Bachelor's Thesis Computing Science

Radboud University Nijmegen

Verifying Structural Red-Black Tree Invariants using Liquid Haskell

Author:
Pelle Meijer
s1037691

First supervisor/assessor:
Dr. Niels van der Weide

Second assessor:
Prof. dr. Herman Geuvers

June 21, 2025

Abstract

This study explores an application of Liquid Haskell, a refinement type system for Haskell, to formally verify the red and black invariants of red-black trees. Specifically we verify the preservation of the red and black invariants for Okasaki's functional insertion algorithm.

Red-black trees are self balancing binary search trees, and maintaining these invariants is crucial for guaranteeing logarithmic time complexity in worst case scenarios, where operations on unbalanced binary search trees can have linear time complexity.

By leveraging Liquid Haskell's ability to augment standard Haskell types with predicates, we demonstrate the preservation of the red and black invariants for the provided insertion implementation. The implementation and verification are worked out step by step, and we include both the process and intuition for this process. This approach offers insights into both the inner workings of red-black tree insertion, as well as a practical example on how to approach writing verifications using Liquid Haskell.

Contents

Chapter 1

Introduction

Binary search trees are data structures that aim to provide efficient implementation of sets, offering logarithmic average time complexity for operations like insertion, deletion and lookup [5]. In order to extend this average time complexity to include the worst cases, we must look towards self-balancing versions of these binary search trees.

One example of self-balancing binary search trees is red-black trees [8]. Red-black trees solve the worst case time complexity for binary search trees by ensuring the trees are balanced, meaning they implement systems to limit the maximum height of the tree. These trees are important components in the implementations of sets in numerous libraries and systems, ranging from standard library components in programming languages [18] to components in operating systems [20] and database management systems [19]. Therefore, ensuring the integrity of these data structures during operations is crucial for ensuring they keep their balanced properties.

In order to to make sure red-black trees remain balanced, two invariants are necessary. We call these the *red invariant* and the *black invariant.* While it is trivial to prove that read-only operations preserve these invariants, proving the same for operations like insertion and deletion is a lot harder.

Manually proving the correctness of complex data structure operations can be error-prone and very time-consuming. Formal verification using a proof assistant or similar alternatives offers a more rigorous approach to guarantee the preservation of these crucial properties, resulting in more reliable software.

While the theory around red-black trees is already well-established, and there has been research into various methods for reasoning about data structure correctness, we propose integrating practical implementation together with formal verification. We achieve this by using Liquid Haskell, a Haskell plugin by Vazou [23].

Liquid Haskell allows formal reasoning about Haskell functions and programs, acting as a middle man between Haskell and an SMT solver. It achieves this through the concept of *refinement types*, which essentially are predicates added to standard Haskell types to allow formal reasoning about these types. For example, we can specify not only that the color of a node has to be `Red` or `Black`, but further refine a type in such a way that if a node is red, its children may not be red. Using these refinement types to verify the preservation of red and black invariants, means we have both a practical implementation as well as a formal verification, bundled together in one program. This means that any simple refactor to the code gets verified at compile time with little to no extra work.

We use a combination of standard Haskell functions and Liquid Haskell refinement types to define the red and black invariants, and use these to describe both full red-black trees, as well as a weaker, partial red-black tree which shows up during some parts of the insertion. With these definitions we can write refinement types that fully check these invariants for our functions, verifying they perform as expected.

We start with a chapter explaining red-black trees, Haskell and how to implement red-black trees in Haskell (Chapter 2). We then move on to a chapter explaining the use and some of the inner workings of Liquid Haskell (Chapter 3). We then combine the information from the previous two chapters, and show how to verify both a smart constructor, and the more complex insertion function (Chapter 4). Next we discuss related works, (Chapter 5), after which we end with a concluding chapter, summarizing the work and findings described in this thesis, while also discussing some possibilities for future research in this direction (Chapter 6).

Chapter 2

Red-black Trees in Haskell

In this chapter we start by describing the basic principles and syntax of Haskell, our functional programming language of choice for this study. We then explain what red-black trees are, how and why they work, as well as how to implement them in a functional environment like Haskell.

2.1 Haskell

We start by discussing the Haskell programming language. We first describe its core principles and ideas, giving a couple small code snippets to illustrate some of these ideas. We then give a short overview of the Haskell syntax, focusing primarily on the aspects used in the implementation described in this thesis.

Core Principles

Haskell is a purely functional language, designed according to the principles outlined in the report on the programming language Haskell [9]. Haskell was designed in a time when many purely functional programming languages started to appear, like Miranda [21] and Hope [4]. Haskell was designed to be the 'common' language used for purely functional programming.

Haskell is a purely functional programming language. This means that functions in Haskell are typically *pure*, meaning they behave like mathematical functions: they deterministically take inputs and produce outputs, without causing any *side effects*. Side effects are actions like modifying variables, reading from files, printing to console, or changing mutable state. While real-world applications often require side effects, Haskell elegantly isolates them primarily to the `IO` type. This isolation is achieved through the use of *monads*. Monads, like the `IO` monad, provide a structured way to sequence computations that may involve effects. Within the pure logic of Haskell, computations are referentially transparent, that is, any expression can be replaced by its value without changing the program's behavior. Monads allow us to "encapsulate" impure actions within a pure context, allowing the pure parts of our code to remain predictable and composable, while still enabling interaction with the outside world in a controlled manner.

Haskell uses a strong, static type system. Types are checked at compile time, catching many potential errors before the program runs. A powerful feature is type inference, where the compiler can deduce the types of expressions without explicit annotations from the user. For instance, defining:

```haskell
addOne x = x + 1
```

Haskell uses the type of (+) and 1 to infer that x must be some kind of Num, and the total function must have a signature like:

```haskell
addOne :: (Num a) => a -> a
```

This essentially means it is a function, where a represents an instance of Num (number), and the function has a signature such that it takes some value of type a (which is a number) and returns some value which is also of type a (also a number).

Another feature distinguishing Haskell from many other languages is *lazy evaluation*. Expressions are not evaluated until their results are actually needed. This allows for the definition of conceptually infinite data structures, as well as possibly leading to performance benefits by avoiding unnecessary computations. For example, the following is a perfectly valid function:

```haskell
ones :: [Integer]
ones = 1 : ones
```

This definition uses the list constructor (:) to create a list of Integers, such that the first element is one, and the remainder of the list is the list 'ones'. This essentially means a list filled with an infinite amount of 1's. However this definition does not cause an infinite loop, because Haskell only computes as far as required. For example:

```haskell
ones !! 2 -- !! is the access operator in haskell, similar
          -- to ones[2] in languages like Python or JavaScript
```

This function calculates the list until:

```haskell
1 : ( 1 : ( 1 : ones ) )
```

Then uses that definition to take the third element from the list, since Haskell is 0-index based.

Another important aspect of Haskell is that of data immutability. Once a value or data structure has been created, it cannot be changed. Operations that seem to 'modify' data are actually creating new data structures with the changes applied, leaving the original data untouched.

This immutability has significant implications for how state is managed in Haskell. In many programming paradigms, programs maintain state by modifying variables over time. However, since Haskell values cannot be changed after creation, the traditional notion of mutable state does not directly apply. Instead, when a computation needs to evolve or manage a sequence of values that appear to change, Haskell utilizes the State monad. The State monad provides a way to encapsulate state within a computation, allowing operations that conceptually modify this state without actually mutating any values. It achieves this by explicitly carrying the state along through the computation as an additional value.

This approach, combined with the purity of functions, makes it easier to reason about program behavior, as the value of an expression depends only on its inputs, not on some hidden global state that might have been altered elsewhere. While Haskell does allow for managing computations that involve state, it does so in a controlled and explicit manner, keeping the core of the language pure and immutable.

Syntax

In order to understand the implementation and verification discussed in this thesis, it is important to have at least a basic understanding of Haskell syntax. This section gives a short overview of most of the Haskell-specific syntax used throughout this thesis.

Haskell allows for the creation of custom *data types* using the *data* keyword. These can be simple enumerations, or more complex structures with multiple fields. We can also use the *deriving* keyword to automatically ensure our type is an instance of another typeclass, like `Show`, which is used for printable representation, or `Ord`, which is used for ordered elements, where elements can be compared in some way to see whether they are equal to, larger or smaller than each other.

```haskell
data Color = Red | Black
  deriving (Show, Eq)

data Coordinate a = Coordinate a a

data Shape = Circle Float | Rectangle Int Int
```

Here we see three examples of custom data types. The first shows a data type `Color`, which can either be red or black. This data type derives Show and Eq, meaning we can compare whether two colors have the same value, and we can get a printable representation. Second we see a `Coordinate` type, with a generic parameter `a`. This means the type `a` represents any type, so we could have a coordinate consisting of two Ints, two Floats or even two Chars. Last is the data type Shape, which has two constructors, either a circle which requires a Float, which could for example represent the radius, while rectangle requires two Ints to represent width and height.

Functions in Haskell often have *type signatures.* While not mandatory, they can greatly improve the readability of the code, and can help the compiler catch type errors. Type signatures show what types are expected as inputs, and what type the resulting output of a function would be:

```
seven :: Int
seven = 7

add :: Int -> Int -> Int
add x y = x + y

isGreater :: (Ord a) => a -> a -> Bool
isGreater a b = a > b
```

The first example is a function that simply returns the integer 7 when called, which has a type signature of simply returning an integer without any input parameters. Second is an addition function. This function takes an integer, then another integer, and finally returns the added values of both integers. In Haskell, all functions are considered *curried.* This means that all functions actually only take one argument. The way this works with functions that seem to take multiple arguments, is that the function taking two integers and returning an integer actually has the following type signature:

```
add :: (Int, Int) -> Int
```

That is, the `add` function is actually a function that takes a single tuple, then adds both its members together. We say the function seeming to take two arguments is the *curried* form of the function taking a single (tuple) argument. This curried form is most often used, as it allows for *partial function application*, which is the idea in Haskell that not providing all function arguments is still valid, but simply returns another function requiring any arguments that were not provided initially. For example, `add 1` returns a function that adds 1 to an input Integer. Lastly, the `isGreater` function makes use of the `(Ord a) =>` notation. This means the generic type `a` has a constraint where it has to be part of the Ord typeclass.

Haskell functions often make use of the *pattern matching* syntax. A function can use a case-like syntax to give a different result when provided a specific input, for example:

```
div :: (Fractional a) => a -> a -> a
div x 0 = error "cannot divide by 0"
div x y = x / y
```

This division function first checks whether the second input is 0, and if it is, throws an error. A similar way of achieving this is through *guard clauses*:

```
div x y
  | y == 0 = error "cannot divide by 0"
  | y > 10 = y / x
  | otherwise = x / y
```

The advantage of guard clauses is that it allows more freedom in the condition, where pattern matching only allows matching on specific structures or values. The otherwise keyword is used in guard clauses as syntactic sugar for `True`, catching all uncaught cases.

Since Haskell does not have mutable variables, the principle of for loops, used in many mainstream languages, cannot exist. Instead most repeating functions make use of recursion:

```
printXTimes :: Int -> String -> IO ()
printXTimes 0 s = print s
printXTimes x s = printXTimes (x-1) (s ++ s)
```

This function uses a counter to recursively count down to 0, where it prints the input string. While at first glance this may seem like it would print something X times, this function actually doubles the string X times before printing. It is important to keep in mind that changing the parameters means the change is run every loop.

This is only a very basic overview of Haskell's syntax, as there are many more parts of Haskell that we have not discussed here [13, 10]. Many of these other concepts are just as important, if not more so, than what was discussed in this section. However the concepts discussed here give a good overview of the basics, and are enough to understand the implementation and verification discussed in this thesis.

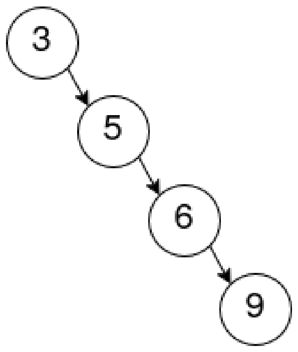

Figure 2.1: A valid binary search tree with $\mathcal{O}(n)$ lookup complexity.

2.2 Red-Black Trees

Binary Search Trees [5] are a fundamental data structure, consisting of *nodes*. Nodes are elements of these trees, where each node has a value to be stored, as well as two children, where each child is either a leaf or another node. This leads to a recursive data structure, that can grow exponentially in size for each layer of height it contains. Each node also has the constraint that all values stored in the left child and its children should be smaller than said node, and all values stored on the right side should be larger. In this way binary search trees are ordered based on the values of their elements. Binary search trees offer efficient average performance for searching, insertion and deletion operations. However, an inefficiently constructed binary search tree can lead to a linear structure, resulting in these operations going from their average case of $\mathcal{O}(\log n)$ to simply being $\mathcal{O}(n)$ time complexity (see Figure 2.1). This is because these algorithms scale based on the height of the tree. All of these algorithms work by traversing through the tree from the top down. The height of a binary search tree with n elements can range from $\log n$ to n, where a tree with height $\log n$ would be perfectly balanced, each node having two children until the final layer, and a tree with height n would be a linear tree.

In order to improve the worst case scenario, various self-balancing binary search trees have been developed, for example AVL trees [1], AA trees [2], and red-black trees [8]. These kinds of trees have specific systems in place in order to ensure that the height of the tree remains small in order to avoid the worst case scenario operations. This study specifically covers red-black trees.

A red-black tree is a binary search tree where each node stores one additional bit representing their color. This is either red or black. This additional property is used to ensure balance in the tree, by stating that any red-black tree must satisfy the following properties:

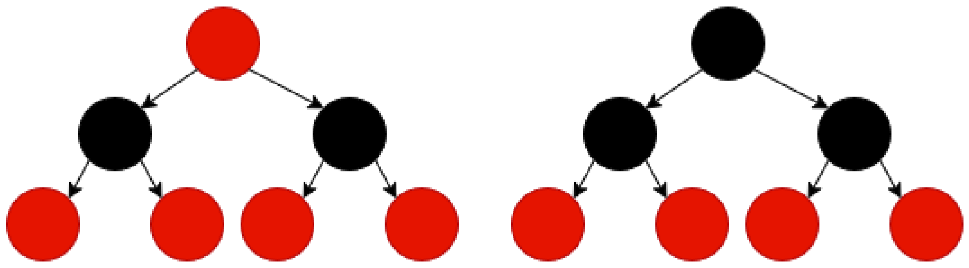

Figure 2.2: Changing root node to black does not violate any invariant.

1. Every node is either red or black.
2. All leaves are black.
3. A red node only has black children.
4. Every path from a node to any of its descendant leaves goes through the same number of black nodes. The amount of black nodes in any such path is called the *black-height* of the node.
5. The root node of the tree must be black.

The fifth property is not included by all authors [14], as it is almost trivial to fix any tree violating this property.

Assuming this fifth property is violated, the root node is red, with two black children (because of property 3). Simply changing the color of the root node to black, never violates any of the discussed properties, and as such is of little consequence (see Figure 2.2).

Together, these properties ensure that every tree is balanced, as the longest possible path, alternating black and red nodes, is never more than twice the length of the shortest possible path, one with black nodes only [16].

As we can see, the first two properties are quite simple, and cannot be violated with a proper data structure and constructor. The more interesting, as well as harder to verify properties are the third and fourth properties. We call these the *red invariant* and *black invariant* respectively.

Read-only operations, like an `isMember` function, do not violate either invariant. However, insertion or deletion often end up changing the tree in such a way that at least one of the invariants is violated. It is easy to color the entire tree red, to ensure the black invariant is satisfied, or to color the entire tree black, to ensure the red invariant is satisfied. However satisfying both invariants requires more fine tuning.

The main concept handled in this study is insertion. The standard insertion algorithm is an imperative version as described by Cormen et al. [5], and

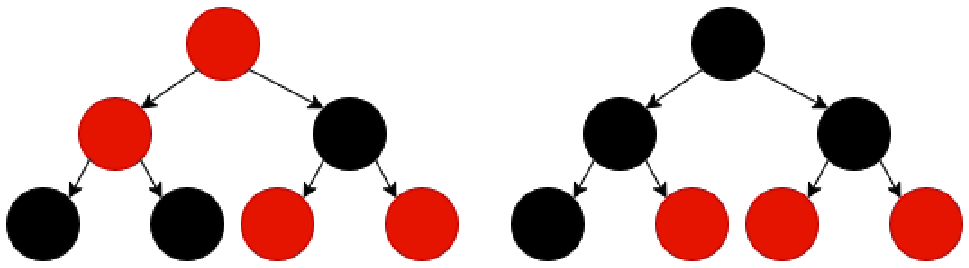

Figure 2.3: Examples of violations of the red and black invariants on the left and right trees respectively.

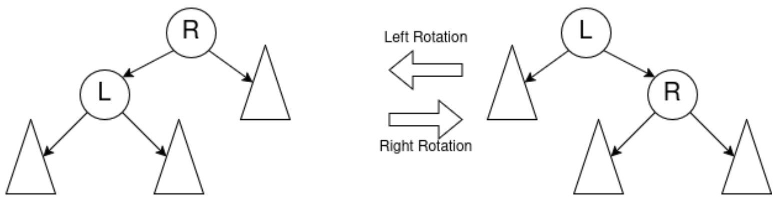

Figure 2.4: Example of left and right rotations.

follows the following structure:

- Pass through the tree to find the proper location for the initial insert, and insert the node in the proper spot. Since the tree is balanced, in a tree with n nodes this would have a time complexity of $\mathcal{O}(\log n)$.
- Color the new node red. The reason for coloring red is that it is generally easier to fix violations of the red invariant than it is to fix violations of the black invariant. Time complexity of $\mathcal{O}(1)$.
- Fixup. If the new node violates the red invariant, a fixup procedure is initiated. This procedure involves looping up through the tree starting from the newly inserted node. In each step of this loop, the algorithm performs some color checks, possible recolorings and possibly one or two rotations (see Figure 2.4). In terms of time complexity, this procedure could loop through the entire height of the tree, performing $\mathcal{O}(1)$ operations throughout the tree, and thus has time complexity of $\mathcal{O}(\log n)$.

Overall this insertion algorithm runs in $\mathcal{O}(\log n)$ time, precisely because the tree is balanced. In the best or average cases, the extra steps add some slight overhead compared to a naive binary search tree, but in terms of worst case performance ensuring a balanced tree leads to much better performance.

2.3 Implementing Red-black Trees in Haskell

When implementing red-black trees in a functional language, some challenges and opportunities show up, that encourage a slightly different implementation than the standard described by Cormen et al. [5]. While an imperative version can easily recolor a node, in a functional environment we work with immutable data structures, so even just recoloring a node leads to recreating the node. In this section we explain how to use this to our advantage to implement red-black trees and specifically insertion in a more elegant way, as described by Okasaki [16].

We start by describing the data structure used to store the trees, and making a constuctor for this structure:

```haskell
data Color = Red | Black
  deriving (Show, Eq)

data BST a = Leaf | Node Color (BST a) a (BST a)
  deriving (Show, Eq)
```

The data type is a simple binary search tree, with an extra field for the color, which is either red or black. We then store the left subtree, the value of the node itself, and the right subtree.

With the base of the red-black trees implemented, we continue with the actual insertion implementation:

```haskell
insert :: (Ord a) => a -> BST a -> BST a
insert x tree = makeBlack $ ins tree
  where
    ins Leaf = Node Red Leaf x Leaf
    ins (Node color left val right)
      | x < val = balance color (ins x left) val right
      | x > val = balance color left val (ins x right)
      | otherwise = Node color left val right -- x == val
    makeBlack (Node _ left val right) =
      Node Black left val right
    makeBlack Leaf = Leaf
```

Here, `insert` is the main function called. It starts by calling `ins`, which is a helper that recursively walks down the tree, until it finds the correct `Leaf` to replace with the new value. The function `ins` then walks back up the tree, having our balance function fix any violations.

The function `ins` only actually inserts a new node when it reaches the bottom of the tree, where it inserts a new red node. Since it replaces a black leaf, but adds a new red node with black leaves as children, the black-height of the tree does not change. Because of this the only violation possible is one where the newly inserted node has a red parent, violating the red invariant.

At the root node, if there is such a violation, this is fixed by coloring it black, which increases the black-height of the tree by one, but since we are at the root node, this happens for the entire tree, meaning we do not violate the black invariant (see Figure 2.2).

Since our insert function adds exactly one red node, there is at most one red violation in the tree. This is either at the root node, or in one of its subtrees. If it is in one of the subtrees, the red violation must have a black parent, since otherwise we would have two red violations. Because of this, matching any black node with a red violation in one of its children, together with fixing a red violation at the root, means we have all possible violations matched and taken care of.

Since we know how to fix a red violation at the root node, all that is left is figuring out how to fix the four cases of a black node with a red child, where said red child also has a red grandchild. As it just so happens, there is a very elegant way of restructuring and recoloring these four trees in such a way that we get rid of the red violation, while keeping the black height the same. The `balance` function has four cases, which each corresponds to one of the outer trees in Figure 2.3. These four cases each get restructured to the same resulting tree, which is shown in the middle tree from the figure.

```haskell
balance :: (Ord a) => Color -> BST a -> a -> BST a -> BST a
balance Black (Node Red (Node Red a x b) y c) z d =
  Node Red (Node Black a x b) y (Node Black c z d)
balance Black (Node Red a x (Node Red b y c)) z d =
  Node Red (Node Black a x b) y (Node Black c z d)
balance Black a x (Node Red (Node Red b y c) z d) =
  Node Red (Node Black a x b) y (Node Black c z d)
balance Black a x (Node Red b y (Node Red c z d)) =
  Node Red (Node Black a x b) y (Node Black c z d)
balance color left val right = Node color left val right
```

The way this balancing function works is the main difference in this approach compared to the approach described in Section 2.2. Like its imperative counterpart, this balancing function also runs in $O(1)$. The reason for this difference is that working with immutable data structures means there is no easy way to quickly recolor a node without generating an entire new tree. Okasaki makes full use of having to recreate the tree anyways, and goes for a more elegant solution, running in the same time complexity. The main drawback here is that this version has slightly more overhead, since it has

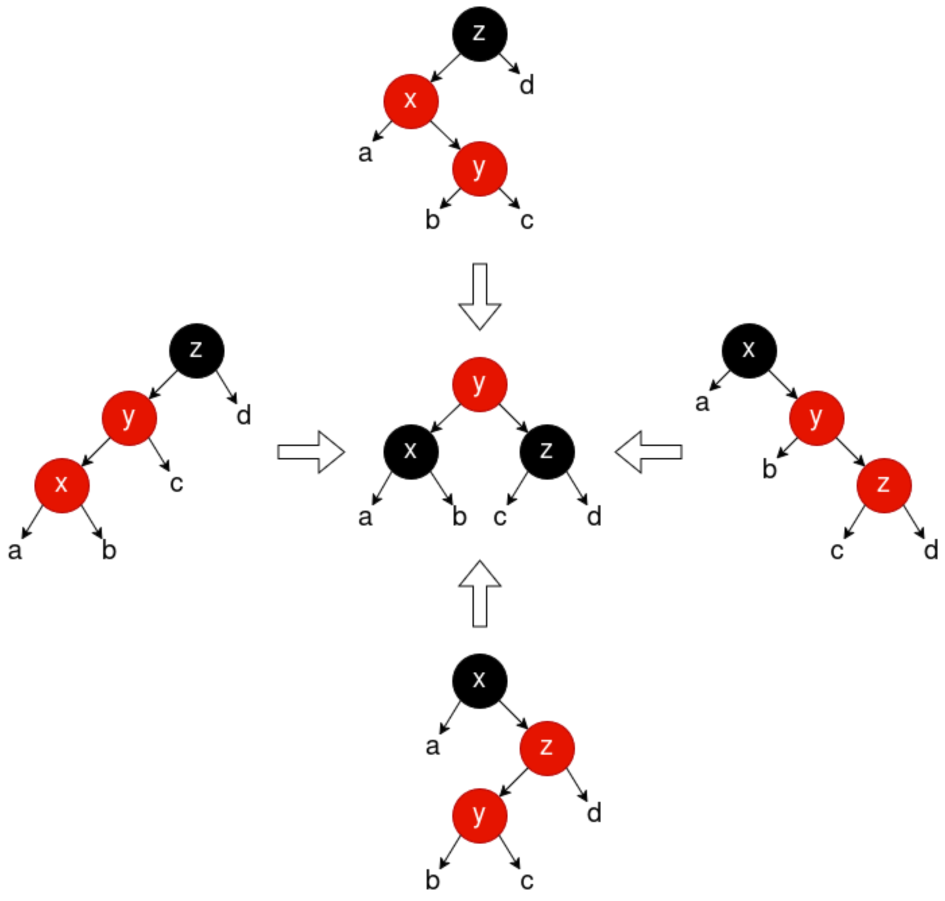

Figure 2.5: Remade version of Okasaki showing his balance function [16]. a, b, c and d are trees without any violations and with the same black-height.

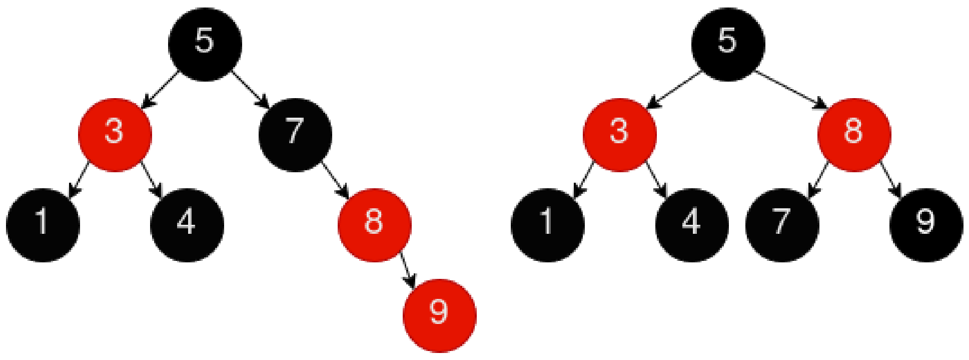

Figure 2.6: Simple example of insertion, using balance to fix a violation.

to create new nodes where in some cases, the imperative algorithm can get away with a couple pointer changes or changing a single bit in the color field. Working in a language with immutable data structures means this overhead exists no matter what, so the simpler alternative is usually preferred.

In conclusion, working in a functional environment, while perhaps not allowing for the most efficient implementation, does allow us to create a more elegant algorithm, while taking advantage of the pros of working in such a functional language. We have shown how this version can be implemented in Haskell, and explained how and why it works.

Chapter 3

Liquid Haskell

In this chapter we provide an introduction to Liquid Haskell [23], a Haskell plugin for verifying properties of Haskell programs beyond what is possible with standard type checking. We explain some of the weaknesses in Haskell's type system, motivate why this might lead to problems, and give the intuition of how Liquid Haskell attempts to solve these problems. We then dive deeper into Liquid Haskell's refinement types, explain how Haskell functions can be integrated into refinement specifications, before spending the final section of this chapter on a more theoretical overview of the process by which Liquid Haskell verifies its refinement types. Note that throughout this chapter we use Liquid Haskell's standard syntax, specifically the use of `{-@` and `@-}` to encapsulate any Liquid Haskell annotations.

Haskell's type system is able to prevent many errors, thanks to its static typing. However, there are many cases where the types usable in Haskell are not precise enough to exactly cover the allowed types for a function, but instead have to allow a slightly broader type. Take for example the type signature for a division function on integers:

```haskell
divInt :: Int -> Int -> Int
```

Of course, a division function should also account for the fact that dividing a number by 0 is an illegal operation. In Haskell, we could change the output of the function to a `Maybe Int`, and in the body of the function specify that dividing by 0 leads to a `Nothing` result.

While this approach does catch the illegal operation before performing it, there are some downsides. Encapsulating the return type in the `Maybe` monad forces any function interacting with it to also work with the monad in turn. This leads to less concise code, forcing developers to often use the convoluted do notation, or having to use complex transformers. Liquid Haskell proposes a different solution to this problem. Essentially, Liquid

Haskell's philosophy is based on the idea that the type `Integer` is not appropriate here for the divisor parameter. The actual type of this divisor is any integer that does not equal 0.

Liquid Haskell lets us annotate functions with predicates, allowing us to specify things exactly like the second argument of this function not being 0. These stronger, annotated types are known as *refinement types*.

Liquid Haskell translates these refinement types into a logical language. This translation process converts the refinement types into a set of constraints, which is fed into an SMT solver. The SMT solver then attempts to find a solution satisfying all constraints. If it cannot find such a solution, the program is rejected. If it can find a solution, it provides a formal guarantee that our program behaves exactly as specified in our refinement types.

3.1 Refinement Types

Liquid Haskell's core feature is its support for *refinement types*. In this section we explore the usage and syntax of these refinement types.

In essence, refinement types are Haskell types that have been extended with predicates, for example, by adding a predicate ensuring that a specific input parameter may not be 0:

```
{-@ divInt :: Int -> {x:Int | x /= 0} -> Int @-}
```

Looking at this example, we can recognize the Liquid Haskell annotation symbols encapsulating a function signature. Within these symbols we see what looks like a function signature, except for the second argument of the function. This parameter has been further refined using Liquid Haskell's syntax, and can be read as follows: "x is an integer, for which it holds that x does not equal 0".

In order to better understand how to annotate functions with refinement types, we walk through a practical example. Say we want to make a function that creates a very big square, with an area of over 100, using a `Rectangle` data type. We illustrate how to do this using refinement types to ensure that the output of this function is in fact a big square. We start by creating a `Rectangle` data structure, which we use as a base for our `makeBigSquare` function. We define `makeBigSquare` simply as another constructor for `Rectangle`, then use refinement types to ensure that it creates a big square:

```
data Rectangle = Rectangle Int Int
makeBigSquare :: Int -> Int -> Rectangle
makeBigSquare w h = Rectangle w h
```

We also create some functions to determine when a rectangle is a square, when it is big, and when it is a big square:

```
isSquare :: Rectangle -> Bool
isSquare (Rectangle w h) = w == h

isBig :: Rectangle -> Bool
isBig (Rectangle w h) = w * h > 100

isBigSquare :: Rectangle -> Bool
isBigSquare r = isSquare r && isBig r
```

We now use refinement types to put some requirements on the `makeBigSquare` function. We require that the width and height parameters are equal, and that they have a value bigger than 10:

```
{-@ makeBigSquare ::
    w:{Int | w > 10} ->
    h:{Int | h == w} ->
    v:Rectangle @-}
```

There are a few things important to note in this refinement type. First is that we can have multiple refinement types within our refined function signature. Second, we see that we can reference the width within the height refinement, since we bound the first parameter to the variable `w`. Important here is that we cannot reference the variable `h` in the refinement type for `w`, as the ordering here matters and thus `h` is unbound when parsing the first refinement type. Finally, we have not yet specified the output type, meaning at this point, Liquid Haskell simply restricts the input parameters for this function, and while we intuitively understand that this means the output is a big square, Liquid Haskell does not guarantee this.

In order to get Liquid Haskell to guarantee that the output type of `makeBigSquare` is actually a big square, we attempt to define a refinemed data type for big squares, and use that in `makeBigSquare`'s refinement type:

```
{-@ type BigSquare = {v: Rectangle | isBigSquare v} @-}
{-@ makeBigSquare ::
    w:{Int | w > 10} ->
    h:{Int | h == w} ->
    v:BigSquare @-}
```

Note that the type `BigSquare` exists completely outside of Haskell's knowledge, and only exists within Liquid Haskell. This means we can use this type in refinement types, but not outside of them.

While this is the correct approach, Liquid Haskell cannot verify that the output is a `BigSquare` at the moment, as we are making use of `isBigSquare`, and by extension, `isSquare` and `isBig`. Liquid Haskell does not know anything about these functions, and cannot reason about them. Without explicitly telling Liquid Haskell how to reason about user-defined functions, it is limited to its internal logical language, in particular the logic known as the Quantifier Free logic of Equality, Uninterpreted functions and Linear Integer Artihmetic, or QF-EUFLIA [23, 15] (see Section 3.3.1). There are some ways to give more information about user-defined functions to Liquid Haskell, to allow it to reason about these functions enough to finish many refinement types.

3.2 Lifting Functions to Refinement Logic

When writing refinement types, we are limited by the predicates that can be parsed by Liquid Haskell to feed to the SMT solver. Sometimes, we require more complex information in our refinement types in order to properly specify them. This section explains the two of the three main methods used to integrate Haskell functions into Liquid Haskell, allowing us to use custom functions in our refinement types. The third method, reflection [26], is deeply tied with equational reasoning in Liquid Haskell and falls outside the scope of this thesis.

3.2.1 Inlines

The simplest and most restricted method of letting Liquid Haskell reason about custom functions. Inlines are aliases, which simply get expanded at compile time, and are therefore limited to the same domain which restricts refinement types. They are allowed to call other inline functions, but may not recursively call themselves. Recursion is not allowed since these recursive calls would lead to infinitely expanding the function at compile time.

For example, we can inline `isBigSquare`, assuming that `isSquare` and `isBig` are also lifted into Liquid Haskell's environment in some way:

```haskell
{-@ inline isBigSquare @-}
isBigSquare :: Rectangle -> Bool
isBigSquare r = isSquare r && isBig r
```

3.2.2 Measures

Measures make use of the structural properties of algebraic data types to give the SMT a complete understanding of the properties of the data type, by converting the function to refinement types corresponding to each of the data type's constructors. Measures are highly restricted, but it is precisely because of these restrictions that such a conversion can work. Measures are meant as a way to measure one specific property of a data type instance. For example, a typical `length` function operating on a list, would return the amount of elements in the target list. This length can be considered a property that any list possesses, and is thus intuitively a good function for a measure.

Like inlines, there are some pretty strict requirements for a function for it to be considered a possible measure [24]:

1. Measures can only take one input parameter, which must be an algebraic data type.
2. Measures may, in their bodies, only call functions that exist within the domain of Liquid Haskell, that is, either primitive functions or functions that have been integrated into the domain of Liquid Haskell through any of the methods described in Section 3.2.
3. Measures must be defined by a single equation for each constructor.

The reason for the first constraint is that a measure is meant to be a property of an algebraic data structure. Giving a second argument means possibly returning different results based on this second argument, which goes against the idea of a function representing an inherent property. For example, the length of a list should not be dependent on any sort of secondary argument, but only require the list itself.

The reason for the second constraint is to ensure that no functions outside the domain of Liquid Haskell are used in the measure, ensuring that Liquid Haskell can still properly reason about the measure.

The third constraint is the often the most complex of the three to fully grasp. Essentially, this constraint means that each constructor must be handled using exactly one equation. For example, the constructors for lists in Haskell are `[]` and `(:)`. In order to write a measure we need to pattern match each of these constructors exactly once, and ensure that each pattern match has exactly one equation. This means we should avoid guard clauses, case expressions and if else statements, as these can lead to branching paths. We can use measures to lift both `isSquare` and `isBig` into Liquid Haskell's environment. We also show some illegal measures, as well as a measure indicating the use of recursion within measures:

```haskell
-- Proper measures
{-@ measure isSquare @-}
isSquare :: Rectangle -> Bool
isSquare (Rectangle w h) = w == h

{-@ measure isBig @-}
isBig :: Rectangle -> Bool
isBig (Rectangle w h) = w * h > 100

{-@ measure len @-}
len :: [a] -> Int
len [] = 0
len (x:xs) = 1 + len xs

-- Breaks constraint 1. by having two arguments
{-@ measure isMember @-}
isMember :: (Eq a) => a -> [a] -> Bool
isMember x [] = False
isMember x (y:ys) = x == y || isMember x ys

-- Breaks constraint 2. by using (++)
{-@ measure reverse @-}
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

-- Breaks constraint 3. by having two equations for (:)
{-@ measure containsZero @-}
containsZero :: [Int] -> Bool
containsZero [] = False
containsZero (0:_) = True
containsZero (_:xs) = containsZero xs
```

The function `isMember` has two arguments, and can never be a measure. The only way to make a similar function as a measure would be with a constant value to search for, to remove the need for the second argument. The `reverse` function cannot be a measure due to the use of `(++)`. There actually is a way to do this, by reflecting [26] the `(++)` function, allowing us to use it in a measure.

The reason for these strict constraints on measures is that they allow Liquid Haskell to interpret each measure by generating refinement types for the corresponding data constructors. For example, the `isBig` and `isSquare` measures generates the following refinement type:

```
{-
Rectangle :: w:Int -> h:Int ->
  {v:Rectangle |
    isSquare v == (w == h) &&
    isBig v == (w * h > 100) }
-}
```

Using this method of translating measures to refinement types, Liquid Haskell can reason about them and allows us to use our measures in other refinement types. Since there are multiple measures for an algebraic data type, Liquid Haskell conjoins their refinement types [23].

3.3 Verification Process

In order to understand how Liquid Haskell verifies the correctness of its refinement types, we need to understand how Liquid Haskell translates these refinement types to a language accepted by the SMT solver, and what this accepted language is. Besides that there are some challenges that arise from Haskell's lazy evaluation. We first explain this process of translating the refinement types, then discuss these challenges and how Liquid Haskell solves them.

3.3.1 Verification Conditions

Liquid Haskell's goal is to translate refinement types into some logical language that can be interpreted and solved by the SMT solver. In order to understand how to do this, we first need to understand what language we are translating to. Liquid Haskell uses the Quantifier Free logic of Equality, Uninterpreted Functions and Linear Integer Arithmetic, or QF-EUFLIA [25]. QF-EUFLIA is similar to first order logic [6]. The primary difference is that QF-EUFLIA lacks the universal and existential quantifiers. These quantifiers can make a logic undecidable, meaning avoiding these is important for allowing automatic verification. Besides that, QF-EUFLIA also incorporates equality, allowing for reasoning about the equality of objects. The equality operator in QF-EUFLIA is reflexive, symmetrical and transitive. QF-EUFLIA also incorporates uninterpreted functions, which are functions without any specific meaning but for which congruence is assumed. These uninterpreted functions are mainly used for reflection [26]. Lastly QF-EUFLIA incorporates linear integer arithmetic, allowing our refinement types to not just make use of booleans, but also use numerical values and mathematics.

Liquid Haskell uses the concept of *subtyping* to generate these logical formulas, known as *verification conditions.* We define subtypes as:

Definition 3.3.1 (Subtype)**.** We say that τ_1 is a subtype of τ_2, denoted by $\tau_1 \preceq \tau_2$, if for each $t : \tau_1$, it also holds that $t : \tau_2$

Intuitively this means some type τ_1 is a subtype of some type τ_2 if all possible values for τ_1 are also of type τ_2. We can translate this concept of subtypes into our verification conditions using implications.

We consider the `makeBigSquare` function:

```
{-@ makeBigSquare ::
    w:{Int | w > 10} ->
    h:{Int | h == w} ->
    v:BigSquare @-}
```

When Liquid Haskell analyzes `makeBigSquare`, it must prove that the return value is (a subtype) of the `BigSquare` type. This implies that the predicate `isBigSquare (Rectangle w h)` must hold under the given input restraints. Liquid Haskell starts by generating the following verification condition:

$$(\mathtt{w} > 10 \land \mathtt{h} = \mathtt{w}) \implies \mathtt{isBigSquare}(\mathtt{Rectangle\ w\ h})$$

To make sure the SMT solver can solve this formula, `isBigSquare`, and in turn `isBig` and `isSquare` must be understood within the QF-EUFLIA logic. From the inline for `isBigSquare` we know:

$$\mathtt{isBigSquare}(r) \iff (\mathtt{isSquare}(r) \land \mathtt{isBig}(r))$$

From the measure specifications for `isSquare` and `isBig` we know:

$$\mathtt{isSquare}(\mathtt{Rectangle\ w\ h}) \iff (\mathtt{w} = \mathtt{h})$$

$$\mathtt{isBig}(\mathtt{Rectangle\ w\ h}) \iff (\mathtt{w} * \mathtt{h} > 100)$$

By substituting these definitions into the verification condition for `makeBigSquare`, we end up with the following condition:

$$(\mathtt{w} > 10 \land \mathtt{h} = \mathtt{w}) \implies ((\mathtt{w} = \mathtt{h}) \land (\mathtt{w} * \mathtt{h} > 100))$$

This verification condition is then solvable by the SMT solver.

3.3.2 Dealing with Lazy Evaluation

The method for generating verification conditions of the previous section has one big flaw. Haskell's lazy evaluation means this method is unsound. Liquid Haskell addresses this by recognizing that in a lazy language, not all expressions are guaranteed to evaluate to a value. It then removes reliance on refinement predicates for potentially diverging functions within verification conditions.

In an eagerly evaluated language, arguments are fully evaluated before getting used. However, in Haskell's lazy evaluation, expressions are only evaluated when their values are actually needed. Take for example the following functions:

```
{-@ diverge :: Int -> {Int | False} @-}
diverge x = diverge x

badDiv x = let n = diverge 1 in x / 0
```

The refinement type for `diverge` signals that the function returns an integer such that `False`. This effectively means that the function does not terminate, and therefore not return a result. Using this `diverge` function in our `badDiv` function would mean a non-terminating function if we eagerly evaluated our arguments. However, due to Haskell's lazy evaluation, the call to `diverge 1` will never evaluate, as it is not needed anywhere. This means the function is dividing `x` by 0, which should be an illegal operation. Looking at the generated verification condition from these functions however, we see the following:

$$\text{false} \implies (v = 0) \implies (v \neq 0)$$

Since the first part of this entire implication is false, the entire implication holds true. This is contrary to what we expect, as dividing by 0 should get caught by Liquid Haskell. The problem here lies in the assumption that `diverge` will evaluate. If it would evaluate, the function would never terminate and thus never try to divide by 0. The problem is that due to Haskell's lazy evaluation, some functions might not evaluate.

The way Liquid Haskell solves this is by introducing the concepts of *stratification* and *termination analysis*. Instead of assuming all expressions evaluate to values, Liquid Haskell's type system labels variables as either "potentially diverging" or "guaranteed to terminate". Any potentially diverging function gets designated as a *Div type*. The key is that Liquid Haskell does not use knowledge from the refinement types of Div types, but instead simply evaluates them as true. This means our verification condition from earlier would instead look as follows:

$$\text{true} \implies (v = 0) \implies (v \neq 0)$$

This condition would indeed get rejected by the SMT solver.

The way Liquid Haskell assigns these Div types, is through termination analysis. Liquid Haskell attempts to prove that a function eventually reaches a base case and produces a value, even in recursive functions. If it manages to prove this termination, then the full refinement type can be used. If Liquid Haskell cannot prove this termination, the function gets assigned a Div type, and is handled accordingly.

Chapter 4

Verifying Red-black Trees

In this chapter we discuss how to verify the preservation of the red and black invariants for the implementation discussed in Chapter 2. We start by showing how to alter the described implementation to improve the verification. We then show how to define the red and black invariants, and how to use them in combination with Liquid Haskell to verify that, if these invariants hold before, then they also hold after running our functions.

4.1 Adaptations for Liquid Haskell

In order to simplify the needed refinement types, we make some changes to our implementation. We discuss these changes and their purpose in this section.

We start by making a smart constructor, which we use to make properly checked red-black trees. This can also be done with our current `BST` data type, however we have already seen that during the insertion process, we sometimes end up with trees that only partially satisfy the red invariant. We also want to allow these trees to exist, and if we were to fully refine the `BST` data type, we would need an entire new data type for this.

By making the `Node` constructor private, and only allowing public access to `newRBT`, we ensure all created trees are proper red-black trees.

```
newRBT :: Color -> BST a -> a -> BST a -> BST a
newRBT color left val right = Node color left val right
```

This smart constructor functions basically the same as the `Node` constructor, but we only give a refinement type to `newRBT`, allowing us to ensure `newRBT` creates proper red-black trees, while the `Node` constructor can create any binary search tree.

A small stylistic choice has also been to move `ins` and `makeBlack` outside of the where clause within `insert`. When adding refinement types, these functions look bigger, making the code potentially less readable. This does mean explicitly giving the value to be inserted to `ins` as well:

```haskell
ins :: (Ord a) => a -> BST a -> BST a
ins newVal Leaf = Node Red Leaf newVal Leaf
ins newVal (Node color left val right)
  | newVal < val = balance color (ins newVal left) val right
  | newVal > val = balance color left val (ins newVal right)
  | otherwise = Node color left val right -- newVal == val
```

The biggest and most important adaptation we make is splitting the `balance` function into a left and right part. When calling this function from `ins`, we already use a guard clause to decide on which side we are inserting. Knowing this, we have the knowledge that the other side remains unchanged. This means the verification requires different assumptions for the left and right balancing functions. While it may be possible to keep the knowledge of which side we are balancing by using a very complex refinement type for the `balance` function, this would become unnecessarily complex. This does also mean another slight refactor of the `ins` function.

```haskell
ins newVal (Node color left val right)
  | newVal < val = balanceL color (ins newVal left) val right
  | newVal > val = balanceR color left val (ins newVal right)

balanceL :: (Ord a) => Color -> BST a -> a -> BST a -> BST a
balanceL Black (Node Red (Node Red a x b) y c) z d =
  Node Red (Node Black a x b) y (Node Black c z d)
balanceL Black (Node Red a x (Node Red b y c)) z d =
  Node Red (Node Black a x b) y (Node Black c z d)
balanceL color left val right = Node color left val right
```

As we can see, `balanceL` checks only the two cases where the red violation is in the left subtree, as we already know the right subtree was unchanged. Similarly, `balanceR` is the symmetrical version, checking the other two cases, with the same default case at the end.

Those are all the changes we need to make. One of the core principles of Liquid Haskell is taking away the burden from the programmer, while still attempting to provide robust verifications. Two of the three changes we made are mostly stylistic choices, while the only one that actually makes the refinement significantly easier is splitting the balancing function into two parts.

4.2 Defining Invariants in Liquid Haskell

In order to verify red and black invariants, we first need to define these invariants. In this section we define the red and black invariants using Haskell, in such a way that they are available to Liquid Haskell. We also explain why they actually test for the invariants we want. We then combine these to create functions that check whether a tree can be considered a proper red-black tree, as well as the weaker version of red-black tree which we encounter during insertion.

4.2.1 Red Invariant

We start by defining the red invariant as a measure:

```haskell
{-@ measure redInvariant @-}
redInvariant :: BST a -> Bool
redInvariant Leaf = True
redInvariant (Node color left _ right) =
  isNotRedViolation color left right
    && redInvariant left
    && redInvariant right
```

What we want for this function to check is whether or not the entire tree satisfies the red invariant, that is, for all nodes in the tree, if a node is red, it can not have any red children.

This function can be split into two parts. The first part is the condition `isNotRedViolation`, which is a function that returns true if the current node does not violate the red invariant:

```haskell
{-@ inline isNotRedViolation @-}
isNotRedViolation :: Color -> BST a -> BST a -> Bool
isNotRedViolation color left right =
  if color == Red
    then getColor left /= Red && getColor right /= Red
    else True
```

While the last two parts of the conjunction recursively check whether the function itself also returns true for both the left and right subtree. This means `isNotRedViolation` should hold for the root node, as well as all nodes in the left and right subtrees in order to hold for the entire tree.

When we look at `isNotRedViolation`, we see that we can distinguish two cases depending on the color of the root node. First is the case where the color is black, in which case the implication vacuously holds. For the red invariant this makes sense, as a black node can never be a red node with a red child. Looking at the second case, where the current node is red, we proceed to look at the colors of the left and right subtrees. If either of them is red, this means we have found a red violation, meaning this part of the conjunction returns `False`, and thus the entire red invariant fails. This means that if the function returns `True`, we can conclude that the red invariant holds for every node in the tree.

4.2.2 Black Invariant

The black invariant refers to all paths from the root to a leaf. However, to simplify the verification, we discuss a simpler version where only one path is considered. We use the following function to calculate the black-height:

```
{-@ measure getBlackHeight @-}
getBlackHeight :: BST a -> Int
getBlackHeight Leaf = 1
getBlackHeight (Node color left _ _) =
  (if color == Black then 1 else 0) + getBlackHeight left
```

This function looks at the leftmost path, and returns the amount of black nodes in that path. We choose to count leaves as well, since they are considered black. This choice is one of preference, as not counting leaves lowers the black-height of every tree by exactly 1, since every path from a node to a leaf contains exactly one leaf by definition.

Next we use this function to define our black invariant:

```
{-@ measure blackInvariant @-}
blackInvariant :: BST a -> Bool
blackInvariant Leaf = True
blackInvariant (Node _ left _ right) =
  getBlackHeight left == getBlackHeight right
    && blackInvariant left
    && blackInvariant right
```

This function works in two parts, similar to the `redInvariant` function. The second part recursively calls itself on the left and right subtrees, which is how we traverse throughout the input tree to check the condition for every node. The condition we check is the first part of the function, which simply calculates the black-heights of the left and right subtrees, and returns false if they are not equal, causing an early termination to the function. If they are equal, we continue recursively calling the function.

We want to show that the function `blackInvariant` is equivalent to the usual black invariant. In the rest of this section, we prove that this is the case.

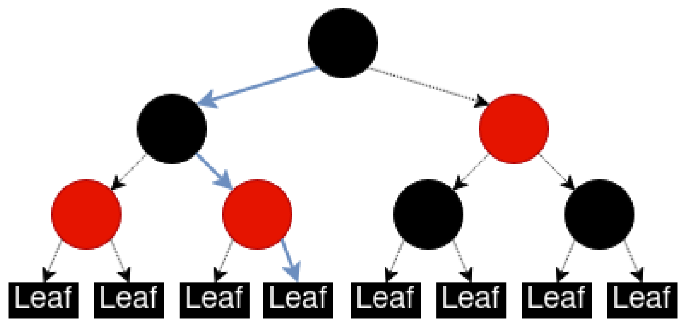

Figure 4.1: Example of the path [l,r,r] through a tree.

Real Black Invariant

In his initial paper on red-black trees, Sedgewick defines the black invariant as follows: For each internal node, all paths from it to external nodes contain the same number of black arcs [8]. We further translate this as:
Definition 4.2.1 (`real_black_invariant`)**.** For all nodes in a given tree, all simple paths from it to any leaf nodes contain the same number of black nodes

We further define a simple path as:
Definition 4.2.2 (Simple path)**.** A simple path from a node, is a queue of instructions, such that each instruction is either l (left) or r (right). Following those instructions from the starting node ends up exactly on a leaf node, with no more instructions left in the queue (see Figure: 4.1).

We also prove the following lemma:
Lemma 4.2.3. *If `real_black_invariant` holds for a given tree t, then the simple path [l,l,...] through t contains the same number of black nodes as any other simple path through t.*

This lemma holds, since by definition of `real_black_invariant`, all simple paths through a given tree contain the same number of black nodes.

Simple Black Invariant

Next we define `simple_black_invariant`, which we use to represent the black invariant as we implemented it.
Definition 4.2.4 (`simple_black_invariant`)**.** We say a tree `t` satisfies `simple_black_invariant` if either:

- The path [l,l,...] contains the same number of black nodes as the path [r,l,l,...], and `simple_black_invariant` holds for both children of the root node.
- t is a leaf node.

In this definition, we used the paths [l,l,...] and [r,l,l,...] to represent the condition `getBlackHeight left == getBlackHeight right`, since the `getBlackHeight` function works by calculating the amount of black nodes contained in the path [l,l,...].

We now use structural induction on `t` to prove the following theorem:
Theorem 4.2.5.

$$\mathit{simple\_black\_invariant}\ t \implies \mathit{real\_black\_invariant}\ t$$

We then also prove that:
Theorem 4.2.6.

$$\mathit{real\_black\_invariant}\ t \implies \mathit{simple\_black\_invariant}\ t$$

Finally, we combine Theorem 4.2.5 and Theorem 4.2.6 to prove:
Theorem 4.2.7.

$$\mathit{simple\_black\_invariant}\ t \iff \mathit{real\_black\_invariant}\ t$$

Proof of Theorem 4.2.5. **Base Case**

For our base case we take `t = Leaf`. From this it follows that `simple_black_invariant` holds by definition. For `real_black_invariant`, we note that there is only one simple path from a `Leaf` node to a `Leaf` node, being the path []. Since there exists only one path, it must contain the same amount of black nodes as itself, so `real_black_invariant` also holds. We now see that:

$$\texttt{simple\_black\_invariant Leaf} \implies \texttt{real\_black\_invariant Leaf}$$

Which holds because both `simple_black_invariant Leaf` and `real_black_invariant Leaf` hold.

Inductive Step

For our inductive step, we take `t = Node color left val right`, with our induction hypotheses:

$$\texttt{simple\_black\_invariant left} \implies \texttt{real\_black\_invariant left}$$
$$\texttt{simple\_black\_invariant right} \implies \texttt{real\_black\_invariant right}$$

Assuming `simple_black_invariant t` holds, we know that:

- The path [l,l,...] contains the same number of black nodes as [r,l,l,...]. We call this value $\texttt{bh}_t$.
- `simple_black_invariant left` and `simple_black_invariant right` hold

From our induction hypotheses, we now deduce that `real_black_invariant left` and `real_black_invariant right` also hold.

Let `p` be a path in `t`. If `p` starts with l, the amount of black nodes encountered in `p` equals that of [l,l,...], because `real_black_invariant left` holds. If `p` starts with r, the amount of black nodes encountered in `p` equals that of [r,l,l,...], because `real_black_invariant right` holds. This means that any path starting either with r or l, contains $\texttt{bh}_t$ black nodes, thus proving `real_black_invariant t` holds in these cases. Since there are no paths not starting with either r or l, this proves that:

$$\texttt{simple\_black\_invariant t} \implies \texttt{real\_black\_invariant t} \qquad \square$$

Next we prove Theorem 4.2.6:

Proof of Theorem 4.2.6. Assuming `real_black_invariant t` holds, we know that both [l,l,...] and [r,l,...] are simple paths to a leaf, and both encounter the same number of black nodes. We also know this holds for any node in `t`, including `left` and `right`. This is the definition of `simple_black_invariant`, meaning we know that `simple_black_invariant` also holds for `t`. □

We have proved both Theorem 4.2.5 and Theorem 4.2.6, thus proving the full logical equivalence of `simple_black_invariant` and `real_black_invariant` as described in Theorem 4.2.7.

4.2.3 Refined data types

We want to combine our invariants together in such a way that we can say something about whether or not any given tree is a proper red-black tree. In this case this simply means a tree that satisfies both the red invariant, and the black invariant. We then also create a type alias in Liquid Haskell to make our refinement types a bit cleaner when verifying our other functions:

```haskell
{-@ type RBT a = {v:BST a | isRBT v} @-}
{-@ inline isRBT @-}
isRBT :: BST a -> Bool
isRBT t = redInvariant t && blackInvariant t
```

However, we also saw that during insertion, we sometimes end up with a tree that temporarily has a red violation at the root node (see Section 2.3). We also define these trees and make a type alias for them, since we run into such trees at multiple points in the verification. We call these *weak red-black trees*.

First we need to properly define what makes a tree a weak red-black tree. A weak red-black tree is a red-black tree with possibly one red violation at the root node. That is, it is a red-black tree, satisfying the red and black invariants, except it is allowed to have a red root node, with one red child. If we make the root node of a weak red-black tree black, we get a red-black tree, since this removes any possible red violation at the root without violating the black invariant.

In other words, a weak red-black tree fully satisfies the black invariant, but does not fully satisfy the red invariant. It does however, satisfy the red invariant for the left and right subtrees, as the only violation is allowed to exist at the root node. In code:

```
{-@ type weakRBT a = {v:BST a | isWeak v} @-}
{-@ inline isWeak @-}
isWeak :: BST a -> Bool
isWeak t =
  blackInvariant t
    && redInvariant (getLeft t)
    && redInvariant (getRight t)
```

We have now defined what a full red-black tree should look like, as well as a weak version of a red-black tree, using the black and red invariants. These type aliases and invariant measures allow us to reason about the input and output types of our functions in order to write good refinement types for them.

4.3 Applying Refinement Types

In this section we describe the process of adding refinement types to our functions. We start each function by refining the output type, from which point we delve deeper into what is required to satisfy those refinements.

4.3.1 Refining the Constructor

We start by refining our smart constructor. Without any refinements, this function simply takes the parts of a possible red-black tree and creates a tree from them, with not a single guarantee that this tree actually is a red-black tree. Our goal here is to refine this function in such a way, that we can guarantee the output to be a proper red-black tree:

```
{-@ newRBT :: Color -> BST a -> a -> BST a -> RBT a @-}
```

Attempting this, we get the following error:

```
Liquid Type Mismatch
  .
  The inferred type
    VV : {v : (RedBlackTree.BST a) | ...}
  .
    is not a subtype of the required type
      VV : {VV##5993 : (RedBlackTree.BST a) |
        RedBlackTree.redInvariant VV##5993}
  .
    in the context
      left##a1IEI : (RedBlackTree.BST a)
      right##a1IEK : (RedBlackTree.BST a)
      color##a1IEH : RedBlackTree.Color
  Constraint id 342
```

Note that the actual error has a lot more information, and is a lot bigger. The added information is present in the inferred type, where the errors displays the refinement type as far as it manages to deduce it. Since we have given basically no information in the refinement type, other that what we expect the result to be, this inferred type is simply displaying the results of mesure calls, i.e. `getLeft v == left`.

We see that this error message is displayed in three parts, the inferred type, the required type and the context. In the inferred type, Liquid Haskell is stating what it is able to verify about the resulting type. In this case, it was only able to identify that the output tree `v` is a `BST a`, together with the results of the measures we defined on `BST`s. The second part is the required type. This is Liquid Haskell stating what it was not able to prove about the output, in this case, it was unable to prove that the red invariant holds for the resulting tree. Third we have the context, which provides some extra information on information Liquid Haskell was able to verify outside of the resulting tree `v`.

Since we want the output tree to satisfy the red and black invariants, both the left and right subtrees should do the same. This means that both input trees should already be red-black trees in order for the output tree to be one:

```
{-@ newRBT :: Color -> RBT a -> a -> RBT a -> RBT a @-}
```

This gives us a bit more information to work with, but not quite enough. Looking at the required type in the error message shows us that the solver currently cannot guarantee the red invariant holds for the tree. Since we do know the subtrees satisfy the red invariant, the problem is that currently, a red root with red in either of the two subtree nodes is allowed. To prevent this, we tell the solver that there is no case where both the color of the new tree and the color of one of the subtree is red, that is, at least one of them must not be red:

```
{-@
newRBT :: c:Color ->
          l:{RBT a | c /= Red || getColor l /= Red} -> a ->
          r:{RBT a | c /= Red || getColor r /= Red} -> RBT a
@-}
```

We can now see that the solver does figure out that the red invariant holds for the output tree, since the error no longer shows the red invariant in the required type. However there is still an error, because we have a similar problem with the black invariant. Currently, the left and right trees might have different black-heights, meaning the new tree would not satisfy the black invariant. Similar to the red invariant, we add another condition to our inputs to fix this:

```
{-@
newRBT :: c:Color
        -> {l:RBT a | c /= Red || getColor l /= Red}
        -> a
        -> {r:RBT a | (c /= Red || getColor r /= Red) &&
                      getBlackHeight r == getBlackHeight l}
        -> RBT a
@-}
```

This is enough to guarantee our postcondition. While this is a relatively simple example, it serves to illustrate the process of refining functions, by showing how to work from a normal function to a fully verified refinement type. Even though the function itself is an almost trivial translation of the `Node` constructor from the `BST` data type, we still need to put some thought into what we want to have as a result from this function, as well as how we refine the inputs in order to achieve this desired result.

4.3.2 Refining Insert

In this section we explain the process of refining our insert function and its helpers. Throughout this section, we do not handle `balanceR`, but only `balanceL`. As these two functions are very similar, the refinements are almost symmetrical and handling both would provide little to no added benefit. For the full refinement of `balanceR` see Section A.1.

Refining our insert function starts with refining the function itself, after which we transition into the helper functions. For `insert` itself, we of course want that the result of inserting a value into a red-black tree also returns a red-black tree. This does require the input to be a red-black tree as well, since our function does not fully recolor and rebalance the tree, but rather only fixes the violations caused by the insertion itself.

```
{-@ insert :: (Ord a) => a -> RBT a -> RBT a @-}
```

At this point, we see an error appear in `insert`. Since we have not specified the refinement types for `ins` and `makeBlack`, Liquid Haskell has no idea what their outputs are.

At this point we consider what exactly these functions do. First, we call `ins`, which inserts a red node at the bottom of the tree, then recursively calls balance to move any violation back up the tree, and finally it returns a tree with at most one red violation at the root node, and no black violations. Second, `makeBlack` colors the root node black.

We make use of the weak red-black trees here, which describe exactly the structure returned by `ins`, with at most one red violation at the root node, and no black violations. Since this type is used to describe exactly the return type of `ins`, we specify this in its refinement type. We then also specify the refinement type of `makeBlack` to take this result of `ins`, and remove the possible violation at the root node, turning it into a full red-black tree.

```
{-@ makeBlack :: WeakRBT a -> RBT a @-}
{-@ ins :: (Ord a) => a -> RBT a -> WeakRBT a @-}
```

Next we need to think about what our balancing function does. In `ins`, the call to `balanceL` is:

```
balanceL color (ins newVal left) val right
```

We can see that the return value for `ins` is the result of the call to `balanceL`, so the output type in the refinement for `balanceL` should be the same as the output type of `ins`. Furthermore we know that `isRBT color left val right == True` within `ins`, which leads to the conclusions that `right` is a red-black tree, and `ins newVal left` is a weak red-black tree, as this is the output type for `ins`.

Here we can see why splitting the balance function into a left and right part allows us to keep more information, as without this, we would not be aware which of these trees is a full red-black tree, and would have to classify both as possibly being weak. This does not make the verification impossible, but a lot more complex.

A first attempt for the refinement type for `balanceL` then looks as follows:

```
{-@ balanceL :: (Ord a) =>
  Color -> WeakRBT a -> a -> RBT a -> WeakRBT a @-}
```

Doing this, we see another error, stating that the required type for the output of `balanceL` is one where the black invariant should hold, but the inferred type does not reflect this. We know that there are two ways in which we alter the tree, either we add a red node, or we perform a rotation in our balancing function. Neither of these influence the black invariant, from which we can conclude that the black invariant always holds during our insertion. The reason Liquid Haskell does not pick up on this, is because we split our trees into parts using pattern matching, alter some of these parts, and then return a new, somewhat similar tree. Because of this we have to explicitly state what happens to the black-height.

```
{-@ ins :: (Ord a)
    => a
    -> t:RBT a
    -> v:{WeakRBT a |
          getBlackHeight v == getBlackHeight t} @-}
{-@ balanceL :: (Ord a)
    => color:Color
    -> left:WeakRBT a -> a
    -> right:{RBT a |
      getBlackHeight right == getBlackHeight left}
    -> v:{WeakRBT a |
          getBlackHeight v ==
            (if color == Black then 1 else 0) +
            getBlackHeight left} @-}
```

Note that in `balanceL`, the black-height does not change, but since we only have access to left and right, we have to manually add one to the count if the input tree parts have a black root node.

At this point, there is only one error left that Liquid Haskell is not able to fully verify with the information we have provided so far, in the final line of `balanceL`:

```
balanceL color left val right = Node color left val right
{-
Liquid Type Mismatch
  The inferred type
    ...
  is not a subtype of the required type
    {v : (RedBlackTree.BST a) |
    RedBlackTree.redInvariant (RedBlackTree.getLeft v)}
  ... -}
```

At this point, we are telling Liquid Haskell that `balanceL` returns a weak red-black tree. In the first two cases that get matched, Liquid Haskell can see from the inputs that `a, b, c` and `d` are all full red-black trees, with the same black-height. From this it concludes that the rotation performed in the first two cases does indeed return a weak red-black tree.

The problem lies in showing that the default case, where neither of the first two patterns are matched, also returns a weak red-black tree. Looking at the information we have about the red invariant for the input parameters, we know that our left subtree is a weak red-black tree, and our right subtree is a full red-black tree.

We distinguish 16 cases for such an input tree for `balanceL` (for all cases, see Figure 4.2). Within these cases, some already form weak red-black trees, and thus satisfy the output refinement type for `balanceL`. There are also two cases that get pattern matched before reaching the default case, so these two cases also result in a weak red-black tree. Ignoring these cases, we are left with four cases that currently do fall under the refinement type for the `balanceL` input parameters, but do not result in a weak red-black tree (see Figure 4.3).

The thing these four cases all have in common is that they all have multiple violations of the red invariant. If we look at the alterations that might be made to our tree during `ins` and `balanceL`, we see the following return calls:

- `ins` returning a new red node with no children.
- `ins` returning the result of `balanceL`. In this case, we change the left subtree for the result of a call to `ins`, which falls either in this or the previous case.
- `balanceL` returning a rotated tree, fixing a red violation, but possibly creating one with the parent node, by coloring the current node red.
- `balanceL` returning the input tree.

The only one here that might add another violation is `ins` returning a new red node with no children, since it adds a red node possibly under another red node. Since `ins` only calls itself recursively at most once in each call, and does not call itself again in the `Leaf` case, this `Leaf` case is called at most once. From this we can conclude that during insertion, a tree never has more than one red violation.

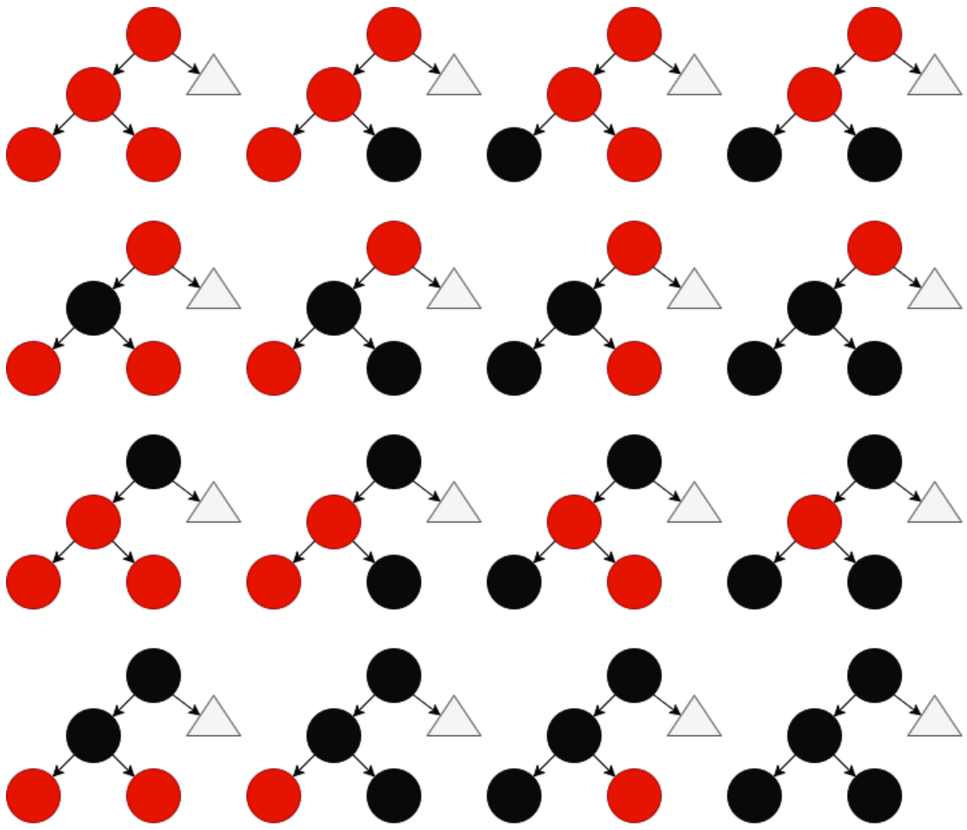
Figure 4.2: 16 possible cases for `balanceL`. Triangles represent full red-black trees.

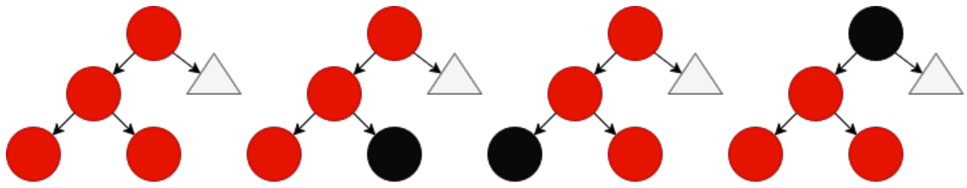
Figure 4.3: Four cases for `balanceL` that result in illegal trees.

We see here that the four trees that are still allowed as inputs for `balanceL` by our refinement type, are actually never given as input to `balanceL` when inserting a node. In order to provide this information to Liquid Haskell, we divide these trees into two categories, one with a red root node, and the other with a black root node.

For the first case, we assume the root node of the input to `balanceL` is red. Since we know there is at most one red violation, this means there can be no red violation in the left subtree, since this would mean the root node of the left subtree is also red, leading to a second red violation between the root node and the left child. This means that if the root color input to `balanceL` is red, we can conclude that the red invariant holds for the left subtree:

```
{-@
balanceL :: (Ord a)
    => color:Color
    -> left:{WeakRBT a | color == Red => redInvariant left}
    -> a
    -> right:{RBT a | getBlackHeight
         right == getBlackHeight left}
    -> v:{WeakRBT a | getBlackHeight v ==
                        (if color == Black then 1 else 0) +
                        getBlackHeight left}
@-}
```

For the second case, where the root node is black, we look at the resulting output of `balanceL`. We know that there is at most one red violation. If there are none, the function returns the input tree, which then also satisfies the red invariant. Otherwise, if there is a red violation with a black root node, we see that `balanceL` matches both cases to remove the violation, meaning the resulting tree also satisfies the red invariant. Since the result of `balanceL` is also the result of the recursive calls in `ins`, we add this refinement to both functions:

```
{-@
ins :: (Ord a) => a
        -> t:RBT a
        -> v:{WeakRBT a |
             (getColor t == Black => redInvariant v) &&
             getBlackHeight v == getBlackHeight t}
@-}
```

```
{-@ balanceL :: (Ord a)
    => color:Color
    -> left:{WeakRBT a | color == Red => redInvariant left}
    -> a
    -> right:{RBT a |
         getBlackHeight right == getBlackHeight left}
    -> v:{WeakRBT a |
           (color == Black => redInvariant v) &&
           getBlackHeight v ==
             (if color == Black then 1 else 0) +
             getBlackHeight left} @-}
```

Doing this, we run into one of the limitations of working with custom data types in Liquid Haskell. Since the data type `Color` only has two possible values, we think of it as a binary data type, where black is the same as not red. However, Liquid Haskell does not always have access to this information, so in some cases, there is a distinction that has to be made between black and not red (or vice versa). With our refinement types as they are, we see there is still an error about not being able to prove that the red invariant holds for a specific tree. However changing the lines we just added gives us:

```
{-@ ins :: (Ord a)
    => a
    -> t:RBT a
    -> v:{WeakRBT a | (getColor t /= Red => redInvariant v) &&
                      getBlackHeight v == getBlackHeight t} @-}
{-@ balanceL :: (Ord a)
    => color:Color
    -> left:{WeakRBT a | color == Red => redInvariant left}
    -> a
    -> right:{RBT a |
         getBlackHeight right == getBlackHeight left}
    -> v:{WeakRBT a |
           (color /= Red => redInvariant v) &&
           getBlackHeight v ==
             (if color == Black then 1 else 0) +
             getBlackHeight left} @-}
```

This does allow Liquid Haskell to fully verify the red and black invariants for our insertion implementation.

4.4 Conclusion

In this chapter, we systematically detailed the process of formally verifying the preservation of the red and black invariants for our Haskell implementation of red-black trees, leveraging the capabilities of Liquid Haskell.

We first detailed how to adapt the implementation, in order to make the verification more readable, as well as detailing how to split the balancing function to allow easier reasoning about our refinement types. We have explained how to implement functions that check both the red and black invariant, and shown why these functions do work as we expect them to. This is a crucial step for the verification, since these invariants form the basis for our refinement types. We then proceeded to show the iterative process of refining our functions, showing the challenges in this process, while not only explaining what refinements to add and why, but also how we derive these refinements.

Through this methodical process, we eventually verified that our insert function preserves the red and black invariants, by properly refining its helper functions and showing where we might temporarily violate one of these invariants.

One of the key challenges we met was the existence of a weak variant of the red-black tree, which only partially satisfied the structural invariants. Properly defining the structure and properties of this variant allowed us deeper insights into the way our implementation works, and how it guarantees the final result to guarantee the structural invariants.

Finally, we also thoroughly discussed and proved the way our black invariant works, and through it, proved why a simplistic, efficient implementation of a black-height function was possible, which uses the knowledge that the black invariant holds in order to circumvent having to check whether the black invariant holds every time we calculate the black-height.

Chapter 5

Related Work

In this chapter we contextualize the contributions of this thesis by examining existing related works.

One of the most closely related works is John Kastner's implementation of red-black trees in Liquid Haskell [12]. Kastner used Liquid Haskell to verify the same invariants, and used similar ideas for his refinement types. However there are a few key differences.

- Kastner uses *predicates* from Liquid Haskell, which is a concept that has been deprecated according the the Liquid Haskell documentation [22]. We replace most predicates with inlined functions and measures instead.
- Kastner also directly specifies the structural invariants into the data type. This allows him to make the recursive nature of these invariants less explicit, but instead simply assume they hold for every node. This does also lead to a second data type for the weak trees, after which Kastner needs to make some awkward conversions between these two types, since there is no shared overarching binary search tree type. This contrasts with the Liquid Haskell philosophy of prioritizing minimal modifications to the original code. We instead use a single binary search tree type `BST a`, which we then further refine for both the weak and strong red-black trees, allowing less changes to the original code.
- Lastly, Kastner leaves part of his implementation implicit, and does not discuss the process of adding his refinement types. Kastner's implementation is part of a lecture, so this lack of background information makes sense. We instead provided the full implementation, verification, process and include proofs that our invariants function as

expected.

Overall, while Kastner does show the possibility of using Liquid Haskell for the verification of the structural invariants, his work is not sufficient to give many insights by itself, as it was not made with that purpose in mind.

Further, red-black trees are often discussed in other literature. There have been multiple different verifications performed on red-black trees and algorithms for them in varying languages and environments, including but not limited to Rocq [3], Java [11], and Idris [17]. Many of these verifications rely on similar ideas, while some like Appel [3] come up with different approaches. The contributions made in this thesis compared to these other verifications lies in our analysis of Liquid Haskell, combined with the approach of showing the process of verifying our implementation, which is something many other studies gloss over, choosing to focus on the results instead.

Looking at related research on Liquid Haskell, we find relatively few case studies using Liquid Haskell to verify some complex data structure. Possible reasons for this lack of popularity for liquid type systems in general is studied by Gamboa et al. [7]. They found multiple reasons for this lack of popularity. For example, they found that many developers had trouble with what they called 'confusing verification features'. This essentially means many developers had trouble understanding when and how to lift functions as measures, or had trouble with differing scopes and sets of variables and operations between Haskell and Liquid Haskell. Another big problem they found was unhelpful error messages. Since the error messages thrown by Liquid Haskell rely on the reasoning performed by the SMT solver, these error messages are usually more complicated than what many developers are used to, leading to another barrier to overcome.

Chapter 6

Conclusions and Future Work

This thesis demonstrated the application of Liquid Haskell to formally verify the preservation of the red and black invariants in Okasaki's functional insertion algorithm for red-black trees. By defining precise invariants, as well as precise refinement types, we have captured the non-trivial changes that occur during insertion, and verified that the red and black invariants, while temporarily broken throughout the recursive insertion calls, do end up holding after finishing the full insertion algorithm. This work provides a concrete case study, illustrating both the correctness of this implementation, as well as a practical example of the usage of Liquid Haskell. It offers insights into translating informal invariants into proper functions. Specifically, it demonstrates an effective strategy for defining refinement types that capture nuanced, temporarily broken states within recursive algorithms like insertion, ensuring that the invariants are ultimately restored upon completion. This work further shows how to use Liquid Haskell for structural verification, providing a practical methodology for specifying and proving correctness for non-trivial data structures.

However, there are still possibilities for future research, some of which are detailed below:

Verification of additional operations: extending the current work to verify other operations, most notably deletion, is an interesting way to continue further down this path.

Performance analysis: For a relatively small verification as discussed in this thesis, the overhead added by Liquid Haskell at compile time is also relatively little. Further works could study how this overhead grows with the size of the program and verification, or even the complexity of either. Besides looking at overhead at compile time, there is also another avenue of investigating any possible overhead at runtime. While Liquid Haskell does very little (if anything) at runtime, there is an argument to be made about having to specify additional functions for verifications, or having changed functions precisely because the program is verified. These new or changed functions could, either positively or negatively, impact the runtime.

Other data structures: branching out sideways is also another option, examining other trees, or completely different data structures could yield more insight into their inner workings.

Comparison and integration with existing testing frameworks: Liquid Haskell performs a similar task to most testing frameworks, and future work could research where traditional testing breaks down but Liquid Haskell does not, or the other way around. Besides the differing use cases, future research could also investigate the overhead added at compile time for traditional testing frameworks compared to Liquid Haskell in the cases there either could theoretically be used effectively.

One of the aspects of Liquid Haskell that was left mostly unexplored in this thesis is its power for equational reasoning. This could be used to prove the equivalence between two different forms of the insertion algorithm, i.e. Okasaki's [16] compared to the mainstream imperative algorithm described by Cormen et al. [5]. Another avenue for research here is using this equational reasoning to prove other properties of algorithms, for example proving $x \in \texttt{insert}\ x\ t$.

Bibliography

[1] Georgy M. Adelson-Velsky and Evgenii M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962. In Russian. English translation: Soviet Mathematics - Doklady, vol. 3, pp. 1259–1263, 1962.

[2] Arne Andersson. Balanced Search Trees Made Simple. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides, editors, *Algorithms and Data Structures, Third Workshop, WADS '93, Montréal, Canada, August 11-13, 1993, Proceedings*, volume 709 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1993. `https://doi.org/10.1007/3-540-57155-8_236`.

[3] Andrew W Appel. Efficient Verified Red-Black Trees, 2011. `https://www.cs.princeton.edu/~appel/papers/redblack.pdf`.

[4] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: an experimental applicative language. Technical report, University of Edinburgh, 1980. `https://homepages.inf.ed.ac.uk/dts/pub/hope.pdf`.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 4th Edition*. MIT Press, 2009. `http://mitpress.mit.edu/books/introduction-algorithms`.

[6] José Ferreirós. The road to modern logic - An interpretation. *Bull. Symb. Log.*, 7(4):441–484, 2001. `https://doi.org/10.2307/2687794`.

[7] Catarina Gamboa, Abigail Reese, Alcides Fonseca, and Jonathan Aldrich. Usability Barriers for Liquid Types. In *Proceedings of the ACM on Programming Languages*, volume 9 of *PACMPL*, jun 2025. `https://doi.org/10.1145/3729327`.

[8] Leonidas J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21. IEEE Computer Society, 1978. `https://doi.org/10.1109/SFCS.1978.3`.

[9] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, May 1992. https://doi.org/10.1145/130697.130699.

[10] Hutton, Graham. *Programming in Haskell.* Cambridge University Press, 2nd edition, 2016.

[11] ir. H.M. Nguyen. Formal verification of a red-black tree data structure, March 2019. http://essay.utwente.nl/77569/.

[12] John Kastner. Liquid Structures: statically verifying data structure invariants with Liquid Haskell. https://www.cs.umd.edu/class/spring2019/cmsc631/files/Liquid_Structures.pdf.

[13] Miran Lipovaca. *Learn You a Haskell for Great Good! A Beginner's Guide.* No Starch Press, USA, 1st edition, 2011.

[14] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox.* Springer, 2008. https://doi.org/10.1007/978-3-540-77978-0.

[15] Greg Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981. https://people.eecs.berkeley.edu/~necula/Papers/nelson-thesis.pdf.

[16] Chris Okasaki. Red-Black Trees in a Functional Setting. *J. Funct. Program.*, 9(4):471–477, 1999. https://doi.org/10.1017/s0956796899003494.

[17] Chukwuyem J. Onyibe. Verification of Red Black Tree Algorithm in Idris, 2020. https://www.researchgate.net/profile/Chukwuyem-Onyibe/publication/379535199_Verification_of_Red_Black_Tree_Algorithm_in_Idris/links/660d85b1390c214cfd322fdd/Verification-of-Red-Black-Tree-Algorithm-in-Idris.pdf.

[18] Oracle. Class TreeMap. https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html, Java SE 8 Documentation, Accessed: May 24, 2025.

[19] The PostgreSQL Global Development Group. src/backend/lib/rbtree.c: Generic Red-Black Tree Implementation. https://github.com/postgres/postgres/blob/master/src/backend/lib/rbtree.cAccessed: May 24, 2025.

[20] Linus Torvalds and The Linux Foundation. lib/rbtree.c: Red-Black Tree Library. `https://github.com/torvalds/linux/blob/master/lib/rbtree.c`, Accessed: May 24, 2025.

[21] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. *Functional Programming Languages and Computer Architecture*, 201:1–16, 1985. `https://www.cs.kent.ac.uk/people/staff/dat/miranda/nancypaper.pdf`.

[22] Niki Vazou. Liquid Haskell Documentation: Spec Reference. `https://ucsd-progsys.github.io/liquidhaskell/specifications/`Accessed: May 28, 2025.

[23] Niki Vazou. *Liquid Haskell: Haskell as a Theorem Prover.* PhD thesis, University of California, San Diego, USA, 2016. `http://www.escholarship.org/uc/item/8dm057ws`.

[24] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. Theorem proving for all: equational reasoning in Liquid Haskell (functional pearl). In Nicolas Wu, editor, *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, pages 132–144. ACM, 2018. `https://doi.org/10.1145/3242744.3242756`.

[25] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for Haskell. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 269–282. ACM, 2014. `https://doi.org/10.1145/2628136.2628161`.

[26] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: complete verification with SMT. *Proc. ACM Program. Lang.*, 2(POPL):53:1–53:31, 2018. `https://doi.org/10.1145/3158141`.

Appendix A

Appendix

A.1 Source Code

```
{-# OPTIONS_GHC -Wno-unused-binds #-}
{-# OPTIONS_GHC -fplugin=LiquidHaskell #-}

module RedBlackTree
  ( Color (..),
    newRBT,
    isMember,
    insert,
    listToTree,
    getColor,
    getLeft,
    getRight,
    getBlackHeight,
    BST,
  )
where
```

```haskell
------------------------------------------------------------
------------------- DATA TYPES -----------------------------
------------------------------------------------------------

data Color = Red | Black
  deriving (Show, Eq)

-- private constructor
data BST a = Leaf | Node Color (BST a) a (BST a)
  deriving (Show, Eq)

-- public smart constructor
{-@ newRBT :: c:Color
      -> {l:RBT a | c /= Red || getColor l /= Red}
      -> a
      -> {r:RBT a | (c /= Red || getColor r /= Red) &&
                    getBlackHeight r == getBlackHeight l}
      -> RBT a @-}
newRBT :: Color -> BST a -> a -> BST a -> BST a
newRBT color left val right = Node color left val right

{-@ type RBT a = {v:BST a | isRBT v} @-}
{-@ type WeakRBT a = {v:BST a | isWeak v} @-}
```

```haskell
------------------------------------------------------------
------------------- PUBLIC GETTERS -------------------------
------------------------------------------------------------

{-@ measure getColor @-}
getColor :: BST a -> Color
getColor (Node color _ _ _) = color
getColor Leaf = Black

{-@ measure getLeft @-}
getLeft :: BST a -> BST a
getLeft Leaf = Leaf
getLeft (Node _ left _ _) = left

{-@ measure getRight @-}
getRight :: BST a -> BST a
getRight Leaf = Leaf
getRight (Node _ _ _ right) = right

{-@ measure getBlackHeight @-}
getBlackHeight :: BST a -> Int
getBlackHeight Leaf = 1
getBlackHeight (Node color left _ _) =
  (if color == Black then 1 else 0) + getBlackHeight left
```

```
------------------------------------------------------------
------------------- PUBLIC FUNCTIONS -----------------------
------------------------------------------------------------

{-@ listToTree :: (Ord a) => [a] -> RBT a @-}
listToTree :: (Ord a) => [a] -> BST a
listToTree [] = Leaf
listToTree (x : xs) = insert x (listToTree xs)

{-@ isMember :: (Ord a) => a -> RBT a -> Bool  @-}
isMember :: (Ord a) => a -> BST a -> Bool
isMember _ Leaf = False
isMember x (Node _ left val right)
  | x < val = isMember x left
  | x > val = isMember x right
  | otherwise = True -- x == val

{-@ insert :: (Ord a) => a -> RBT a -> RBT a @-}
insert :: (Ord a) => a -> BST a -> BST a
insert x tree = makeBlack $ ins x tree
```

```
------------------------------------------------------------
------------------- PRIVATE HELPERS ------------------------
------------------------------------------------------------

{-@ makeBlack :: WeakRBT a -> RBT a @-}
makeBlack :: BST a -> BST a
makeBlack (Node _ left val right) =
  Node Black left val right
makeBlack Leaf = Leaf

{-@ ins :: (Ord a) => a -> t:RBT a
      -> v:{WeakRBT a |
           (getColor t /= Red => redInvariant v) &&
            getBlackHeight v == getBlackHeight t} @-}
ins :: (Ord a) => a -> BST a -> BST a
ins newVal Leaf = Node Red Leaf newVal Leaf
ins newVal (Node color left val right)
  | newVal < val =
      balanceL color (ins newVal left) val right
  | newVal > val =
      balanceR color left val (ins newVal right)
  -- newVal == val
  | otherwise = Node color left val right

{-@ balanceL :: (Ord a)
      => color:Color
      -> left:{WeakRBT a |
           color == Red => redInvariant left}
      -> a
      -> right:{RBT a |
           getBlackHeight right == getBlackHeight left}
      -> v:{WeakRBT a |
           (color /= Red => redInvariant v) &&
            getBlackHeight v ==
              (if color == Black then 1 else 0) +
               getBlackHeight left} @-}
balanceL::(Ord a)=> Color -> BST a -> a -> BST a -> BST a
balanceL Black (Node Red (Node Red a x b) y c) z d =
  Node Red (Node Black a x b) y (Node Black c z d)
balanceL Black (Node Red a x (Node Red b y c)) z d =
  Node Red (Node Black a x b) y (Node Black c z d)
balanceL color left val right =
  Node color left val right
```

```haskell
{-@ balanceR :: (Ord a)
      => color:Color
      -> left:RBT a
      -> a
      -> right:{WeakRBT a |
           (color == Red => redInvariant right) &&
            getBlackHeight right == getBlackHeight left}
      -> v:{WeakRBT a |
           (color /= Red => redInvariant v) &&
            getBlackHeight v ==
              (if color == Black then 1 else 0) +
               getBlackHeight left}
@-}
balanceR::(Ord a)=> Color -> BST a -> a -> BST a -> BST a
balanceR Black a x (Node Red (Node Red b y c) z d) =
  Node Red (Node Black a x b) y (Node Black c z d)
balanceR Black a x (Node Red b y (Node Red c z d)) =
  Node Red (Node Black a x b) y (Node Black c z d)
balanceR color left val right = Node color left val right
```

```
------------------------------------------------------------
------------------- PRIVATE INVARIANTS ---------------------
------------------------------------------------------------

{-@ inline isNotRedViolation @-}
isNotRedViolation :: Color -> BST a -> BST a -> Bool
isNotRedViolation color left right =
  if color == Red
    then getColor left /= Red && getColor right /= Red
    else True

{-@ measure redInvariant @-}
redInvariant :: BST a -> Bool
redInvariant Leaf = True
redInvariant (Node color left _ right) =
  isNotRedViolation color left right
    && redInvariant left
    && redInvariant right

{-@ measure blackInvariant @-}
blackInvariant :: BST a -> Bool
blackInvariant Leaf = True
blackInvariant (Node _ left _ right) =
  getBlackHeight left == getBlackHeight right
    && blackInvariant left
    && blackInvariant right

{-@ inline isRBT @-}
isRBT :: BST a -> Bool
isRBT t = redInvariant t && blackInvariant t

{-@ inline isWeak @-}
isWeak :: BST a -> Bool
isWeak t =
  blackInvariant t
    && redInvariant (getLeft t)
    && redInvariant (getRight t)
```