this initial approach is indeed correct and terminates, as proven by the Termination metric set in the data annotation. We didn't have to change our way of writing Haskell code too much to prove it with LH. Some features were missing from the logic we expected to be there, but overall, it was an OK experience. One of our main takeaways is that the actual implementation of LH can be classified as academic for the last few years and only really stabilised in the last 2 to 3 years. We had problems with the online LH editor, which used an older version (i.e. 8.6.10) and was incompatible with the latest versions we used for our work. Further, specific pragmas for LH were added since 2021, which changed certain recursive refinements fundamentally, e.g. −−bscope, see here also the This also affects a lot of older code examples found since they are often not working anymore with the more recent versions.*

4.2.3 Problems with Version 1

we started encountering smaller problems when trying to implement the delete function. We wanted to implement it like this:

Listing 4.8: delete function v1

```
{-@ delete :: (Ord a) => a -> s:Wavl
        -> {t:Wavl | (rkDiff t v 0) || (rkDiff s t 1)} @-}
delete _ (Nil _) = nil
delete y (Tree x n l r)
  | y < x     = balLDel x n l' r
  | x < y     = balRDel x n l r'
  | otherwise = merge y l r n
    where
      l' = delete x l
      r' = delete x r
```

But LiquidHaskell was returning an Error about the Pattern Matching in the l and r cases, which were not exhaustive. Thus, we ended up with the following:

Listing 4.9: actual delete function, v1

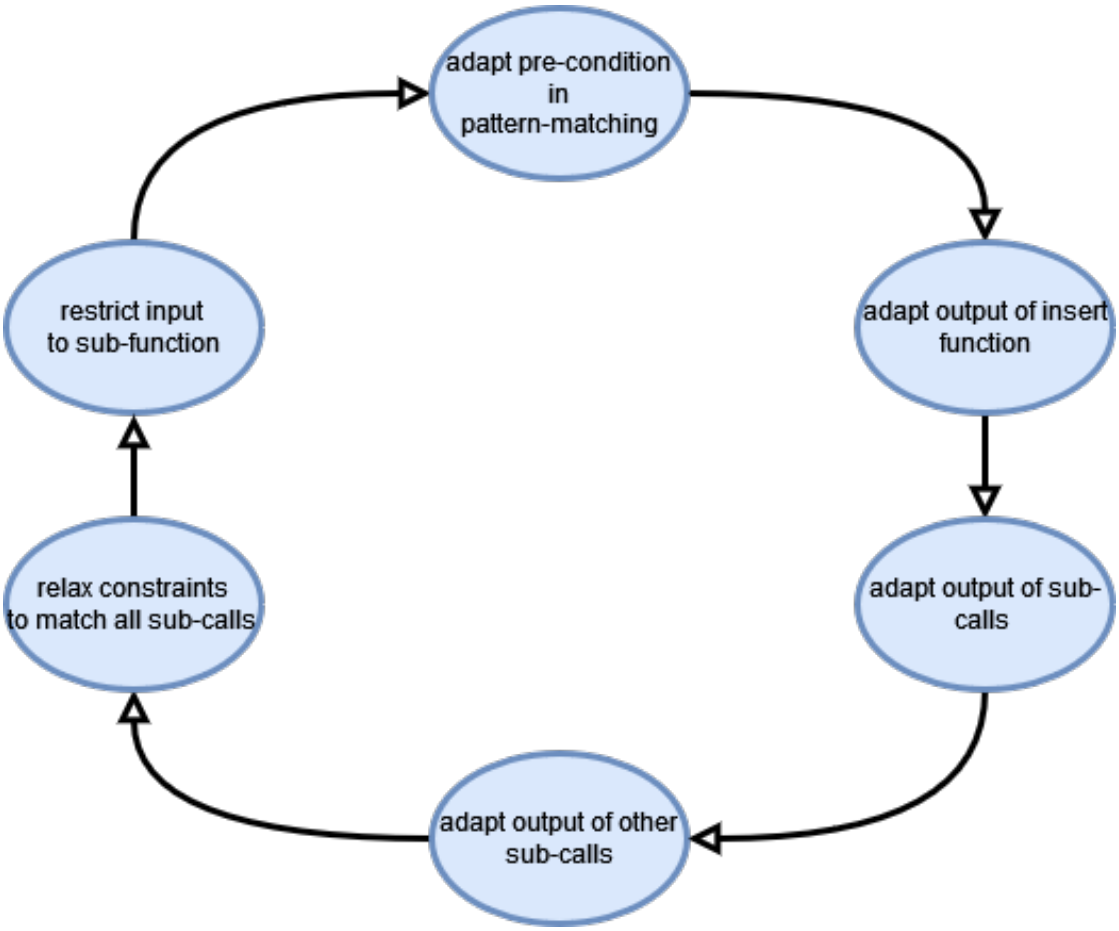
```
delete _ (Nil _) = Nil
delete y (Tree x n l@(Nil _) r@(Nil _))
  | y < x     = balLDel x n l' r
  | x < y     = balRDel x n l r'
  | otherwise = merge y l r n
    where
      l' = delete x l
      r' = delete x r
delete y (Tree x n l@(Nil _) r@(Tree _ _ _ _))
  | y < x     = balLDel x n l' r
  | x < y     = balRDel x n l r'
  | otherwise = merge y l r n
    where
      l' = delete x l
      r' = delete x r
delete y (Tree x n l@(Tree _ _ _ _) r@(Nil _))
...
delete y (Tree x n l@(Tree _ _ _ _) r@(Tree _ _ _ _))
...
```

We couldn't pinpoint the exact problem, but it became clear from the problem in listing 4.9 that the *balanced* constraint did not get expanded constraint-wise for the children. In private communication with the LHmaintainers, we also couldn't solve this problem then (communication via LH Slack channel with Niki Vazou and Ranjit Jhala in early 2023). This first approach did not work out for us because we were unable to prove the amortised complexity with it. In our approaches, the biggest hurdle was finding a suitable refinement for the recursive call in *insert*. Finding the balance between a relaxed enough but still strict enough definition is hard.

4.2.4 amortised cost analysis, version 1

Similar problems arose with the amortised cost analysis. We couldn't get a hold of a simple annotation for the recursive *insert* function. We amassed constraints on top of

Figure 4.4: Workflow of finding suitable constraints for the recursive function *insert*



constraints to restrict the input of functions to *satisfy* LH, but that introduced new problems with the calling functions since that function output was the input to the sub-functions and vice versa, see figure 4.4. A different approach was needed.

4.3 WAVL Version 2

In our second attempt to fix the problem with the recursive definitions, we started from the top and redefined our data type:

Listing 4.10: WAVL data type revisited

```
    {-@ data Tree [rk] a = Nil | Tree { val :: a,
                                       rd :: {v:Int | v >= 0},
                                       left :: ChildT a rd,
                                       right :: ChildT a rd } @-}
data Tree a = Nil | Tree { val :: a,
                           rd :: Int,
                           left :: (Tree a),
                           right :: (Tree a)} deriving Show

{-@ type ChildT a K = {v:Tree a | rk v < K
                                && K <= rk v + 2 } @-}
```

Restricting the tree to only rank differences of up to 2 ingrains the rank difference invariant even more in the data structure.

Now, the only thing left is to restrict the structure to the only allowed configurations:

Listing 4.11: additional WAVL types, v2

```
{-@ type Wavl = {v:Tree a | structLemma v } @-}
{-@ type NEWavl = {v:Wavl | not (empty v) } @-}
```

With the structure Lemma being:

Listing 4.12: Structure instead of balancedness

```
{-@ measure structLemma @-}
structLemma :: Tree a -> Bool
structLemma Nil = True
structLemma t@(Tree _ n l r) = isWavlNode t
                            && structLemma l
                            && structLemma r
```

and

Listing 4.13: node structure constraints

```
{-@ inline isWavlNode @-}
isWavlNode :: Tree a -> Bool
isWavlNode t = isNode1_1 t || isNode1_2 t
            || isNode2_1 t || isNode2_2 t

{-@ inline isNode1_1 @-}
isNode1_1 :: Tree a -> Bool
isNode1_1 t = rk (left t) + 1 == rk t
              && rk t == rk (right t) + 1
```

```
10
11 {-@ inline isNode2_2 @-}
12 isNode2_2 :: Tree a -> Bool
13 isNode2_2 t = rk (left t) + 2 == rk t
14             && rk t == rk (right t) + 2
15             && not (empty (right t))
16             && not (empty (left t))
```

The *isNode∗* other than the 2,2 case has the same structure as *isNode*1_1. The 2,2-case needs special attention because we need to forbid nodes of rank 1 from having this form by disallowing that their children need ranks of 0 or greater. While we also had the definition *isWavlNode* in the first version, we did not use it in the definition of the WAVL type, i.e. 4.11.

Then, our *insert* function can be written like this:

Listing 4.14: insert function v2

```
1  insert :: (Ord a) => a -> Tree a -> Tree a
2  insert x Nil = leaf x
3  insert x t@(Tree v n l r) = case compare x v of
4      LT -> insL
5      GT -> insR
6      EQ -> t
7      where
8          l' = insert x l
9          r' = insert x r
10         insL
11             | rk l' < rk t = Tree x n l' r
12             | isNode1_1 t = promoteL t l'
13             | isNode1_2 t && isNode1_2 l' = rotateRight t l'
14             | isNode1_2 t && isNode2_1 l' =
15                                       rotateDoubleRight t l'
```

Again, we omitted that *insR* is symmetric to *insL*. The important part to note is the type signature of the promotion and rotation cases.

Listing 4.15: rotation case v2

```
1 {-@ rotateRight :: {t:NEWavl | isNode1_2 t}
2     -> {l:NEWavl | isNode1_2 l && rk t == rk l}
3     -> {v:NEWavl | rkDiff t v 0 } @-}
4 rotateRight :: Tree a -> Tree a -> Tree a
5 rotateRight t@(Tree x n _ c) l@(Tree y m a b) =
6                     Tree y m a (Tree x (n-1) b c)
```

Note the different approaches to building the balanced tree. Instead of creating an imbalanced tree as an intermediate product, we give the rotate case a balanced tree to substitute it with its former state; in the case of 4.15, we exchange $leftt$ with l and doing the necessary updates to the ranks and rotation in the same function call.

Comparing this approach to the first version in 4.7, we can see that our input and output are of the same data type, i.e. a balanced tree. This solves the problem with the recursive constraint relaxation, as shown in Fig. 4.4 because we have input and output of the same data type.

A further improvement using this substitution of balanced trees can be found with the updated *delete* function:

Listing 4.16: delete function v2

```
{-@ delete :: (Ord a) => a -> s:Wavl
        -> {t:Wavl | ((rkDiff s t 0) || (rkDiff s t 1))} @-}
delete:: (Ord a) => a -> Tree a -> Tree a
delete _ Nil = Nil
delete y t@(Tree x n l r) = case compare x y of
    LT -> delL t l'
    GT -> delR t r'
    EQ -> merge
    where
        l' = delete x l
        r' = delete x r
        merge
            | empty r = l
            | otherwise = let (r'', x) = getMin r
                    in delR (Tree x n l r) r''
```

One thing to note is the *otherwise* case in merge using a *let* definition. The design choice for let can be explained with a relaxation of constraints. By writing the *merge* definition directly inside the *delete* function, only the refinement of $getMin$ is outside the function's scope. To check correctness, LHwill first check $getMin$ and then use the refinement of it to check the calling function for correctness. The actual code of $getMin$ is not inside the LHlogic in this scenario.

Compare the usage of *merge* with our first versions at 5.1 and 5.1.

The *delL* represents our rebalancing steps and needs a separate function apart outside the delete function because we want to restrict the input to only non-empty trees:

Listing 4.17: delete rebalancing function

```
{-@ delR :: t:NEWavl
    -> {r:Wavl | (rkDiff (right t) r 0
                || rkDiff (right t) r 1)}
    -> {v:NEWavl | (rkDiff t v 0) || (rkDiff t v 1)} @-}
delR t@(Tree x n Nil _) r = treeR t r
delR t@(Tree x n l@(Tree _ _ ll lr) _) r
    | rk t <= rk r + 2 = treeR t r
    | child3 t r && child2 t l = demoteR t r
    | child3 t r && child2 l lr && child2 l ll =
                                    doubleDemoteR t r
    | child3 t r && child1 l ll = rotateRightD t r
    | child3 t r && child1 l lr = rotateDoubleRightD t r
```

Here, we describe the input tree for the substitution analogue and what we expect from the output definition of *delete* so that LH can prove delete as a whole. A special case here that needs the case expansion is the left child of t: Because we need differentiation in the following rotation cases and LH cannot infer if l is empty from the rk differences, we need a separate case where we explicitly write the type pattern as $delRt@(Tree x n Nil_) r$.

The function *treeR* is special because it is basically a wrapper for the tree function but substitutes the r child for the second parameter.

Listing 4.18: default case for insert with no rebalancing required

```
{-@ treeR :: t:NEWavl
    -> {r:Wavl | (rkDiff (right t) r 0 || rkDiff (right t) r 1)
         && rk t <= rk r + 2}
    -> {v:NEWavl | rkDiff t v 0 || rkDiff t v 1} @-}
treeR :: Tree a -> Tree a -> Tree a
treeR (Tree x 1 Nil (Tree _ 0 Nil Nil)) Nil = leaf x
treeR (Tree x n l _) r = Tree x n l r
```

As noted previously, because *insert* and *delete* are recursive, we need to concisely match the input and output refinements. On the other hand, after we find a fitting definition, this hassle proves to be of great value since, without a concrete mention of the recursive nature of the function, we prove its functional correctness.

4.3.1 refinement of delete and insert

Now to the actual LH refinement for the two functions:

Listing 4.19: insert annotation, v2

```
insert :: (Ord a) => a -> s:Wavl
    -> {t:NEWavl | ((rkDiff t s 1) || (rkDiff t s 0))
         && ((rkDiff t s 1 && rk s >= 0)
                => (isNode1_2 t || isNode2_1 t))}
```

The *insert* annotation in 4.19 got beside the already known *NEWavl* and *rkDiff* parts, which we already know from 4.6, another clause which describes the situation that if an input tree is not empty, then we expect the output tree if a rank difference to the original tree exists, i.e. a promote step was executed, then the resulting node/tree must be a 1,2-node. This information is necessary because we check further down for 1,2-nodes and whether a rotation shall happen. This case is only possible on *NEWavl* trees, so the rotation cases are constrained on their resp. inputs.

This information is implicit knowledge of the developer, and we, as the designer, can see that, but LH needs to be made aware of. Otherwise, a "Pattern matching non exhaustive" exception will be thrown.

Compared to the insert refinement, the one for *delete* becomes more straightforward, which only needs the information about rank differences in the output tree t, see listing 4.16.

Listing 4.20: delete annotation, v2

```
{-@ delete :: (Ord a) => a -> s:Wavl
    -> {t:Wavl | ((rkDiff s t 0) || (rkDiff s t 1))} @-}
```

4.3.2 Summarised Improvements for Version 2

In our effort to find a suitable definition for the complexity analysis, we adapted the data type constraints and added the structural Lemma. This results in shorter code but also restricts us. Since we disallowed imbalanced trees, we have to use the tree substitution strategy. We find ourselves adjusting our code for the analysis to the needs of LH. On the other hand, functional correctness is proven by relatively simple one-liners of annotation over concrete functions. LH automates the whole reasoning about the structural invariant. In our opinion, this seems like a reasonable trade-off. Further, by using the notion of isNode directly in the *structLemma* definition, we can reuse it in the pattern matching in delete and insert. This makes for a more explicit connection to the original definition.

4.4 Amortised Cost Analysis

The original amortised analysis by Häupler et al. considers two separate cases: delete and insert, which both receive their own analysis.

In the original paper, the authors state the following theorems:

- Theorem 4.1. "In a WAVL tree with bottom-up rebalancing, there are at most d demote steps over all deletions, where d is the number of deletions"
- Theorem 4.2. "In a WAVL tree with bottom-up rebalancing, there are at most $3m + 2d \geq 5m$ promote steps over all insertions, where m and d are the number of insertions and deletions, respectively."

The proofs for these two can be looked up in [HST15], site 10ff. Häupler et al. call their amortised cost analysis *potential analysis*. Still, it can also be seen as an *Accounting Method*, since the potential can be compared one to one with *charge* accounted for.

We found that the second Theorem could be improved upon, and a theorem that includes both insert and delete calls for both functions could be formulated.

Our improved Theorem is now thus:

Theorem 2. *In a WAVL tree with bottom-up rebalancing, there are at most 3 demote steps for a deletion and at most 3 promote steps for an insertion over all insertions and deletions combined.*

The proof follows from the safety check by LH. In the following, we explain the code and show that our implementation is sound.

For the proof, we use the RTick library to wrap our tree data type in the RTick monad to count actual cost. For costs, we counted 1 for each promote step in insert and the same for each demote / double demote step.

Listing 4.21: insert annotation with Tick monad

```
{-@ insert :: (Ord a) => a -> s:Wavl ->
    {t':Tick ({t:NEWavl | ((rkDiff t s 1) || (rkDiff t s 0))
        && ((rkDiff t s 1 && rk s >= 0)
            => (isNode1_2 t || isNode2_1 t)) } )
        | amortizedStmt s t' && (empty s => amortized s t' ) }
@-}
insert :: (Ord a) => a -> Tree a -> Tick (Tree a)
```

And with the actual implementation:

Listing 4.22: insert function with Tick monad

```
insert x Nil = pure (leaf x)
insert x t@(Tree v n l r) = case compare x v of
    LT -> insL
    GT -> insR
    EQ -> pure t
    where
        l' = insert x l
        r' = insert x r
        l'' = tval l'
        r'' = tval r'
        insL
            | rk l'' < rk t = inTreeL t l'
            | isNode1_1 t = promoteL t l'
            | isNode1_2 t && isNode1_2 l'' = rotateRight t l'
            | isNode1_2 t && isNode2_1 l'' =
                                    rotateDoubleRight t l'
```

Let's break that apart:

- the output of the function is now of type Tick, which wraps a WAVL tree
- We state analogue to 4.14, that the function's output is a non-empty WAVL Tree with either rank difference (rkDiff) of up to 1. We write it like this because the solver would otherwise expand it similarly, but at the same time, he could fail at doing so. We want precisely these two cases for which we define some further implications, so this is a deliberate choice in our definition.
- (rkDiff t s 1 $\wedge$ rk s $\geq$ 0) $\implies$ (isNode1_2 t $\vee$ isNode2_1 t): a bit complicated way of saying *if the insertion happens to a non-empty tree and a promote case was executed, then the output node (tree) is a 1,2-node.* This is necessary for the rotation cases since we want to state that the rotation case is the last step in the rebalancing process, and in a pointer-based setting, we could exit the function call.

Why didn't we need that constraint in listing 4.14? Because we don't need to argue about the amortised cost, we only show that each call produces a legit WAVL tree and has a specific rank difference.

So, to say that this is the last step, we state that a rank difference happens in the previous step, but it still needs to pass the check *rk l" < rk t*. This is the equivalent of saying if the tree is balanced again, you have to return to the root node (or exit the recursion at that point)

- *amortizedStmt*: implementation at 4.23 states our constraint of only ever needing costs per call of only up to 3.
- *pure*: wraps an object in the appropriate monad, in our case RTick with zero costs.
- *tval*: extract the actual object from a monad
- *inTreeL*: create the tree but constrain the input/output tree via the annotation s.t. it fits the insertion. s. the appendix for details 5.1.
- *insR* is symmetric to *insL* and left out for shortness.
- The *actual* cost is added in the function *promoteL* by updating the Tick monad. The potential change is updated by comparing input/output trees in all functions via the *amortised* call inlined in the refinement.

With that, we can move on to the actual proof:

Proof. We use the amortised potential analysis technique as described in 2 for the proof. We define the problem such that we are only interested in the needed rebalancing steps during an insert or delete action on a given WAVL tree. We also assume that the actual insertion of a node into a tree has an actual cost in $O(1)$ and is not regarded in this proof since we only look at the cost of the rebalancing steps. The rebalancing itself has two parts: the promotion/demotion and the rotation step. A rotation step only happens once during an insert/delete action, while the promotion/demotion steps are only bounded at this point by the height of a tree, which is bounded by the number of nodes in the tree. Theorem 3.

For our analysis, we define that a tree's potential is only increased by a rotation step and decreased by the promotion and demotion steps, respectively. We further say that the potential of 2,2-nodes (and of 2,3-nodes) is equal to 2, and the potential of 1,1-nodes (and of 1,0-nodes) with a rank greater than 0 is equal to 1 (i.e. non-leaf nodes). The potential of all other nodes is 0.

For insertion, we look at the three possible cases which are part of a rebalancing process:

- **promote**: a promote changes a node from 0,1 to 1,2 and thus decreases the potential of a node by 1. This *consumption* of potential is balanced out by adding actual costs

of 1. This consumption of potential only works **IF** the node has potential to begin with. An insertion at rank 0 makes a previous leaf into a 0,1-node, but we argue that we are missing the previous step, which explicitly increases the structure's potential. So, to account for that, the first promote step does need to be covered and incurs an *potential increase* of 1 in our calculation, s. the implication in the annotation at 4.24.

- **rotate**: two nodes are affected, i.e. a 0,2- and a 1,2-node become 1,1-nodes. Thus, the potential is increased by 2. No actual costs are counted, and we have a total increase of at most 2. *Remark*: the potential increase can be only 1 if the lower 1,1-node is a leaf.
- **double rotate**: We start on the input side with at most one 1,1-node and end up with at most three 1,1-nodes. At most, there is a potential increase of at most 2.

For all three cases, the symmetric ones apply as well. By adding up the potential changes of the first promote step with the worst case in the rotation, we end with at most 3 costs per insert. This statement is also in the refinement of 4.23, and with LH proving the code safe, the proof follows from the soundness of LH. □

Listing 4.23: Amortised Lemma as a constraint

```
{-@ inline amortizedStmt @-}
{-@ amortizedStmt :: Wavl -> Tick (Wavl) -> Bool @-}
amortizedStmt :: Tree a -> Tick (Tree a) -> Bool
amortizedStmt t v = (rkDiff (tval v) t 0 => amortized3 t v)
                 && (rkDiff (tval v) t 1 => amortized1 t v)
```

Remark. *In 4.23 we state that as long as there is a rank difference there can be at most 1 cost to account for all function calls in the rebalancing process. On the other hand, if both trees are equal in rank, we need to pay up to 3 to cover the actual cost plus the potential change. The amortised3 statement is the coded version of the amortised definition 2.1 with adding 3 costs to the one side of the equation. The actual code is at 5.1.*

The constraint relies on the fact that we only allow rank difference (rkDiff) of either 0 or 1. We use this case distinction on many separate occasions to differentiate between the rebalancing process still going on and being concluded. Again, this is necessary for our functional setting, in an imperative language we could write a proof like in Dafny differently, probably.

Listing 4.24: promote case with Ticks and amortised Lemma

```
{-@ promoteL :: {t:NEWavl | isNode1_1 t}
    -> {l:Tick (NEWavl) | rk (tval l) == rk t &&
        (rk (left t) >= 0 => amortized1 (left t) l ) &&
        (rk (left t) ==(-1) => amortized  (left t) l ) }
```

```
5       -> {v:Tick ({v':NEWavl | rkDiff v' t 1 && isNode1_2 v' }) |
6                   amortized1 t v } @-}
7   promoteL :: Tree a -> Tick(Tree a) -> Tick (Tree a)
8   promoteL t@(Tree a n _ r) l =
9           Tick (tcost l + 1) (Tree a (n+1) (tval l) r)
```

Proof. similar to the proof 4.4, we look at the 4 cases of the rebalancing process to show the potential changes taking place:

- **demote**: changes a 3,2-node to a 2,1-node, decreasing the potential by 2.
- **double demote**: changes a 3,1-node with a child 2,2-node to a 2,1-node with a child 1,1-node, thus decreasing the potential by 1.
- **rotate**: Depending on the structure, this generates a new 2,2-node, thus increasing the potential by at most 2. After a rotation, if z is demoted again and becomes a leaf, y becomes a 2,2-node, resulting in a potential increase of 2.
- **double rotate**: depending on the structure of the children, the potential is increased most when the v is a 1,1-child. Then we get a potential increase of at most 3.

For all cases, mirror structures apply. We find that the total increase per rebalancing process is at most 3 (caused by the rotations), and thus, we end up by definition of the potential analysis and our code being proven safe by LH that theorem 2 holds for the delete case. □

The code follows a similar way to the insert function, for details see appendix 5.2.

4.5 Remarks on the potential cost analysis approach

In Theorem 2, we find a few advantages to the original theorems:

- we do not have to make a case for how often a method is executed because our Theorem addresses the combined value over all actions
- With a combined analysis of overall function calls, we can argue that at most per function call amortised over all function calls, a better amortised value of 3 rebalancing steps per call

Thus, we can define the following Lemma:

Lemma 1. *For a WAVL tree with bottom-up rebalancing, there are amortised at most 3 demote or promote steps for insert and delete function calls.*

1 follows from the Proof 2.

Remark. *In our analysis, we omitted the cost for the concrete insertion, the rotation and the search of the insertion spot / to be deleted node. This is grounded in*

- *the search for the node is limited by the height/rank of a tree, s. the proof for that in 4.6.1*
- *The insertion and the rotation steps are constant in relation to the promotion/demotion steps.*

Our analysis shows that the demote and promote cases are indeed bounded and in conclusion, we show that the total amortised cost for the rebalancing process is bounded by $O(1)$ steps.

4.6 logarithmic height

But wait, we only ever proved that the *rebalancing* steps are in $O(1)$. What about the actual runtime costs of insert and delete? We prove the Termination of these functions via the rank function rk. So, an execution of inserting a leaf on a given tree will reach a leaf after at most rk steps within a full tree. But how does the rank relate to the amount of nodes in the tree? Similar to Red-Black trees, an initial guess would say that rank behaves similarly to the black height of Red-Black trees, [GS78].

Thus, we define the height of a tree as in Listing 4.25 with the max function as in Listing 4.26.

Listing 4.25: Height property

```
{-@ measure height @-}
{-@ height :: t:Tree a ->
    {r:Int | ((empty t) <=> (r = -1)) && ((not (empty t)) <=> (r >= 0))}
@-}
height :: Tree a -> Int
height Nil = -1
height (Tree _ _ l r) = 1 + max (height l) (height r)
```

Listing 4.26: Max function

```
{-@ inline max @-}
max :: Ord a => a -> a -> a
max a b = if a >= b then a else b
```

The height of a tree is the same as the worst-case execution path in a tree. Further, we define the number of nodes in a tree as follows:

Listing 4.27: tree size function

```
{-@ measure size @-}
{-@ size :: Tree v -> Nat @-}
size :: Tree v -> Int
size Nil = 0
size (Tree _ _ l r) = 1 + size l + size r
```

Comparing the rank function *rk* and height, one can see that we defined them similarly and define thus the following lemma 2:

Lemma 2. *A WAVL tree's rank is at least its height and at most 2 times its height.*

Listing 4.28: Height Lemma 2 codefied

```
{-@
thm_height :: {t:Wavl | not (empty t)}
    -> {height t <= rk t && rk t <= 2 * height t}
@-}
thm_height :: Tree v -> ()
thm_height (Tree _ _ Nil Nil) = ()
thm_height (Tree _ 1 ll rr) = case (ll, rr) of
                                (Nil, Tree _ _ _ _) -> ()
                                (Tree _ _ _ _, Nil) -> ()
                                (Tree _ 0 Nil Nil, Tree _ 0 Nil Nil) -> ()
thm_height (Tree _ r ll rr) | height ll >= height rr = thm_height ll
                            | otherwise = thm_height rr
```

The proof for Lemma 2 is trivially proven with LH, as can be gauged from the many empty proof clauses () in code 4.28.

4.6.1 Logarithmic height

So now that we have the rank and the height connected via Lemma 2, we can go on to show that both are bound logarithmically, i.e. $O(log\ n)$.

Theorem 3. *A WAVL Tree of size n is bounded in its rank r by the formula:*

$$r \leq 2 * log_2(n)$$

For the proof 4.6.1, we need to prove the Corollary 1 to show that Theorem 3 holds.

Corollary 1. *For a WAVL tree of size n and rank r the following inequality holds:*

$$n \geq 2^{\lceil \frac{r}{2} \rceil}$$

In Haskell or LiquidHaskell, the mathematical symbols for $\lceil\ \rceil$, the logarithmic function with base 2 *log*2 and the exponential function are not defined. For them to work and for the proof, we need definitions of their functional behaviour and, more importantly, their monotonicity. Furthermore, the relationship between logarithmic and exponential functions with the same base must be added to our code.

Proof. The Theorem 3 is encoded with 4.29 and relies on PLE. The proof follows from the LH check showing soundness. □

Listing 4.29: Proof of Logarithmic Height with LH

```
{-@
thm_size :: {t:Wavl | not (empty t)} -> {rk t <= 2 * log2 (size t)}
@-}
thm_size :: Tree v -> ()
thm_size t@(Tree _ _ _ _) = prove $
                          (log2 (size t) >= log2 (pow2 (ceilDiv2 (rk t)))
                              ? ( thm_size_help t
                               , log2_mon (pow2 (ceilDiv2 (rk t))) (size t)))
                      && (log2 (size t) >= ceilDiv2 (rk t)
                              ? log2_pow2 (ceilDiv2 (rk t)))
```

and the corollary 1 is written thus as in listing 4.30.

Listing 4.30: Size to Rank relation proof of Corollary 1

```
{-@
co_size2rank :: {t:Wavl | not (empty t)}
                    -> {size t >= pow2 (ceilDiv2 (rk t))}
@-}
co_size2rank :: Tree v -> ()
co_size2rank (Tree _ 0 ll rr) = ()
co_size2rank (Tree _ 1 ll rr) = case (ll, rr) of
                                  (Nil, Tree _ _ _ _) -> ()
                                  (Tree _ _ _ _, Nil) -> ()
                                  (Tree _ _ _ _, Tree _ _ _ _) -> ()
co_size2rank t@(Tree _ r ll rr) =
    prove $
            (ceilDiv2 (r - 2) == (ceilDiv2 r) - 1
                ? thm_ceilDiv2_minus2 r)
        && (size ll >= pow2 (ceilDiv2 r - 1)
                ? ( co_size2rank ll
                , thm_ceilDiv2_mon (rk ll) (r - 2)
                , pow2_mon (ceilDiv2 (rk ll)) (ceilDiv2 (r - 2))))
        && (size rr >= pow2 (ceilDiv2 r - 1)
                ? ( co_size2rank rr
                , thm_ceilDiv2_mon (rk rr) (r - 2)
                , pow2_mon (ceilDiv2 (rk rr)) (ceilDiv2 (r - 2))))
```

Let's break that apart:

- *ceilDiv*2 is the equivalent of $f(a) = \lceil \frac{a}{2} \rceil$
- *log*2*_mon*, *pow*2*_mon* and *thm_ceilDiv*2*_mon* describe the monotonicity of the functions and are needed to prove the inequalities between functions of the same type (s. implementation at Listing 5.10)

- $log2\_pow2$ defines the relationship between logarithmic and exponential function, i.e. $log_2(2^a) = a$

The proof for the corollary is longer since there are more *jumps* between the inequalities to come to a conclusion, while the code proof 4.29 relies heavily on the Corollary 1.

On the part of the concrete implementation and details about the accompanying proofs on monotonicity of $log2$, $pow2$, and $ceilDiv2$, refer to the appendix section 5.1.

CHAPTER 5

Conclusio

We verified the amortised cost analysis on the rebalancing process of WAVL trees and added additional proofs for the functional correctness and the logarithmic height of such trees. Doing so gives developers more confidence in using these data structures in their code. Comparing current real-life applications between AVL, Red-Black and B-trees, we found a distinct mistrust against AVL trees because their worst-case rebalancing is not provable in $O(1)$ but in $O(log\ n)$.

5.1 future work

Using the rank-based framework and our approach to verify and maybe even improve upon the existing pen-and-paper proofs of the following tree structures:

1. Top-down rank-balanced trees according to [HST15]
2. general-balanced Trees, Scapegoat Trees and relaxed balanced trees [And99, IG93, STK16]
3. formalizing different implementations of Red-Black-trees [HOSS97, Lar02]

1. and 2. improve upon current data structures by showing better concurrent access and delete management. The 3. item in the list consists of work to prove different approaches and constraints put on forms of the original Red-Black tree to get different performance gains. The respective papers were often written way back in the 80's and 90's of the old century. With current techniques, one could re-check these approaches and re-prove or disprove some claims made back then.

Appendix

merge function

The function *merge* in our first version suffered similar problems to the *delete* function 4.8, i.e., we could not constrain the case expansion need with stricter refinements:

```
{-@ merge :: x:a -> l:Wavl -> r:Wavl
  -> {v:Rank | WavlRankN v l r && v >= 0 }
  -> {t:Wavl | balLDel v t || RkDiffN v t 1 } @-}
merge :: a -> Tree a -> Tree a -> Int ->  Tree a
merge _ Nil Nil _ = nil
merge _ Nil r _   = r
merge _ l Nil _   = l
merge x l r n     = (balRDel y n l r')
  where
   (r', y)      = getMin r
```

Similarly, we had to define the *getMin* function, which finds the immediate successor of a node's value.

```
{-@  getMin :: v:NEWavl
  -> ({t:Wavl | (rkDiff t v 0) || (rkDiff v t 1) }, a) @-}
getMin :: Tree a -> (Tree a, a)
getMin (Tree x 0 Nil Nil) = (nil, x)
getMin (Tree x 1 Nil r@(Tree _ _ _ _)) = (r, x)
getMin (Tree x n l@(Tree _ _ _ _) r@Nil) =
    ((balLDel x n l' r), x')
    where
    (l', x')              = getMin l
getMin (Tree x n l@(Tree _ _ _ _) r) = ((balLDel x n l' r), x')
    where
    (l', x')              = getMin l
```

The observant reader probably has already guessed that we can also use the immediate predecessor to replace the to-be-deleted node. Functionally, this would only require the implementation to go down the left branch once and then follow along the right-hand side until a leaf is encountered. This implementation is symmetric to ours.

rebalancing Delete with balDel

Listing 5.1: balLDel refinement: rebalance step for delete, version 1

```
balLDel :: a -> {n:Rank | n >= 0 }
  -> {l:Wavl | Is3ChildN n l}
  -> {r:MaybeWavlNode | Is2ChildN n r}
  -> {t:NEWavl | (rk t == n || rk t + 1 == n) }
```

balLDel and its symmetric case are the rebalancing steps in this scenario. The function output is the same as for the *delete* function in 4.9. To note is the additional hoops we had to take, i.e. we needed to define a nearly WAVL Tree with $MaybeWavlNode$, which defines a Tree which is either empty or follows the (in version 2 more centrally used) Lemma of being a WAVL node structurally, i.e. either being one of the four allowed structures like in listing 4.12. This is our first approach to the *structLemma* and it is not used throughout the whole function refinement, only at places where the structLemma is needed to constrain the input/output for like the 2,2-node clause, i.e. which are not allowed at rank 1.

These experiments with *structLemma*, even though we did not call it like that at that point in time, are crucial to our later simplified version 2.

Leaf nodes

```
{-@ singleton :: a -> {v:NEWavl | ht v == 0 && rk v == 0 } @-}
singleton a = Tree a 0 Nil Nil
```

Wavl tree, version 2

The updated version of delete made the explicit *merge* definition obsolete since enough constraint information is available at the time of calling it in the second version at 4.16.

Further, from the fact that we do not have to check the inputs

```
{-@  getMin :: t:NEWavl
  -> ({v:Wavl | (rkDiff t v 0) || (rkDiff t v 1) }, a) @-}
getMin :: Tree a -> (Tree a, a)
getMin (Tree x _ Nil r) = (r, x)
getMin t@(Tree x n l r) = (delL t l', x')
  where
    (l', x')              = getMin l
```

Amortised cost analysis, version 2

We were unable to prove the amortised cost analysis with our first approach with version 1. After redoing the whole data type definition and finding solutions for insert and delete we found that the cost analysis could also be stated in a relatively concise manner.

In this section, we enumerate some additional parts of our cost analysis to give a better overview of what the single parts are responsible for.

The code snippets can all be found at our Github Repo [Gen24].

insert parts analyzed

```
{-@ inTreeL :: t:NEWavl
  -> {l:Tick (NEWavl) |
     ((rkDiff (tval l) (left t) 0 ||
       rkDiff (tval l) (left t) 1))
    && rk (tval l) < rk t
    && amortizedStmt (left t) l }
  -> {v:Tick ({v':NEWavl | rkDiff t v' 0}) |
       rkDiff t (tval v) 0 && amortized3 t v } @-}
inTreeL :: Tree a -> Tick (Tree a) -> Tick (Tree a)
inTreeL t@(Tree x n _ r) l =
  Tick (tcost l) (Tree x n (tval l) r)
```

The *inTreeL* function is used in the insert for one, limiting the input of the recursion so that we can return `Tick (tcost l) (Tree x n (tval l) r)`. Furthermore, we want to state that the *amortisedStmt* holds. This function call is essentially used to prove that after a rotation case was executed on the tree, no more potential is changed and no costs are added. This can be seen if you compare the input l tree, which is the same refinement as the output of *insert*, i.e. 4.21.

```
{-@ rotateRight :: {t:NEWavl | isNode1_2 t}
  -> {l:Tick (NEWavl) | rk (tval l) == rk t
    && isNode1_2 (tval l) && amortized1 (left t) l}
  -> {v:Tick ({v':NEWavl | rkDiff t v' 0 }) |
    rkDiff t (tval v) 0 && amortized3 t v} @-}
rotateRight :: Tree a -> Tick (Tree a) -> Tick (Tree a)
rotateRight t@(Tree x n _ c) (Tick tl (Tree y m a b)) =
                Tick tl (Tree y m a (Tree x (n-1) b c))
```

```
{-@ rotateDoubleRight :: {t:NEWavl | isNode1_2 t}
  -> {l:Tick (NEWavl) | rk (tval l) == rk t
    && isNode2_1 (tval l) && amortized1 (left t) l}
  -> {v:Tick ({v':NEWavl | rkDiff t v' 0 }) |
    rkDiff t (tval v) 0 && amortized3 t v}
@-}
rotateDoubleRight :: Tree a -> Tick (Tree a) -> Tick (Tree a)
rotateDoubleRight (Tree z n _ d)
                  (Tick tl (Tree x m a (Tree y o b c))) =
  Tick tl (Tree y (o+1) (Tree x (m-1) a b) (Tree z (n-1) c d))
```

The rotation cases for *insert* take in only 2,1-nodes (mirror cases apply). As can be seen in the insert refinement 4.21, we need the concrete node structure as an input child tree to exchange.

Remark. *Note the pattern matching in the rotation cases. Because of LH's way of only proving a function refinement and then only reusing the refinement logic all parts in the dependent parts, we found that we had to pattern match the Tick monad as in 5.1 and 5.1. In our opinion, this is also the more Haskell way of doing it and using pattern matching instead of using logical predicates on types helps to keep proofs shorter since LH does a lot of the proving for us, i.e. data constructors are instantiated as measure in the logic automatically and proved as such.*

```
{-@ inline amortized3 @-}
{-@ amortized3 :: Wavl -> Tick (Wavl) -> Bool @-}
amortized3 :: Tree a -> Tick (Tree a) -> Bool
amortized3 t v = potT t + 3 >= tcost v + pot v
```

```
{-@ inline amortized @-}
{-@ amortized :: Wavl -> Tick (Wavl) -> Bool @-}
amortized :: Tree a -> Tick (Tree a) -> Bool
amortized t v = potT t >= tcost v + pot v
```

The amortised statement is implemented as an *inline* in the LH logic so it can be reused in the refinements of *insert*, *promote* and so on. By proving this sound via LH, we get our proof of the rebalancing process in 4.4.

```
{-@ measure potT @-}
{-@ potT :: t:Wavl -> Nat @-}
potT :: Tree a -> Int
potT Nil       = 0
potT t@(Tree _ n l r)
  | isNode1_1 t && n > 0      = 1 + potT l + potT r
  | child2 t l && child2 t r = 2 + potT l + potT r
  | otherwise                 = potT l + potT r

{-@ inline pot @-}
{-@ pot :: Tick (Wavl) -> Nat @-}
pot :: Tick (Tree a) -> Int
pot t = potT (tval t)
```

The potential function is the coded implementation of proof 4.4, and we only really use it comparatively, i.e. compare the potential difference in *amortised* functions between input and output trees, s. 5.1, 5.1.

proof of amortised cost for delete

Similar to the insert in this section we will explain the code parts for the proof of the amortised cost analysis for the *delete* function.

As stated in proof 4.4, we define the updated delete function as in listing 5.2. The algorithm follows the same line as the original one without the monad. In the refinement, we find the amortised statement for the *delete* function at work, i.e. compare listing 5.3.

The amortised statement follows the same thought line for *insert*, i.e. as long as the rebalancing process is ongoing, represented by a changed rank difference (rkDiff). At the same time, as soon as this condition doesn't hold, we know that with a worst case, a rotation will change the potential by 3, ending the rebalancing process.

Listing 5.2: delete with Tick monad and amortised Statement

```
{-@ delete :: (Ord a) => a -> t:Wavl
  -> {v:Tick ({v':Wavl | ((rkDiff t v' 0) || (rkDiff t v' 1))}) |
          amortDelStmt t v}
@-}
delete::  (Ord a) => a -> Tree a -> Tick (Tree a)
delete _ Nil = pure Nil
delete y t@(Tree x n l r) = case compare x y of
    LT -> delL t l'
    GT -> delR t r'
    EQ -> merge
    where
        l' = delete x l
        r' = delete x r
        merge
            | empty r = pure l
            | otherwise = let (r'', z) = getMin r
                          in delR (Tree z n l r) r''
```

Listing 5.3: Amortised Statement for *delete*, compare it to code 4.23

```
amortDelStmt :: Tree a -> Tick (Tree a) -> Bool
amortDelStmt t v = ((rkDiff t (tval v) 0) || (amortized t v) )
                    && (rkDiff t (tval v) 1 || (amortized3 t v))
```

Proof of the logarithmic Height

In the following, we describe some parts of the proof in LiquidHaskell concerning the respective code parts.

Logarithmic Axioms

In the Proof 4.6.1 on the logarithmic Height we use some axioms on the monotonicity of the logarithmic function and the relation between the exponential and the logarithmic function. The definition of $log2$ is recursively defined at 5.4 and the basis. We only use the logarithm of 2 in our code base since we don't need any other. Similarly, the exponential function for base 2 $pow2$ is defined in 5.6. A full proof without the simplifications as we use it with the PLE is done in [Hoc24]. The same is done for the monotonicity axiom in 5.5. To note is that we use the Haskell default implementation of *div*, [1].

[1]URL: https://hackage.haskell.org/package/ghc-internal-9.1001.0/docs/src/GHC.Internal.Real.html#div

Listing 5.4: logarithmic function for base 2

```
{-@
reflect log2
log2 :: {i:Int | i >= 1} -> {r:Nat | r < i}
@-}
log2 :: Int -> Int
log2 1 = 0
log2 n = 1 + log2 (div n 2)
```

Listing 5.5: proof for logarithmic monontonicity

```
{-@
log2_mon :: {a:Int | a >= 1}
          -> {b:Int | a <= b}
          -> {log2 a <= log2 b}
@-}
log2_mon :: Int -> Int -> ()
log2_mon 1 x = const () (log2 x >= 0)
log2_mon _ 1 = ()
log2_mon x y = log2_mon (div x 2) (div y 2)
```

Listing 5.6: exponential function 2^a

```
{-@ reflect pow2 @-}
{-@ pow2 :: Nat -> Pos @-}
pow2 :: Int -> Int
pow2 0 = 1
pow2 n = 2 * (pow2 (n - 1))
```

Listing 5.7: proof for relation between $log_2(a)$ and 2^a

```
{-@ log2_pow2 :: n:Nat -> {log2 (pow2 n) = n} @-}
log2_pow2 :: Int -> ()
log2_pow2 0 = ()
log2_pow2 n = log2_pow2 (n-1)
```

Listing 5.8: proof to exponential monontonicity

```
{-@ pow2_mon :: a:Nat -> {b:Nat | a >= b} -> { pow2 a >= pow2 b } @-}
pow2_mon :: Int -> Int -> ()
pow2_mon 0 _ = ()
pow2_mon n 0 = pow2_mon (n-1) 0
pow2_mon n k = pow2_mon (n-1) (k-1)
```

Listing 5.9: Ceil operation for division by 2

```
  {-@
  reflect ceilDiv2
  ceilDiv2 :: n:Nat -> {v:Nat | 2 * v >= n}
  @-}
  ceilDiv2 :: Int -> Int
  ceilDiv2 n | mod n 2 == 0 = div n 2
             | otherwise = check (mod n 2 == 1) (1 + div n 2)
```

Listing 5.10: monotonicity Theorem for the ceil and div function combo

```
{-@
thm_ceilDiv2_mon :: a:Nat -> {b:Nat | a >= b}
  -> {ceilDiv2 a >= ceilDiv2 b}
@-}
thm_ceilDiv2_mon :: Int -> Int -> ()
thm_ceilDiv2_mon a b | a == b = ()
                     | mod a 2 == 0 && mod b 2 == 0 = ()
                     | mod a 2 == 0 && mod b 2 == 1 = ()
                     | mod a 2 == 1 && mod b 2 == 0 = ()
                     | mod a 2 == 1 && mod b 2 == 1 = ()
```

reproducibility of the proofs

The code was compiled with GHC 9.4.7 and LiquidHaskell 0.9.4.7. The original project was done under Ubuntu 20.04 LTS but should be reproducible with other versions.

The project uses stack for package management and was based on the demo project by the LiquidHaskell team.

at `https://github.com/ucsd-progsys/lh-plugin-demo`.

As an IDE, we used Visual Studio code with a plugin for GHCi integration to show LH errors directly on our IDE. A complete installation guide for it can be found at `https://github.com/Genlight/wAVL-trees/blob/main/install_env.md`.

Overview of Tools used

In this work i used only two generative tools for assistance:

- ChatGPT v4
- Grammarly

I used ChatGPT for getting a rough outline on my acknowledgements and chapters 1 and 2 but heavily redacted the texts. Grammarly is a tool for spell-checking and giving alternative text phrases. I used it on the whole text mostly for grammar and spelling checks but also some re-phrasing of some passages.

List of Figures

Bibliography

[And99] Arne Andersson. General balanced trees. *Journal of Algorithms*, 30(1):1–18, 1999.

[CFG19] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Non-polynomial worst-case analysis of recursive programs. *ACM Transactions on Programming Languages and Systems*, 41(4):1–52, 2019.

[GAV62] Evgenii Landis Georgy Adelson-Velsky. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 1(3):263–266, 1962.

[Gen24] Alexander Genser. Proving theorems about weak avl trees: https://github.com/genlight/wavl-trees, 2024.

[GS78] Leo J Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21. IEEE, 1978.

[Hoc24] Jamie Hochrainer. Formalizing an amortized cost analysis of binomial heaps and fibonacci heaps in liquid haskell, 2024.

[HOSS97] Sabine Hanke, Thomas Ottmann, and Eljas Soisalon-Soininen. Relaxed balanced red-black trees. In *Italian Conference on Algorithms and Complexity*, pages 193–204. Springer, 1997.

[HST09] Bernhard Haeupler, Siddhartha Sen, and Robert E Tarjan. Rank-balanced trees. In *Workshop on Algorithms and Data Structures*, pages 351–362. Springer, 2009.

[HST15] Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. Rank-balanced trees. *ACM Transactions on Algorithms*, 11(4):1–26, 2015.

[HVH19] Martin A. T. Handley, Niki Vazou, and Graham Hutton. Liquidate your assets: reasoning about resource usage in liquid haskell. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–27, 2019.

[Hø14] Andrea Høfler. Smt solver comparison. *Graz, July*, page 17, 2014.

[IG93] R. L. Rivest Igal Galperin. Scapegoat trees. *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 165–174, 1993.

[JSV15] Ranjit Jhala, Eric Seidel, and Niki Vazou. Programming with refinement types (an introduction to liquidhaskell), 2015.

[Lar02] Kim S Larsen. Relaxed red-black trees with group updates. *Acta informatica*, 38(8):565–586, 2002.

[Nip17] Tobias Nipkow. Verified root-balanced trees. pages 255–272. Springer, 2017.

[Pal20] Josh Cohen; Paul Palmer. Verified binomial and skew heaps using liquidhaskell, 2020.

[STK16] Siddhartha Sen, Robert E. Tarjan, and David Hong Kyun Kim. Deletion without rebalancing in binary search trees. *ACM Transactions on Algorithms*, 12(4):1–31, 2016.

[Tar85] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[Vaz23] Niki Vazou. On the practicality and soundness of refinement types. In Carlos Ciaffaglione, Alberto; Olarte, editor, *Proceedings of the 18th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, Cornwell University, 2023. EPTCS 396.

[VG22] Niki Vazou and Michael Greenberg. How to safely use extensionality in liquid haskell. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, pages 13–26, 2022.

[VSJ14] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. Liquidhaskell. *ACM SIGPLAN Notices*, 49(12):39–51, 2014.