Formalizing an Amortized Cost Analysis of Binomial Heaps and Fibonacci Heaps in Liquid Haskell

master thesis in computer science

by

Jamie Hochrainer, BSc.

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Master of Science

supervisor: Univ.-Prof. Dr. Georg Moser
Department of Computer Science

Innsbruck, April 29th 2024

universität innsbruck

Master's Thesis

Formalizing an Amortized Cost Analysis of Binomial Heaps and Fibonacci Heaps in Liquid Haskell

Jamie Hochrainer, BSc. (11806301)
jamie.hochrainer@student.uibk.ac.at

April 29th 2024

Supervisor: Univ.-Prof. Dr. Georg Moser

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit eigenständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Textpassagen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht.

An dieser Stelle sei insbesondere darauf hingewiesen, dass bei der Erstellung dieser Arbeit auf das textgenerative KI-System ChatGPT zurückgegriffen wurde. Dies geschah als Formulierungshilfe in Bezug auf die Überprüfung von der Grammatik und das Auffinden von passenden Synonymen.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Datum	Unterschrift

Abstract

In computer science, verifying algorithms and data structures is crucial for ensuring their accuracy and efficiency. Amortized cost analysis is a key aspect of this verification process, offering insights into the efficiency of data structure operations over multiple actions. Binomial and Fibonacci heaps are renowned for their efficiency in specific operations like insertion, deletion, and finding the minimum element. However, proving their properties and analyzing their performance can be challenging, especially when employing rigorous formal methods.

This thesis tackles the formalization of binomial and Fibonacci heaps using Liquid Haskell, a powerful tool for formal verification in Haskell. By harnessing Haskell's expressive type system and Liquid Haskell's refinement capabilities, our aim is to provide a comprehensive analysis of these heaps, encompassing correctness proofs and amortized cost analyses.

This involves encoding essential heap operations such as insertion, merging, and finding the minimum element in a manner that guarantees both correctness and efficiency. We delve into proving properties of these heaps using formal methods. This entails establishing and verifying various invariants, such as heap order and structure, to ensure that the heaps maintain their intended properties after each operation.

In addition to correctness proofs, we conduct an amortized cost analysis of the heap operations. By examining the amortized cost associated with each operation, we gain valuable understanding of the overall efficiency of the heaps. This analysis enables us to evaluate their suitability for various applications and provides insights into their performance over multiple operations.

Acknowledgments

First and foremost, I express my gratitude to my supervisor, Georg Moser, for his guidance, support, and encouragement throughout this master's thesis. His insightful feedback, constructive criticism, and constant motivation played a crucial role in shaping the outcomes of the thesis. I also extend my thanks to Niki Vazou for her valuable discussions, particularly during my research stay in Madrid. Her insights on Liquid Haskell enriched my understanding and enhanced the overall quality of the project. Furthermore, I appreciate the support and assistance of the Theoretical Computer Science and Computational Logic research group throughout my academic journey. Lastly, I am deeply grateful to my family and friends for their unwavering support and encouragement, which have served as a constant source of motivation throughout my studies.

Contents

1 Introduction **1**

2 Related Work **4**

3 Introduction to Binomial Heaps **6**
3.1 Binomial Tree . . . 6
3.2 Binomial Heap . . . 8
3.3 Functions of Binomial Heaps . . . 9

4 Introduction to Fibonacci Heaps **12**
4.1 Fibonacci Heap . . . 12
4.2 Functions of Fibonacci Heaps . . . 13
4.3 Relation to Fibonacci Numbers . . . 18
4.4 Comparison with Binomial Heaps . . . 19

5 Amortized Analysis **21**
5.1 Fundamentals . . . 21
5.2 Binomial Heap Bounds . . . 23
5.3 Fibonacci Heap Bounds . . . 28

6 Liquid Haskell **32**
6.1 Setup and Workflow . . . 32
6.2 Refinement Types . . . 34
6.3 Proving Extrinsic Theorems . . . 36
6.4 Formalization of Refinements . . . 38
6.4.1 Binomial Heap Refinements . . . 38
6.4.2 Fibonacci Heap Refinements . . . 41
6.5 Program Efficiency . . . 43

7 Formalization of Amortized Analysis **46**
7.1 Amortized Cost Analysis of Binomial Heaps . . . 47
7.1.1 Insertion . . . 48
7.1.2 Merge . . . 50
7.1.3 Deletion and FindMin . . . 51
7.2 Amortized Cost Analysis of Fibonacci Heaps . . . 52
7.2.1 Insertion . . . 53
7.2.2 Deletion . . . 54

8 Conclusion and Future Work **57**

Bibliography **58**

1 Introduction

Binomial heaps are recognized as significant data structures renowned for their efficiency in managing priority queues and executing associated operations. They offer streamlined implementations of operations like insertion, deletion, and extraction of extremal elements, making them ideal for applications with frequent priority queue manipulations. Even for large datasets, most operations exhibit logarithmic time complexities, and they have optimal space complexity. Their versatility extends to various algorithms and data structures, serving as essential components. Despite their relatively complex structure, binomial heaps remain easy to implement and comprehend in a functional programming language [3, 13, 15]. Their suitability for parallel processing further enhances their appeal, as binomial trees facilitate efficient splitting and merging operations, enabling parallel execution of priority queue operations and enhancing performance in multi-core environments [5].

In contrast, binomial heaps in imperative implementations, are often perceived as complex due to pointer management issues. Functional implementations of these data structures simplify pointer handling and more directly express high-level concepts, offering the advantage of persistence without additional effort [15].

Moreover, functional implementations leverage the recursive structure inherent in binomial heaps, allowing for elegant and concise code that reflects the recursive nature of binomial trees and the merge operation. This recursive approach facilitates intuitive implementation and reasoning about operations like insertion, deletion, and merging, compared to their imperative counterparts. By abstracting away from low-level pointer manipulations, functional implementations provide a clearer and more natural representation of the underlying data structure, enhancing readability and maintainability [13, 15].

Fibonacci heaps complement binomial heaps with their efficient implementation of certain priority queue operations, particularly decrease-key and merge operations. While binomial heaps excel in maintaining compact structures and supporting efficient insertion and deletion operations, Fibonacci heaps shine in scenarios where these priority queue operations are predominant. Their ability to perform decrease-key operations in constant time is crucial for algorithms like Dijkstra's shortest path algorithm and Prim's minimum spanning tree algorithm. Additionally, Fibonacci heaps have a more relaxed structure compared to binomial heaps, allowing for faster merging of heaps, which is advantageous for applications requiring frequent heap merges [3]. By leveraging the strengths of both data structures, algorithms can benefit from the efficiency of binomial heaps for

basic operations and the versatility of Fibonacci heaps for more complex priority queue manipulations, resulting in improved overall performance and algorithmic efficiency.

Our first goal is to formally verify correctness of invariants of these two data structures, involving mathematically proving that they adhere to certain specifications or properties under all possible circumstances. This process guarantees proper functionality devoid of errors or unforeseen outcomes. Since we implement binomial and Fibonacci heaps in the functional programming language Haskell, we use a plugin called Liquid Haskell to formally verify properties of Haskell programs. Liquid Haskell extends the existing type system with refinement types, allowing programmers to specify precise properties about code behavior. By employing automated theorem proving, Liquid Haskell provides assurances about correctness and reliability [22]. Programmers only need to provide the Haskell source file with the refinement type specification, and Liquid Haskell utilizes an SMT solver as part of its verification process, that returns SAFE if the code meets the specifications, or otherwise returns UNSAFE with the corresponding error message. We illustrate this process in Figure 1.1.

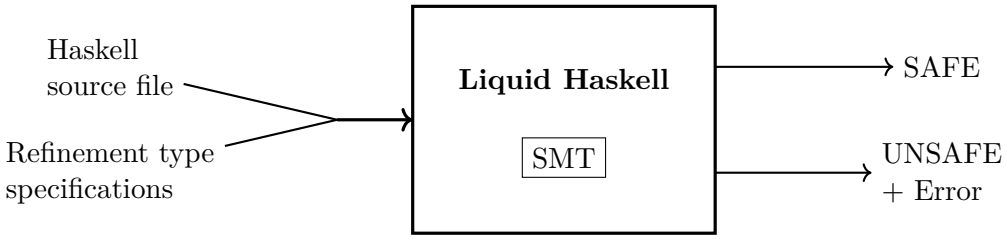

Figure 1.1: Liquid Haskell (simplified).

Liquid Haskell allows us to specify precise properties and invariants about our Haskell code. Those properties can be used intuitively in the refinement type as shown in Code Snippet 1.1. Here, we define the refinement type **Even** in Line 1. It says that an integer is even iff it is divisible by two without remainder. Then we can set a number, for example six, that is of this type. Without the Liquid Haskell refinement in Line 3, it would also be allowed to set an odd number as `evenNumber`, which is obviously not intended in this case. Hence, Liquid Haskell protects us from making a mistake. The advantage of setting a refinement type is that desired properties of the functions are visible directly in their signatures and automatically checked by Liquid Haskell.

```
{-@ type Even = {v:Int | v mod 2 == 0} @-}

{-@ evenNumber :: Even @-}
evenNumber :: Int
evenNumber = 6
```

Code Snippet 1.1: Refinement Example.

Since binomial and Fibonacci heaps have good amortized bounds, which are constant

or logarithmic, we also want to automatically prove them within our program. Hence the second goal of this thesis is then to perform an amortized cost analysis of the formalized data structures. For this purpose, we again use Liquid Haskell with an additional library that allows us to keep track of costs. This enables us to perform the analysis directly within the refinement type. Moreover, we formalized that for a binomial heap, the amortized cost is bound not only by the number of binomial trees it contains but also by the logarithm of the number of its nodes.

Overview. The subsequent chapters of this thesis are structured as follows: Chapter 2 discusses related work, while Chapters 3 and 4 respectively introduce the preliminaries of binomial and Fibonacci heaps including their functions. Chapter 5 outlines the fundamentals of amortized cost analysis, and we apply the analysis to our data structures on paper. In the next chapter, we focus on Liquid Haskell, explaining the workflow, refinement types, proofs, program efficiency, and formalizing binomial and Fibonacci heaps within the program. Finally, we provide a formalization of an amortized analysis of both data structures in Chapter 7 and conclude with potential future work in Chapter 8.

2 Related Work

Certainly, formalizing binomial and Fibonacci heaps is not a novel endeavor. Previous literature has explored these data structures or attempted to provide amortized cost analyses on paper [1, 3, 11, 15, 19, 20, 21]. Notably, Daniel Stüwe's master's thesis [20] delves into the verification of Fibonacci heaps in Imperative HOL, while Simon Griebel's master's thesis [9] builds upon Stüwe's work by focusing on the decrease-key operation of Fibonacci heaps. However, there are significant distinctions between these prior works and the present thesis, which we outline in this short chapter. There also exists prior work on refining invariants of data structures such as Red-Black trees [23] or analyzing (non-amortized) costs in Liquid Haskell [10]. However, in this thesis, we provide an amortized cost analysis for two data structures within Liquid Haskell while also refining required invariants. Additionally, we discovered an existing Liquid Haskell implementation of binomial heaps [11]. In this chapter, we will clarify the rationale behind providing another implementation of binomial heaps.

We start by highlighting the differences in the formalization of Fibonacci heaps. Stüwe's formalization places a strong emphasis on proving invariants and correctness properties of Fibonacci heaps. In contrast, my formalization mainly focuses on the amortized cost analysis of the data structure, aiming to set as few properties as needed. Stüwe employs Isabelle HOL, a formal proof assistant, for the formalization and verification of Fibonacci heaps, whereas I utilize Liquid Haskell, which can be applied to an existing implementation of the programming language Haskell. Unlike proving the amortized cost of the functions separately in lemmas, we can directly declare it in the refinements of Liquid Haskell. Refining the types of the function can be useful while programming to directly check if the implemented function indeed returns the desired output. One aspect missing in Stüwe's thesis, which he also mentions in his outlook chapter, is a formalization of the decrease-key and delete functions. This is addressed in Griebel's master's thesis. Griebel verifies the delete-key operation of Fibonacci heaps in Imperative HOL. However, Griebel employs a different approach than Stüwe, making it a separate formalization rather than a direct extension of Stüwe's work.

The original implementation of binomial heaps by Paul Palmer was developed using Liquid Haskell Version 0.8.6.2. However, this version of the program is incompatible with the newer version of Liquid Haskell (0.9.2.5) that we are using. When attempting to run Palmer's program, we encounter unbound symbol errors in the refinement. This is because in the newer version, refinement types cannot be set in the same manner. For example, the refinement types used in Palmer's version cannot access the outer type within the inner one, as shown below:

$ts : [\{t : \textbf{Tree a} \mid \texttt{length}\ ts > \texttt{rank}\ t\}]$.

To express the desired property within the new Liquid Haskell version, the refinements would have to be completely renewed. Instead of rewriting all those refinements in Palmer's version, there exists the {-@ LIQUID `"--bscope"` @-} flag which eliminates this refinement problem. However, even when solving the unbound symbol errors, other issues still persist. The online Liquid Haskell demo[1] uses version 0.8.6.0, where Palmer's program compiles and is verified as safe.

Moreover, Palmer's implementation relies on the Liquid.Bag library, which in turn depends on Data.Map. If we aim to prove that those imported operators have certain properties, we would need to reimplement them in order to state them in the refinement. Additionally, Palmer works with tree lists, necessitating the conversion of heaps to tree lists and lists to heaps in all heap functions. In contrast, we view a heap as a type that is exactly a list of binomial trees with some refinement. Due to these complexities, conducting an amortized cost analysis would be very complex in Palmer's version. Therefore, we decided to formalize binomial heaps in a simpler manner rather than extending the existing code.

[1] http://goto.ucsd.edu:8090/index.html#?demo=permalink%2F1675311619_18955.hs

3 Introduction to Binomial Heaps

Binomial heaps are composed of binomial trees and therefore a separate explanation of them is required. This chapter initiates the definition of binomial trees in Section 3.1 by highlighting key properties. Subsequently, attention is directed towards binomial heaps, encompassing their representation in Section 3.2 and operational functions in Section 3.3.

3.1 Binomial Tree

Definition 3.1. *Binomial trees*, as delineated by Okasaki [15], are recursively defined as follows:

- A binomial tree of rank 0 is a singleton node.
- A binomial tree of rank $r + 1$ is formed by linking two binomial trees of rank r, making one tree the leftmost child of the other.

An illustration of binomial trees is provided in Figure 3.1. We adopted Okasaki's definition of binomial trees [15] as the foundation for our functional implementation. Through this definition, several properties of binomial trees can be derived. However, for certain properties not explicitly defined by Okasaki [15] but outlined in Lemma 3.4 and Corollary 3.5, we refer to Cormen et al. [3]. We include the proofs of these properties in this thesis for clarity and completeness.

Definition 3.2. The *height* of a node p in a tree is the number of edges on the longest path from the node to a leaf. If p is a leaf, then the height of p is 0 [8].

Definition 3.3. The *depth* of a node p in a tree is the number of edges from the node to the tree's root node. If p is a root node, then the depth of p is 0 [8].

Note that the rank of the root node corresponds to the height of the tree and that height and depth of a tree are equal.

Lemma 3.4. *For a binomial tree B_k*

1. *there are 2^k nodes,*
2. *the rank of the tree is k,*
3. *there are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \ldots, k$, and*

4. *the root has rank k, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by $k-1, k-2, \ldots, 0$, child i is the root of a subtree B_i.*

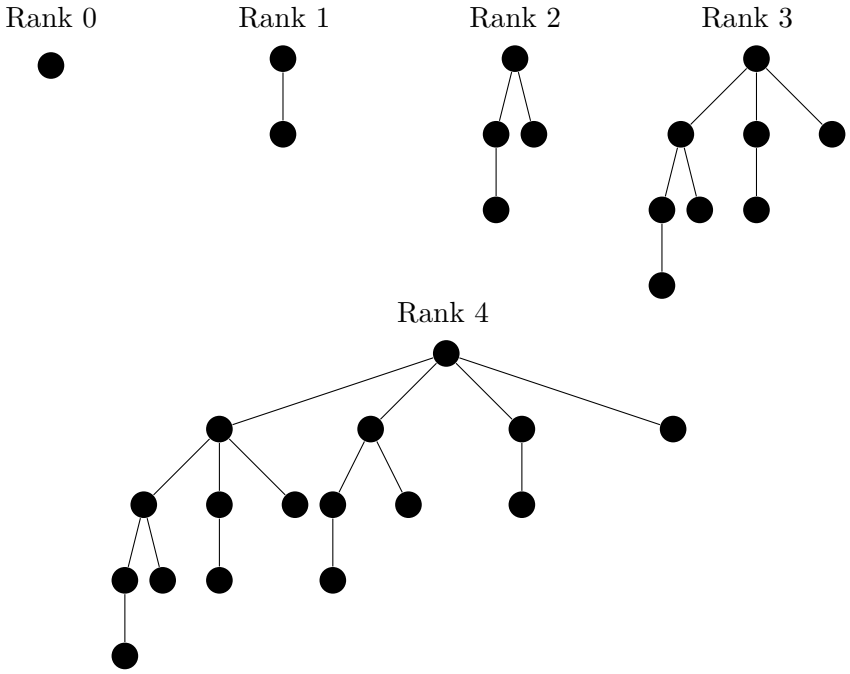

Figure 3.1: Binomial trees of ranks $0-4$.

Proof. The proof is by induction on k. To establish each property, we begin with the base case of the binomial tree B_0. Confirming the validity of each property for B_0 is straightforward. For the inductive step, we assume that the lemma holds for B_{k-1}.

1. The binomial tree B_k is formed by combining two copies of B_{k-1}, resulting in a total of $2^{k-1} + 2^{k-1} = 2^k$ nodes.

2. Due to the way in which the two instances of B_{k-1} are linked to create B_k, the rank of a node in B_k exceeds that of B_{k-1} by one. According to the inductive hypothesis, this increased rank equals $(k-1)+1 = k$.

3. Let $D(k,i)$ represent the number of nodes at depth i of the binomial tree B_k. Since B_k consists of two copies of B_{k-1} linked together, nodes at depth i in B_k compose of the nodes of B_{k-1} at depth i and the nodes at depth $i-1$ of the appended tree B_{k-1}. Thus,

$$\begin{aligned} D(k,i) &= D(k-1,i) + D(k-1,i-1) \\ &= \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}. \end{aligned}$$

4. The only node with a higher rank in B_k than in B_{k-1} is the root, which gains one more child compared to B_{k-1}. Since the root of B_{k-1} has rank $k-1$, the root of B_k has rank k. Furthermore, based on the inductive hypothesis, the children of the root of B_{k-1} are roots of $B_{k-2}, B_{k-3}, \ldots, B_0$, arranged from left to right. Consequently, when B_{k-1} is linked to B_{k-1}, the children of the root of the resulting tree are roots of $B_{k-1}, B_{k-2}, B_{k-3}, \ldots, B_0$.

□

Corollary 3.5. *The maximum rank of any node in an n-node binomial tree is* $\log_2(n)$.

Proof. This proof is deduced from properties 1 and 4 of Lemma 3.4. Since the root contains the maximum rank k and there are 2^k nodes in the tree, we conclude

$$k = \log_2(2^k) = \log_2(n).$$

□

The term "binomial tree" derives its name from property 3 of Lemma 3.4 because the expression $\binom{k}{i}$ is the binomial coefficient.

3.2 Binomial Heap

A binomial heap consists of heap-ordered binomial trees, each with a distinct rank, and the collection is represented as a list of trees arranged in ascending order based on their ranks [15]. Again we refer to the properties presented by Cormen et al. [3] that define a binomial heap as in Definition 3.6.

Definition 3.6. A *binomial heap* H is a set of binomial trees that satisfies the following properties:

- Each node has a key.
- Each binomial tree in H obeys the min-heap property: the key of a node is greater than or equal to the key of its parent. We say that each such a tree is *min-heap-ordered*.
- For any non-negative integer k, there is at most one binomial tree in H whose root has rank k.

The trees in a binomial heap of size n correspond to the binary representation of n. This is because each tree B in a binomial heap has a unique rank k and B_k has 2^k nodes [15].

Example 3.7. Consider a binomial heap with 22 nodes, whose binary representation is 10110. This heap would contain one tree each with 2 nodes, 4 nodes and 16 nodes.

The trees have rank 1, 2, and 4 corresponding to the position of the ones in the binary representation.

3.3 Functions of Binomial Heaps

The implementation of the basic functions of binomial heaps in this thesis is based on Okasaki's book [15]. These functions are `insert`, `merge`, `findMin` and `deleteMin`. In addition to explaining the functions, we already provide some Liquid Haskell code. This code looks similar to the known Haskell type system but extends it with refinements and is hence more expressive. Moreover, the Liquid Haskell code directly states intuitive properties of data structures and functions which helps to understand the code. The more rigorously we refine our data structure, the easier it becomes to prove further properties.

```
data BiTree a =
  Node
    { rank :: Nat
    , root :: a
    , subtrees :: {s:[BiTree a] | rank == length s &&
                  (length s == 0 || length s > 0 &&
                  rank == getRank (head s) + 1) && sorted root s
                  && ordRank s}
    }
```

Code Snippet 3.1: Refinement of the data structure **BiTree a**.

Before moving on to implementing a binomial heap, which consists of a list of binomial trees, we first need to formalize binomial trees. A node in a binomial tree stores its rank, its root element and a list of its children. While the rank is simply a natural number and the root is an element of type **a**, the refinement type of the children is more complex. This is because we not only want the subtrees to be a list of binomial trees but also to ensure a descending order of ranks. Moreover, we require that the list is exactly as long as the rank of the node and that every child has a rank smaller than the rank of its parent. Additionally, we want the list to be ordered by rank and the tree to be sorted. All those properties which are the intuitive definition of the data structure can be expressed in the refinement as illustrated in Code Snippet 3.1.

In this code snippet we specify that the rank is a natural number instead of just an integer. For the subtrees, the constraint "rank == **length** s" ensures that the length of the child list s is exactly as long as the rank of the node. To assert that the ranks of the children are smaller than that of their parent, we specify that the rank of the first element of the list is exactly one less than the node's rank. Because we need to access the first element of the list, we differentiate between the empty list and a list of length greater than zero. Additionally, `ordRank` is a separate function that ensures the ranks of

the subtrees are ordered in the list. More concisely, `ordRank` checks if the rank of each element is exactly one less than the rank of the previous element. The refinement `sorted` takes the root element and the subtree list as an input and verifies that the root element is smaller than every other element of the list.

As a next step, we define the type of a binomial heap in our code, as shown in Code Snippet 3.2. Here, a binomial heap is simply a list of binomial trees that fulfills a property `ordRankH`. This function is similar to the `ordRank` function, but instead of requiring a rank decrease of exactly one, it simply requires the ranks of the list to be increasing. We will see the connection of `ordRankH` and `ordRank` in Lemma 6.4 in a later chapter.

$$\textbf{type}\ \ \textbf{BiHeap a} = \{ts : [\textbf{BiTree a}] \ \mid \texttt{ordRankH}\ ts\}$$

Code Snippet 3.2: Refinement of the data structure **BiHeap a**.

After defining our heap, we proceed with the heap functions. The implementation of `insert` is presented in Code Snippet 3.3. To insert a new element x into the heap we initially create a singleton. A singleton is merely a node with rank zero, the root element x and a empty list of children. Then, we traverse the list and compare the ranks, which is accomplished in the helper function `insTree`. It's important to note that the rank cannot exceed that of the compared tree because we always insert a node with rank zero. If the rank is lower than that of the compared tree, we insert the node at this point in the tree list. If two trees have equal ranks, we link them together using the `link` function, and then continue inserting the linked tree. For linking, we simply attach the node with the larger root element under the node with the smaller element. In Section 5.2 we demonstrate that `insert` runs in constant amortized time.

```
link t1@(Node r x1 ts1) t2@(Node _ x2 ts2) =
        | x1 ≤ x2 = Node (r+1) x1 (t2 : ts1)
        | otherwise = Node (r+1) x2 (t1 : ts2)

insTree t [] = [t]
insTree t ts@(t' : ts')
        | rank t < rank t' = t : ts
        | otherwise = insTree (link t t') ts'

insert x ts = insTree (Node 0 x []) ts
```

Code Snippet 3.3: Insert function.

The function `merge` combines two heaps, and whenever two trees in the heap have the same rank, they are linked and inserted into the merged heap. This procedure is illustrated in Code Snippet 3.4.

To locate and remove the minimum element of the heap, we define a helper function

```
merge ts1 [] = ts1
merge [] ts2 = ts2
merge ts1@(t1:ts1') ts2@(t2:ts2')
        | rank t1 < rank t2 = t1 : merge ts1' ts2
        | rank t2 < rank t1 = t2 : merge ts1 ts2'
        | otherwise = insTree (link t1 t2) (merge ts1' ts2')
```

Code Snippet 3.4: Merge function.

called `removeMinTree`, which returns a tuple. The first element of the tuple is the tree whose root is the minimum element, and the second element contains the list of remaining trees in the heap without the extracted tree. When deleting the minimum element, we just call `removeMinTree` that extracts the tree with the minimum element in the root and subsequently merge its children with the remaining heap. Additionally, we utilize the Prelude Haskell function `reverse` in `deleteMin` to obtain a heap with an increasing order of ranks. We illustrate the functions in Code Snippet 3.5. In those functions we do not consider an empty heap, because in that case we cannot find or delete anything.

```
removeMinTree [t] = (t ,[])
removeMinTree (t:ts) =
        let (t', ts') = removeMinTree ts in
        if root t < root t' then (t, ts) else (t', t:ts')

findMin ts = let (t,_) = removeMinTree ts in root t

deleteMin ts = let (Node _ x ts1, ts2) = removeMinTree ts in
        merge (reverse ts1) ts2
```

Code Snippet 3.5: Find and delete minimum functions.

All explained functions of binomial heaps run at worst in $O(\log_2 n)$ time [15].

4 Introduction to Fibonacci Heaps

The Fibonacci heap data structure is designed to serve as both a mergeable heap and to support operations with constant amortized time. A mergeable heap is according to Cormen et al. [3] any data structure that supports the five operations: `makeHeap`, `insert`, `findMin`, `extractMin` and `merge`. Further, we will discuss an amortized analysis in Section 5. Operations such as `insert`, `extractMin`, and `decreaseKey` run efficiently in Fibonacci heaps, making them suitable for applications that involve frequent invocations of these operations.

4.1 Fibonacci Heap

Definition 4.1. A *Fibonacci heap* is a collection of rooted trees that are min-heap ordered. That is, each tree obeys the min-heap property: the key of a node is greater than or equal to the key of its parent [3].

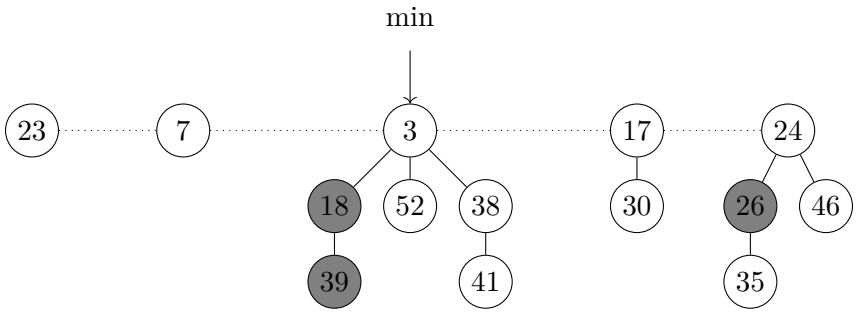

Figure 4.1: Fibonacci heap [3].

We avoid imposing explicit constraints on the number or structure of these trees, instead, constraints are implicit in the manipulation of the trees [6]. The construction of a Fibonacci heap and the operations performed on it will be explained in Section 4.2.

Figure 4.1 provides an example of a Fibonacci heap. In this instance, the heap comprises a total of fourteen nodes and consists of five min-heap-ordered trees. The dotted line delineates the root list of the heap and the minimum element of the heap, indicated by an arrow, is the number three. Nodes colored in grey are marked nodes, whereas all others are unmarked.

In a non-functional implementation, the data structure is often represented with each node containing pointers to its parent and a pointer to any of its children [1, 12, 6]. Additionally, the children are typically linked together in a circular, doubly linked list [3, 6]. However, Haskell does not support access via pointers, which is why we implemented Fibonacci heaps in an abstract manner. This implementation is outlined in Section 4.2.

4.2 Functions of Fibonacci Heaps

In this section, we provide a more concrete representation of Fibonacci heaps by demonstrating the implementation of the heap functions. We begin by defining the data structure in Code Snippet 4.2. To achieve this, we first define a Fibonacci tree in Code Snippet 4.1, where each node consists of a depth, a root element, a list of its subtrees, and an indication of whether it is marked or not. We refine the depth to be positive. Moreover, the function `equalDepth` indicates that all siblings have the same depth. Additionally, we require that the depth of a parent is exactly one smaller than the depth of its children. We will need this constraint for a termination proof, where depth serves as a termination metric. Termination and totality is necessary such that modeled proofs by induction on recursive functions are sound. Hence Liquid Haskell provides a powerful termination and totality checker and rejects any definition that it cannot prove to be terminating and total [25]. To access the first element of the list, we need to differentiate between an empty list and a list that contains at least one element.

```
data FibTree a =
  Node
      { depth :: Pos
      , root :: a
      , subtrees :: {s:[FibTree a] | length s == 0 ||
            (equalDepth s && getDepth (head s) == depth + 1)}
      , marked :: Bool
      }
```

Code Snippet 4.1: Refinement of the data structure **FibTree a**.

A Fibonacci heap, as defined in Code Snippet 4.2, is either empty or composed of two components: minTree and trees. The trees component is a list of trees contained in the heap, excluding minTree. Here, minTree refers to a tree that has the minimum element of the heap as its root, and we will henceforth denote it as the *minimum tree*. Maintaining a record of marked nodes in Fibonacci trees is crucial for the amortized cost analysis of the heap.

Once again, we have fundamental operations such as `insert`, `merge`, `findMin`, and `deleteMin`. As the name indicates, the `insert` function inserts an element x into a given heap. In contrast to the `insert` function of binomial heaps, here, we do not need to link

```
data FibHeap a = E | FH { minTree :: FibTree a
                        , trees :: [FibTree a]}
```

Code Snippet 4.2: Refinement of the data structure **FibHeap a**.

trees with the same degree because we allow multiple trees of the same degree. Therefore, our focus is on ensuring that the tree with the minimum element is separated from the rest of the tree list. This separation happens in the `merge` function, which compares the minimum elements of two heaps and selects the tree with the smaller one to be the minTree of the newly created Fibonacci heap. Subsequently, `insert` merges a singleton containing the element x with the current heap. In this scenario, a singleton refers to a Fibonacci heap that solely consists of one unmarked node with the element x. Code Snippet 4.3 illustrates the functions `singleton`, `merge` and `insert`.

```
singleton x = FH (Node 1 x [] False) []

merge E h = h
merge h1@(FH minTr1 ts1) h2@(FH minTr2 ts2)
      | root minTr1 < root minTr2 =
          FH minTr1 (minTr2 : ts2 ++ ts1)
      | otherwise =
          FH minTr2 (minTr1 : ts1 ++ ts2)

insert h x = merge h (singleton x)
```

Code Snippet 4.3: Singleton, merge and insert functions.

Figure 4.2 shows a Fibonacci heap where the element 21 is inserted in the Fibonacci heap depicted in Figure 4.1. The newly inserted node is highlighted in red. Since 21 is greater than the current minimum element, which is the number three, we do not alter the minimum tree and simply append the node to the root list. In general, the arrangement of the trees is not important in this data structure. However, due to the similarities to binomial heaps, we chose to implement it as a list.

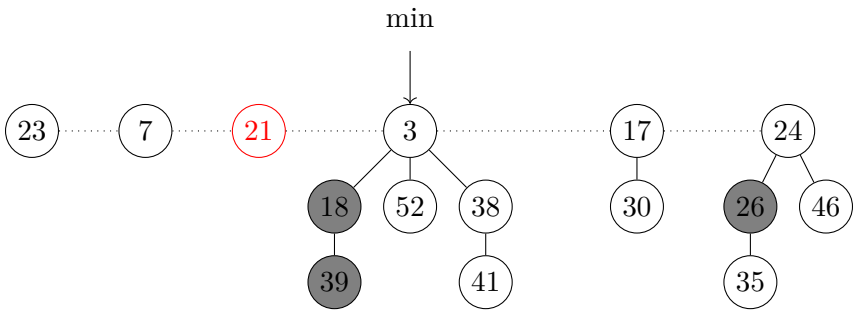

Figure 4.2: Inserting a node [3].

Finding the minimum element in the heap is straightforward due to the representation of the Fibonacci heap. The minimum element is simply the root element of the separated minimum tree.

```
findMin = root . minTree
```

Code Snippet 4.4: Finding the minimum element.

Deleting the minimum element of the heap is more complex. This function involves postponed work of consolidating trees in the root list. Initially, when we remove the minimum element, which is the root of minTree, we add all of its subtrees to the tree list. The subsequent step involves attempting to reduce the length of the tree list by linking trees of the same degree. Finally, we need to extract the new minimum tree that includes the minimum element. To execute these steps, we utilize the auxiliary functions `extractMin`, `consolidate`, `meld`, and `link` that are illustrated in Code Snippet 4.5. The `extractMin` function traverses the list of Fibonacci trees and extracts the one with the minimum element. The `consolidate` function invokes `meld` with each element, which links trees of the same degree, where the *degree* of a tree denotes number of the root node's children. For calculating the degree of a tree, we use the helper function `getDegree`. Additionally when linking, it unmarks the node where a tree is attached to.

We present a step-by-step example of deleting the minimum element of the heap in Figure 4.3. It is important to note that the order of linking may vary depending on the implementation. In this example, we use the Fibonacci heap obtained from the insertion process depicted earlier in Figure 4.2. We highlight changes in each step by marking affected nodes in red. Firstly, we remove the minimum node containing the element three, and add all its three children, that are nodes with elements 18, 52, and 38, to the root list. This step is illustrated in Figure 4.3b. Next, we link all trees until no trees in the heap have the same degree. Since the nodes with roots 7 and 23 both consist of only one node and hence have degree zero, we make the larger element a child of the smaller one. Consequently, in Figure 4.3c, the node with the element 23 becomes a child of the node with the element 7, resulting in a newly created tree with degree one. In Figure 4.3d, we

```
link t1@(Node d x1 ts1 _) t2@(Node _ x2 ts2 _)
        | x1 ≤ x2 = Node d x1 (t2:ts1) False
        | otherwise = Node d x2 (t1:ts2) False

meld [] t = [t]
meld (t1:ts1) t
        | getDegree t1 == getDegree t = meld ts1 (link t t1)
        | otherwise = t1 : meld ts1 t

consolidate [t] = [t]
consolidate (t:ts) = foldl meld [t] ts

extractMin [t] = (t, [])
extractMin (t:ts)
        | root t < root t' = (t, ts)
        | otherwise = (t', t:ts')
        where (t', ts') = extractMin ts

deleteMin (FH (Node _ x [] _) []) = E
deleteMin (FH minTr ts) =
        FH minTr1 ts1 where
        (minTr1, ts1) = extractMin $
                consolidate (subtrees minTr ++ ts)
```

Code Snippet 4.5: Link, meld, consolidate, extract minimum and delete minimum functions.

append the tree with the root 17 to the tree with the root element 7, as both trees are of the same degree. Similar steps are executed in Figures 4.3e, 4.3f, and 4.3g. Finally, in Figure 4.3g, there are only three trees in the heap with different degrees: three, two, and one. The minimum tree is the tree with the root element 7. In this example, deleting the minimum element results in halving the number of trees in the heap.

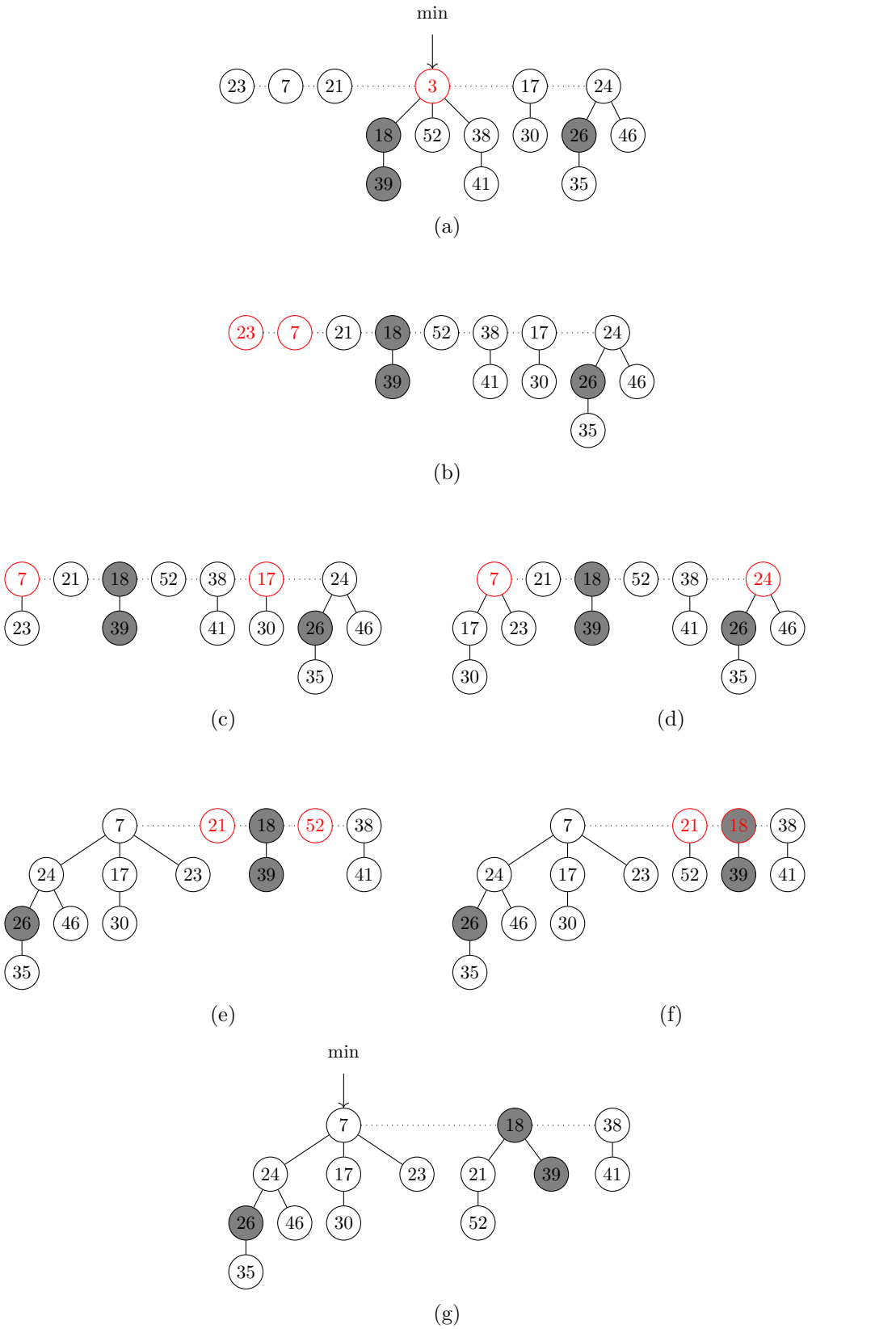

Figure 4.3: Deleting the minimum element [3].

4.3 Relation to Fibonacci Numbers

Fibonacci heaps derive their name from their connection to Fibonacci numbers, a relationship we explore in this section. The lemmas, corollaries, and proofs in this section are sourced and slightly adapted from the book "Introduction to Algorithms" of Cormen et al. [3].

Lemma 4.2. *Let x be any node in a Fibonacci heap, and suppose that* `getDegree` $x = k$. *Let $y_1, y_2, \ldots, y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then* `getDegree` $y_1 \geq 0$ *and* `getDegree` $y_i \geq i - 2$ *for* $i = 2, 3, \ldots k$.

Proof. Clearly, `getDegree` $y_1 \geq 0$.
For $i \geq 2$, we note that when y_i was linked to x, all of $y_1, y_2, \ldots y_{i-1}$ were children of x, and so we must have had `getDegree` $x \geq i - 1$. Because node y_i is linked to x (by `meld`) only if `getDegree` $x =$ `getDegree` y_i, we must have also had `getDegree` $y_i \geq i - 1$ at that time. Since then, node y_i has lost at most one child, since it would have been cut from x if it had lost two children. We conclude that `getDegree` $y_i \geq i - 2$. □

The Fibonacci numbers are recursively defined for $k = 0, 1, 2, \ldots$, as follows.

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

Another way to calculate Fibonacci numbers is described in the following lemma.

Lemma 4.3. *For all integers $k \geq 0$,*

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i.$$

Proof. Trivial by induction on k. □

Lemma 4.4. *Let x be any node in a Fibonacci heap, and let* `getDegree` $x = k$. *Then* `getSize` $x \geq F_{k+2}$, *where* `getSize` x *is the number of nodes of the tree where x is the root.*

Proof. Let s_k denote the minimum possible size of any node of degree k in any Fibonacci heap. Trivially, $s_0 = 1 = F_2$ and $s_1 = 2 = F_3$ and hence $s_k \geq F_{k+2}$ for $k = 0$ and $k = 1$. The number s_k is at most `getSize` x and, because adding children to a node cannot decrease the node's size, the value of s_k increases monotonically with k. Consider some node z, in any Fibonacci heap, such that `getDegree` $z = k$ and `getSize` $z = s_k$. Because $s_k \leq$ `getSize` x, we compute a lower bound on `size`(x) by computing a lower bound on s_k. As in Lemma 4.2, let $y_1, y_2, \ldots, y_k$ denote the children of z in the order in which they

were linked to z. To bound s_k, we add one to the count for z itself and one for the first child y_1 (for which `getSize` $y_1 \geq 1$). Additionally, we assume the induction hypothesis $s_i \geq F_{i+2}$ for $i = 0, 1, ..., k-1$, giving

$$\texttt{getSize}\ x \geq s_k \geq 2 + \sum_{i=2}^{k} s_{(\texttt{getDegree}\ y_i)} \overset{\text{Lem 4.2}}{\geq} 2 + \sum_{i=2}^{k} s_{i-2}$$
$$\overset{\text{IH}}{\geq} 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i \overset{\text{Lem 4.3}}{=} F_{k+2}.$$

□

4.4 Comparison with Binomial Heaps

Fibonacci heaps, like binomial heaps, consist of trees and support main operations such as inserting an element, merging heaps, finding, or deleting the minimum element in the heap. However, there are some differences between them. Fibonacci heaps have less stringent invariants than binomial heaps. They allow multiple trees of the same degree in the tree list, and the list does not need to be ordered by it.

Definition 4.5. *Time complexity analysis* falls within computational complexity theory, serving to characterize how much computational resources an algorithm utilizes. The big-O notation denotes the maximum growth rate of a function and is used to express asymptotic behavior [14].

Moreover, as we will see in Chapter 5, Fibonacci heaps offer better amortized costs than binomial heaps because they postpone expensive heap operations and execute them at a later point. In contrast to binomial heaps, Fibonacci heaps separate the tree with the minimum element from the tree list. Consequently, the function `findMin` in Fibonacci heaps has constant time complexity because it can directly access the tree with the minimum element as its root.

From a theoretical perspective, Fibonacci heaps prove advantageous particularly when the occurrences of `deleteMin` and `delete` operations are comparatively low in relation to other operations [3]. This scenario commonly arises across various applications. For instance, in certain greedy graph algorithms, such as Dijkstra's or Prim's algorithm. Despite Fibonacci heaps outperforming binomial heaps in terms of asymptotic time bounds which is summarized in Table 5.1, they are primarily of theoretical interest due to their complex constant factors and programming intricacies [3]. This is shown in experiments like network optimizations [7] or shortest path algorithms [2], where a Fibonacci heap implementation did not perform as efficient as expected. This might be because constant factors of implementations in Fibonacci heaps are greater than those for binomial heaps.

The representation of Fibonacci trees is similar to binomial trees, but each node in a Fibonacci heap also stores a Boolean indicating whether the node is marked or not. In Section 5.2, we will prove that the length of a binomial heap can be bounded by the logarithm of the number of nodes in the heap. However, this result does not hold for Fibonacci heaps. For example, a Fibonacci heap could consist only of singletons, leading to a linear length in terms of the number of nodes in the heap.

5 Amortized Analysis

Time complexity analysis of algorithms and data structures can be conducted in various ways, including worst-case, average-case, and amortized runtime analysis. Worst-case analysis determines the longest running time of an algorithm, providing an upper bound on the number of operations it takes to complete on any input. While worst-case analysis is essential for understanding an algorithm's behavior in extreme scenarios, it may not accurately represent typical real-world situations and can lead to overly pessimistic estimations. Average-case runtime analysis calculates the expected number of operations an algorithm takes to complete over all possible inputs, considering a specific input distribution. However, this analysis requires accurate modeling of input distributions, which can be complex and may not reflect real-world scenarios [21].

Amortized runtime analysis, on the other hand, does not involve probabilistic considerations. It provides the average time taken per operation in a sequence of operations on a data structure or algorithm. This analysis is particularly useful for data structures supporting numerous operations, focusing on operation sequences rather than individual worst-case scenarios. Section 5.1 provides an overview of the fundamentals of amortized analysis, drawing on descriptions and definitions from Cormen et al. [3]. Here, we focus on the potential method of amortized analysis. Section 5.2 conducts a concrete amortized cost analysis on binomial heaps, while Section 5.3 performs a similar analysis on Fibonacci heaps.

5.1 Fundamentals

Amortized analysis can be performed using different techniques, including aggregate analysis, the accounting method, and the potential method [17]. The aggregate analysis computes the average cost of a sequence of operations by distributing the total cost uniformly across all operations, providing a comprehensive evaluation of the algorithm's overall efficiency. In the accounting method, each operation is assigned a credit, and the credits are used to compensate for the actual costs, ensuring that the total accumulated credits remain non-negative. This approach provides a formal mechanism for analyzing the average performance of a sequence of operations. The potential method involves allocating a potential function to the data structure being analyzed, representing the stored "potential energy" that can be used to offset future operation costs. By comparing the potential before and after a sequence of operations, one can determine the amortized cost of operations, providing insights into the overall efficiency of the algorithm [17]. This

thesis focuses exclusively on the potential function method for amortized cost analysis, and we provide a detailed explanation of this method in this section.

Definition 5.1. Let D_i be the data structure that results after applying the i-th operation to data structure D_{i-1} for $i = 1, 2, \ldots, n$ and D_0 be the initial data structure. A *potential function* ϕ maps each data structure D_i to a real number $\phi(D_i)$, which is the potential associated with data structure D_i.

Definition 5.2. The *amortized cost* a_i of operation i with respect to the potential function ϕ equals the actual cost c_i of the i-th operation plus the increase in potential due to that operation:

$$a_i = c_i + \phi(D_i) - \phi(D_{i-1}).$$

Hence the total amortized cost of n operations $A(D_n)$ is

$$\begin{aligned} A(D_n) = \sum_{i=1}^{n} a_i &= \sum_{i=1}^{n} (c_i + \phi(D_i) - \phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0) \\ &= T(D_n) + \phi(D_n) - \phi(D_0), \end{aligned}$$

where the total cost is denoted as $T(D_n)$.

Definition 5.3. The sum of amortized costs will be an *amortized bound*, if it is an upper bound on the actual cost:

$$A(D_n) = \sum_{i=1}^{n} a_i \geq \sum_{i=1}^{n} c_i = T(D_n).$$

We conclude that there is an amortized bound whenever $\phi(D_n) \geq \phi(D_0)$. If the potential difference of an operation i is positive, the amortized cost a_i represents an overcharge and if it is negative it represents an undercharge to the operation and the potential of the data structure increases or decreases respectively. As we can see from the definition, the amortized cost depends on the potential function. Changing the potential function results in different amortized costs and hence different upper bounds on actual costs.

Amortized cost analysis serves as a powerful tool in algorithmic evaluation, providing a nuanced perspective on computational efficiency over sequences of operations. By distributing costs and incorporating potential functions, this method enables comprehensive assessment of an algorithm's average performance. It can help to identify optimal strategies, to enhance our understanding of the long-term resource requirements, contributing to the refinement and optimization of algorithms in diverse scientific and computational domains [21].

5.2 Binomial Heap Bounds

In this section, we establish an amortized bound for binomial heaps by conducting an amortized analysis using the potential function method. Our analysis builds upon the work presented in Okasaki's book "Purely functional data structures" [15]. First, we define the costs and potential function associated with our data structure.

Definition 5.4. The cost measure of `insTree`, `insert`, etc. is defined as the total number of (recursive) calls.

Definition 5.5. Let the potential ϕ of a binomial heap ts and a pair (t, ts) of a binomial tree t and heap ts be defined as

$$\phi(ts) = l(ts)$$
$$\phi((t, ts)) = l(t : ts),$$

where $l(ts)$ and $l(t : ts)$ denotes the length of the heap ts and $t : ts$ accordingly.

To prove that the length of a binomial heap is bounded by the logarithm, we need some additional lemmas. First, we require the information that the length of the heap is bounded by the largest rank plus one which we denote in Lemma 5.6. Additionally, we need Corollary 5.8 to convert a rank notation back to the number of nodes to complete the proof. By performing these steps, the proof of Lemma 5.9 presented in this thesis is a reduced, direct conversion of the proof done in Liquid Haskell.

Lemma 5.6. *Let ts be a non-empty binomial heap. Then the length of ts is bound by the greatest rank of the heap plus one:*

$$l(ts) \leq r_n + 1.$$

Here r_n denotes the rank of the last tree t_n in the heap. Because the list is ordered by rank, r_n is greatest rank of ts.

Proof. The proof is by structural induction on ts. For $ts = [t_1]$ we have

$$l([t_1]) = 1 \leq r_1 + 1.$$

Because ranks are natural numbers, this case trivially holds. We assume the induction hypothesis $l(ts) \leq r_{n-1} + 1$ for $ts = [t_1, ..., t_{n-1}]$. For $ts = [t_1, ..., t_{n-1}, t_n]$ we have

$$l(ts) = n \leq r_{n-1} + 1 + 1 \leq r_n + 1.$$

The last relation holds because the list is ordered by rank and hence $r_{n-1} < r_n$. □

Lemma 5.7. *Let t be a non-empty binomial tree with rank r. Then*

$$2^r = |t|,$$

where $|t|$ denotes the number of nodes of t.

Proof. The proof is by induction on r.
For $r = 0$: We know by definition that t consists of only one node and hence

$$2^0 = 1 = |t|.$$

For $r = n + 1$: Using our induction hypothesis and the information that a tree of rank $r + 1$ consists of two binomial trees t_1 and t_2 of rank r we conclude

$$2^{r+1} = 2^r + 2^r = |t_1| + |t_2| = |t|.$$

This is because we increase the rank only when linking two trees of the same rank. □

Corollary 5.8. *Let $ts = [t_1, ..., t_n]$ be a non-empty binomial heap and r_i denote the rank of a binomial tree t_i in the heap. Then*

$$|ts| = \sum_{i=1}^{n} 2^{r_i},$$

where $|ts|$ denotes the number of nodes of ts.

Proof. This follows as a consequence of Lemma 5.7. □

Lemma 5.9. *Let ts be a binomial heap. Then the length of ts is bound by the logarithm of the size of the heap plus one:*

$$l(ts) \leq \log_2(|ts|) + 1.$$

Proof. The base case where $ts = []$ trivially holds. Next, we consider a non-empty binomial heap and use the lemmas from before. Additionally, we apply rules of logarithm and sum. Note that log always refers to the logarithm of base two.

$$\begin{aligned} l(ts) &\leq r_n + 1 && \text{(by Lemma 5.6)} \\ &= \log 2^{r_n+1} = \log 2^{r_n} + \log 2 \\ &= \log 2^{r_n} + 1 \leq \log\left(\sum_{i=1}^{n} 2^{r_i}\right) + 1 \\ &= \log_2(|ts|) + 1. && \text{(by Corollary 5.8)} \end{aligned}$$

□

While the `insert` *function on binomial heaps runs in a worst-case time of $O(\log_2 |ts|)$, it runs in constant amortized time [15]. To show that this holds we need to prove that* `insTree` *runs in $O(1)$ amortized time.*

Lemma 5.10. *Let t be a binomial tree and ts be a binomial heap.*

$$A(\texttt{insTree}\ t\ ts) \in O(1).$$

Proof. It suffices to prove that $A(\texttt{insTree}\ t\ ts) \leq 1$.

First, we consider the case where $\texttt{insTree}\ t\ ts = \texttt{insTree}\ (\texttt{link}\ t\ t')\ ts'$, which refers to the otherwise case in Line 8 of Code Snippet 3.3. Then

$$\begin{aligned}
A(\texttt{insTree}\ t\ ts) &= T(\texttt{insTree}\ t\ ts) + \phi(\texttt{insTree}\ t\ ts) - \phi(ts) && (5.1)\\
&= 1 + T(\texttt{insTree}\ (\texttt{link}\ t\ t')\ ts') + \phi(\texttt{insTree}\ t\ ts) - \phi(ts) && (5.2)\\
&= 1 + A(\texttt{insTree}\ (\texttt{link}\ t\ t')\ ts') + \phi(ts') \\
&\quad - \phi(\texttt{insTree}\ (\texttt{link}\ t\ t')\ ts') + \phi(\texttt{insTree}\ t\ ts) - \phi(ts) && (5.3)\\
&= 1 + A(\texttt{insTree}\ (\texttt{link}\ t\ t')\ ts') + l(ts') - l(ts) && (5.4)\\
&= A(\texttt{insTree}\ (\texttt{link}\ t\ t')\ ts') \leq 1. && (5.5)
\end{aligned}$$

In Equation 5.1 we apply the definition of amortized cost. Equation 5.2 steps with cost 1 into our considered case. We can rewrite the total costs by transforming the definition of amortized costs in Equation 5.3. Next, in Equation 5.4, we cancel both potentials of `insTree` and use the length of the heap according to the definition of the potential. In the last step we employ that by definition $l(ts) < l(ts')$. The result that the amortized cost in Equation 5.5 is less than or equal to one follows by the case analysis.

Second, we consider the case $\texttt{insTree}\ t\ ts = t : ts$. This case, that can be seen in Line 7 of Code Snippet 3.3, is non-recursive and hence it is easy to show that it has constant amortized time because the costs in this case are zero.

$$\begin{aligned}
A(\texttt{insTree}\ t\ ts) &= T(\texttt{insTree}\ t\ ts) + \phi(\texttt{insTree}\ t\ ts) - \phi(ts)\\
&= \phi(t : ts) - \phi(ts) = l(t : ts) - l(ts) = 1.
\end{aligned}$$

□

Lemma 5.11. *Let x be an element and ts be a binomial heap.*

$$A(\texttt{insert}\ x\ ts) \in O(1).$$

Proof. By definition of the function we have $\texttt{insert}\ x\ ts = \texttt{insTree}\ (\text{Node}\ 0\ x\ [])\ ts$ and since the amortized cost of `insTree` is in constant time by Lemma 5.10, the same holds for the `insert` function. □

Lemma 5.12. *Let ts_1 and ts_2 be binomial heaps.*

$$A(\texttt{merge}\ ts_1\ ts_2) \in O(\log_2(|ts_1| + |ts_2|)).$$

Proof. In order to prove the lemma, we prove the stronger claim

$$A(\texttt{merge}\ ts_1\ ts_2) \leq l(ts_1) + l(ts_2).$$

First, consider the claim and the case $\texttt{merge}\ ts_1\ ts_2 = \texttt{insTree}\ (\texttt{link}\ t\ t')\ (\texttt{merge}\ ts_1'\ ts_2')$ which corresponds to the otherwise case in Line 6 of Code Snippet 3.4. Whenever we have two or more input data structures, $\phi(D_0)$ equals the sum of all input potentials which in this case is $\phi(ts_1) + \phi(ts_2)$. Then

$$\begin{aligned}
A(\texttt{merge}\ ts_1\ ts_2) &= T(\texttt{merge}\ ts_1\ ts_2) + \phi(\texttt{merge}\ ts_1\ ts_2) - \phi(ts_1) - \phi(ts_2) && (5.6)\\
&= 1 + T(\texttt{insTree}\ (\texttt{link}\ t\ t')\ (\texttt{merge}\ ts_1'\ ts_2')) + T(\texttt{merge}\ ts_1'\ ts_2')\\
&\quad + \phi(\texttt{insTree}\ (\texttt{link}\ t\ t')\ (\texttt{merge}\ ts_1'\ ts_2')) - \phi(ts_1) - \phi(ts_2) && (5.7)\\
&= 1 + A(\texttt{insTree}\ (\texttt{link}\ t\ t')\ (\texttt{merge}\ ts_1'\ ts_2'))\\
&\quad + \phi(\texttt{merge}\ ts_1'\ ts_2') - \phi(\texttt{insTree}\ (\texttt{link}\ t\ t')\ (\texttt{merge}\ ts_1'\ ts_2'))\\
&\quad + A(\texttt{merge}\ ts_1'\ ts_2') + \phi(ts1') + \phi(ts2') - \phi(\texttt{merge}\ ts_1'\ ts_2')\\
&\quad + \phi(\texttt{insTree}\ (\texttt{link}\ t\ t')\ (\texttt{merge}\ ts_1'\ ts_2')) - \phi(ts_1) - \phi(ts_2) && (5.8)\\
&= 1 + A(\texttt{insTree}\ (\texttt{link}\ t\ t')\ (\texttt{merge}\ ts_1'\ ts_2'))\\
&\quad + A(\texttt{merge}\ ts_1'\ ts_2') + \phi(ts1') + \phi(ts2') - \phi(ts_1) - \phi(ts_2) && (5.9)\\
&= 1 + A(\texttt{insTree}\ (\texttt{link}\ t\ t')\ (\texttt{merge}\ ts_1'\ ts_2'))\\
&\quad + A(\texttt{merge}\ ts_1'\ ts_2') - 2 && (5.10)\\
&\leq 1 + 1 + l(ts_1') + l(ts_2') - 2 \leq l(ts_1) + l(ts_2). && (5.11)
\end{aligned}$$

Equation 5.6 applies the definition of amortized cost. Since we consider exactly this case, our costs consist of the cost of the insertion of a tree as well as the costs of merging two smaller heaps. This can be seen in Equation 5.7, where we additionally add the number one for stepping into this case. Next, we rewrite the costs, which results in Equation 5.8. In Equation 5.9 we canceled $\phi(\texttt{merge}\ ts_1'\ ts_2')$ and $\phi(\texttt{insTree}\ (\texttt{link}\ t\ t')\ (\texttt{merge}\ ts_1'\ ts_2'))$ because adding and then subtracting them is redundant. Since the potential corresponds to the length of the heap and the length of ts_1' and ts_2' is each one less than the length of the heap ts_1 and ts_2 respectively, we know that adding the potentials results in negative two in Equation 5.10. In the last step we used the claim and the result that `insTree` has constant amortized time of Lemma 5.10.

Second, consider the case $\texttt{merge}\ ts_1\ ts_2 = t_1 : (\texttt{merge}\ ts_1'\ ts_2)$ of Line 4 of Code Snippet 3.4. Then

$$\begin{aligned}
A(\texttt{merge}\ ts_1\ ts_2) &= T(\texttt{merge}\ ts_1\ ts_2) + \phi(\texttt{merge}\ ts_1\ ts_2) - \phi(ts_1) - \phi(ts_2)\\
&= 1 + T(\texttt{merge}\ ts_1'\ ts_2)\ + \phi(t_1 : (\texttt{merge}\ ts_1'\ ts_2)) - \phi(ts_1) - \phi(ts_2)\\
&= 1 + A(\texttt{merge}\ ts_1'\ ts_2)\ + \phi(ts_1') + \phi(ts_2) - \phi(\texttt{merge}\ ts_1'\ ts_2)\\
&\quad + \phi(t_1 : (\texttt{merge}\ ts_1'\ ts_2)) - \phi(ts_1) - \phi(ts_2)\\
&= 1 + A(\texttt{merge}\ ts_1'\ ts_2)\\
&\leq 1 + l(ts_1') + l(ts_2) = l(ts_1) + l(ts_2).
\end{aligned}$$

Third, consider the case $\texttt{merge}\ ts_1\ ts_2 = t_2 : (\texttt{merge}\ ts_1\ ts_2')$ of Line 5 of Code Snippet 3.4, which can be proven analogously to the former case. This completes the proof of the stronger claim and because we know that the length of a heap is logarithmic in the size of the heap by Lemma 5.9, we conclude

$$\begin{aligned} A(\texttt{merge}\ ts_1\ ts_2) &\leq l(ts_1) + l(ts_2) \\ &\leq \log_2(|ts_1|) + \log_2(|ts_2|) + 2 \\ &\in O(\log_2(|ts_1| + |ts_2|)). \end{aligned}$$

□

Lemma 5.13. *Let ts be a binomial heap.*

$$A(\mathit{removeMinTree}\ ts) \in O(\log_2(|ts|)).$$

Proof. As before, we first prove the stronger claim that $A(\texttt{removeMinTree}\ ts) \leq l(ts)$ and then the result follows by application of Lemma 5.9.

First, consider $\texttt{removeMinTree}\ (t : ts) = (t', t : ts')$ which is the else case of Line 4 of Code Snippet 3.5 where $\texttt{removeMinTree}\ ts = (t', ts')$. Here we do the same as in the proofs before. We begin by applying the definition of amortized costs, then we rewrite the total costs and finally cancel the potentials by using the invariant that $l(ts) = l(t' : ts')$.

$$\begin{aligned} A(\texttt{removeMinTree}\ (t : ts)) = {} & T(\texttt{removeMinTree}\ (t : ts)) + \phi(\texttt{removeMinTree}\ (t : ts)) \\ & - \phi(t : ts) \\ = {} & 1 + T(\texttt{removeMinTree}\ ts) + \phi((t', t : ts')) - \phi(t : ts) \\ = {} & 1 + A(\texttt{removeMinTree}\ ts) + \phi(ts) - \phi((t', ts')) \\ & + \phi((t', t : ts')) - \phi(t : ts) \\ = {} & 1 + A(\texttt{removeMinTree}\ ts) \leq 1 + l(ts) = l(t : ts). \end{aligned}$$

Second, consider $\texttt{removeMinTree}\ (t : ts) = (t, ts)$, which is the then case of Line 4 of Code Snippet 3.5. We perform the same steps as in the first case.

$$\begin{aligned} A(\texttt{removeMinTree}\ (t : ts)) &= T(\texttt{removeMinTree}\ (t : ts)) + \phi(\texttt{removeMinTree}\ (t : ts)) \\ &\quad - \phi(t : ts) \\ &= 1 + T(\texttt{removeMinTree}\ ts) + \phi((t, ts)) - \phi(t : ts) \\ &= 1 + A(\texttt{removeMinTree}\ ts) + \phi(ts) - \phi((t', ts')) \\ &\quad + \phi((t, ts)) - \phi(t : ts) \\ &= 1 + A(\texttt{removeMinTree}\ ts) \leq 1 + l(ts) = l(t : ts). \end{aligned}$$

Hence we conclude that $A(\texttt{removeMinTree}\ ts) \in O(\log_2(|ts|))$ holds. □

Corollary 5.14. *Let ts be a binomial heap.*

$$A(\mathit{findMin}\ ts) \in O(\log_2(|ts|)).$$

Proof. The proof follows directly from the definition of `findMin` and Lemma 5.13. □

Lemma 5.15. *Let ts be a binomial heap.*

$$A(\texttt{deleteMin}\ ts) \in O(\log_2(|ts|)).$$

Proof. The claim follows essentially as a consequence of Lemma 5.12 and Lemma 5.13. First, we again apply the definition of amortized cost in Equation 5.12. Next, we split up the actual costs in Equation 5.13 to be the costs of `removeMinTree` and `merge` because of the functions definition that can be seen in Code Snippet 3.5. Equation 5.14 then rewrites the cost by transforming the definition of amortized cost. We employ the definition $\texttt{deleteMin}\ ts = \texttt{merge}\ (\texttt{reverse}\ ts_1)\ ts_2$ and cancel equal terms in Equation 5.15. In the last steps we apply the definition of the potential function ϕ. Because $l(ts_2) \leq l((\text{Node}\ r\ x\ ts_1) : ts_2)$ and a reverse list has the same length as the list itself, we can simplify the equation and apply Lemma 5.9 and the amortized costs from Lemma 5.12 and Lemma 5.13.

$$\begin{aligned}
A(\texttt{deleteMin}\ ts) &= T(\texttt{deleteMin}\ ts) + \phi(\texttt{deleteMin}\ ts) - \phi(ts) && (5.12)\\
&= T(\texttt{removeMinTree}\ ts) + T(\texttt{merge}\ (\texttt{reverse}\ ts_1)\ ts_2)\\
&\quad + \phi(\texttt{deleteMin}\ ts) - \phi(ts) && (5.13)\\
&= A(\texttt{removeMinTree}\ ts) + \phi(ts) - \phi((\text{Node}\ r\ x\ ts_1), ts_2)\\
&\quad + A(\texttt{merge}\ (\texttt{reverse}\ ts_1)\ ts_2) + \phi(\texttt{reverse}\ ts_1) + \phi(ts_2)\\
&\quad - \phi(\texttt{merge}\ (\texttt{reverse}\ ts_1)\ ts_2) + \phi(\texttt{deleteMin}\ ts) - \phi(ts) && (5.14)\\
&= A(\texttt{removeMinTree}\ ts) - \phi((\text{Node}\ r\ x\ ts_1), ts_2)\\
&\quad + A(\texttt{merge}\ (\texttt{reverse}\ ts_1)\ ts_2) + \phi(\texttt{reverse}\ ts_1) + \phi(ts_2) && (5.15)\\
&\leq A(\texttt{removeMinTree}\ ts) + A(\texttt{merge}(\texttt{reverse}\ ts_1)\ ts_2)\\
&\quad + l(\texttt{reverse}\ ts_1)\\
&\leq A(\texttt{removeMinTree}\ ts) + A(\texttt{merge}(\texttt{reverse}\ ts_1)\ ts_2) + c_3 \cdot \log_2(|ts_1|)\\
&\leq c_1 \cdot \log_2(|ts|) + c_2 \cdot \log_2(|ts_1| + |ts_2|) + c_3 \cdot \log_2(|ts_1|) \in O(\log_2(|ts|)).
\end{aligned}$$

□

5.3 Fibonacci Heap Bounds

Similarly to the previous section, we start by defining the cost measure and potential function of Fibonacci heaps to give amortized bounds for them. Our definitions, examples and lemmas are based on Cormen et al. [3].

Definition 5.16. The cost measure of `insert`, `merge`, etc. is defined as the total number of (recursive) calls.

Definition 5.17. Let the potential ϕ of a Fibonacci heap h be defined as

$$\phi(h) = l(h) + 2 \cdot m(ts),$$

where $l(ts)$ denotes the length of the heap, which is the length of the trees plus one for the minimum tree, and $m(ts)$ denotes the number of marked nodes in ts.

Example 5.18. To create an empty Fibonacci heap h we just return E. Because the length of this Fibonacci heap is zero and we cannot have any marked nodes, since there are no nodes in the heap, the potential of an empty Fibonacci heap h is $\phi(h) = 0$.

Example 5.19. Consider the Fibonacci heap of Figure 4.1. Here we have five trees in our root list including the minimum tree. Moreover, three nodes are marked. Hence the potential of this Fibonacci heap is $5 + 2 \cdot 3 = 11$.

Lemma 5.20. *Let h be a Fibonacci heap. Then*

$$D(h) \leq \lfloor \log_2 |h| \rfloor,$$

where $D(h)$ denotes the the maximum degree of h and $|h|$ denotes the number of nodes of h und $\lfloor \cdot \rfloor$ denotes the floor function [3].

Proof. For $h = []$ the claim $D(h) \leq \lfloor \log_2 |h| \rfloor$ trivially holds. The only case where the degree of a heap and hence also the maximum degree $D(h)$ increase, is if the `link` function is called. This function is only called when melding the heap. After melding, all trees have a unique degree and the Fibonacci heap can be seen as a binomial heap. Hence the proof is analogous to the proof of Lemma 5.9 where the greatest rank r_n corresponds to the maximum degree $D(h)$. □

Further, we show amortized bounds for Fibonacci heap functions. While the functions `insert`, `merge` and `findMin` have constant amortized time, the `deleteMin` function has a logarithmic bound.

Lemma 5.21. *Let h_1 and h_2 be a Fibonacci heaps.*

$$A(\texttt{merge}\ h_1\ h_2) \in O(1).$$

Proof. Consider the case `merge` $h_1\ h_2 =$ FH $minTr_1$ $(minTr_2 : ts_2 +\!+ ts_1)$ of Line 6 in Code Snippet 4.3. Then

$$\begin{aligned}
A(\texttt{merge}\ h_1\ h_2) &= T(\texttt{merge}\ h_1\ h_2) + \phi(\texttt{merge}\ h_1\ h_2) - \phi(h_1) - \phi(h_2) && (5.16)\\
&= 1 + \phi(\texttt{merge}\ h_1\ h_2) - \phi(h_1) - \phi(h_2) && (5.17)\\
&= 1 + l(\texttt{merge}\ h_1\ h_2) + 2 \cdot m(\texttt{merge}\ h_1\ h_2))) \\
&\quad - (l(h_1) + 2 \cdot m(h_1)) - (l(h_2) + 2 \cdot m(h_2)) && (5.18)\\
&= 1 + 2 \cdot m(\texttt{merge}\ h_1\ h_2))) - 2 \cdot m(h_1) - 2 \cdot m(h_2) && (5.19)\\
&= 1. && (5.20)
\end{aligned}$$

First, we apply the definition of amortized cost in Equation 5.16. Since the function `merge` is non-recursive we simply have a total cost of one in Equation 5.17 to step into our case. In Equation 5.18 we employ the definition of our potential function, which is the length added by twice the number of marked nodes. Moreover, in Equation 5.19 we employ $l(\texttt{merge}\ h_1\ h_2) = l(h_1) + l(h_2)$ and therefore we can cancel it. Last, we know that $m(\texttt{merge}\ h_1\ h_2) = m(h_1) + m(h_2)$ because we do not change any marked nodes in this function, which results in an amortized cost of constant one in Equation 5.20.

Second, the other case where $\texttt{merge}\ h_1\ h_2 = \text{FH}\ minTr_2\ (minTr_1 : ts_1 +\!+ ts_2)$ is analogous. □

Lemma 5.22. *Let h_1 and h_2 be Fibonacci heaps.*

$$A(\texttt{insert}\ h_1\ h_2) \in O(1).$$

Proof. This is a direct result from Lemma 5.21. Since the function `singleton` is just a single call with cost one and we know that `merge` has constant amortized time we get that `insert` has constant amortized time. □

Lemma 5.23. *Let h be a Fibonacci heap.*

$$A(\texttt{findMin}\ h) \in O(1).$$

Proof. Finding the minimum node of a Fibonacci heap is in $O(1)$ actual time. This is because we store minTree separately and can directly access the root node of it. Because there is no change in potential of our heap, it follows that the actual cost equals the amortized cost [3]. □

In the function `deleteMin` we delete the minimum element and add its children to the root list. Then we link trees with the same degree. Since the data structure is implemented with pointers in the literature they have direct access to parents, siblings, etc.. Moreover, `consolidate` uses an array that stores the trees by degree and links them whenever there is already a tree with the same degree. However, with our functional implementation we have to iterate through the list again and again until we reach that result. Therefore, we do not count those extra costs but rather refer to the actual costs in the literature [3].

Lemma 5.24. *Let h be a Fibonacci heap.*

$$A(\texttt{deleteMin}^*\ h) \in O(log_2(|h|)).$$

Proof. The function $\texttt{deleteMin}^*$ refers to an imperative implementation of our `deleteMin` that uses pointers and arrays and has a total cost of $T(\texttt{deleteMin}^*\ h) = l(h) + D(h)$ according to the literature. Moreover, $l(\texttt{deleteMin}^*\ h) = D(h)+1$ and $m(\texttt{deleteMin}^*\ h) =$

$2 \cdot m(h)$ because at most $D(h) + 1$ roots remain and no nodes become marked during the operation [3]. In the last step of the proof we apply Lemma 5.20.

$$\begin{aligned} A(\texttt{deleteMin}^*\ h) &= T(\texttt{deleteMin}^*\ h) + \phi(\texttt{deleteMin}^*\ h) - \phi(h) \\ &= l(h) + D(h) + \phi(\texttt{deleteMin}^*\ h) - \phi(h) \\ &= l(h) + D(h) + (l(\texttt{deleteMin}^*\ h) + 2 \cdot m(\texttt{deleteMin}^*\ h)) \\ &\quad - (l(h) + 2 \cdot m(h)) \\ &= l(h) + D(h) + D(h) + 1 + 2 \cdot m(h) - (l(h) + 2 \cdot m(h)) \\ &= D(h) + D(h) + 1 \in O(\log_2(|h|)). \end{aligned}$$

□

Finally, we summarize our results of Section 5.2 and Section 5.3 in Table 5.1.

Table 5.1: Amortized analysis of binomial heap and Fibonacci heap.

Heap Operation	Binomial heap ts	Fibonacci heap h
`makeHeap`	$O(1)$	$O(1)$
`insert`	$O(1)$	$O(1)$
`merge`	$O(1)$	$O(1)$
`findMin`	$O(\log_2(\|ts\|))$	$O(1)$
`deleteMin`	$O(\log_2(\|ts\|))$	$O(\log_2(\|h\|))$

6 Liquid Haskell

Liquid Haskell represents an advanced type checker and refinement type system extension for the Haskell programming language. Developed as an open-source project, Liquid Haskell aims to provide developers with a more precise and formal approach to specifying and verifying the correctness of Haskell programs. By introducing refinement types, Liquid Haskell enables programmers to articulate detailed assertions about the properties that data should uphold, surpassing the capabilities of the traditional Haskell type system [23].

The pivotal feature of Liquid Haskell lies in its capability to statically verify these refined types during the compilation process. This static verification aids in detecting potential errors and bugs early in the development cycle, providing developers with a powerful tool to ensure the correctness and reliability of their Haskell code. Liquid Haskell extends the advantages of formal verification to Haskell programs, enabling developers to express and verify involved properties of their code without the need for runtime checks [16].

Liquid Haskell has found applications across various domains, including critical systems where correctness is paramount. By enabling developers to provide more detailed specifications for their functions and data, Liquid Haskell contributes to writing safer, more resilient code. While the adoption of refinement types and formal verification may entail a learning curve, the heightened assurance in the correctness of Haskell programs can be invaluable, particularly in contexts where software reliability is of utmost importance [23].

In this chapter, we explain the setup and workflow of Liquid Haskell in Section 6.1, provide an introduction to refinement types in Section 6.2, followed by an introduction to proving extrinsic theorems in Section 6.3. Subsequently, we discuss the formalization of the refinements of binomial heap and Fibonacci heap functions in Section 6.4. Finally, we introduce the `Tick` library to perform an amortized cost analysis in Section 6.5.

6.1 Setup and Workflow

Before examining refinements and specifications, we explain the workflow of Liquid Haskell and our setup. To utilize Liquid Haskell, we require an SMT solver installed on our system, such as the Z3 theorem prover from Microsoft Research[1]. Liquid Haskell

[1] https://github.com/Z3Prover/z3

itself is installed and enabled by adding it as a dependency in the project's cabal file. Since we are using the Glasgow Haskell Compiler (GHC) version 9.0.1 for our project, we need to add both `liquidhaskell` and `liquid-base` to the build dependencies. This prompts cabal to automatically:

1. install Liquid Haskell,
2. instruct GHC to use Liquid Haskell during compilation,
3. display liquid type errors during compilation,
4. integrate Liquid Haskell with ghcid for Visual Studio Code.

With this setup, Liquid Haskell can be executed either from the command-line[2] or within a web browser[3]. The workflow is illustrated in Figure 6.1. According to Vazou et al. [23], Liquid Haskell requires three inputs:

1. A single Haskell source file containing both code and refinement type specifications,
2. A set of directories containing imported modules (including the Prelude) that may themselves contain specifications for exported types and functions,
3. A set of predicate fragments called qualifiers, which are utilized to infer refinement types. This set is usually empty, as the default set of qualifiers extracted from the type specifications is typically sufficient for inference.

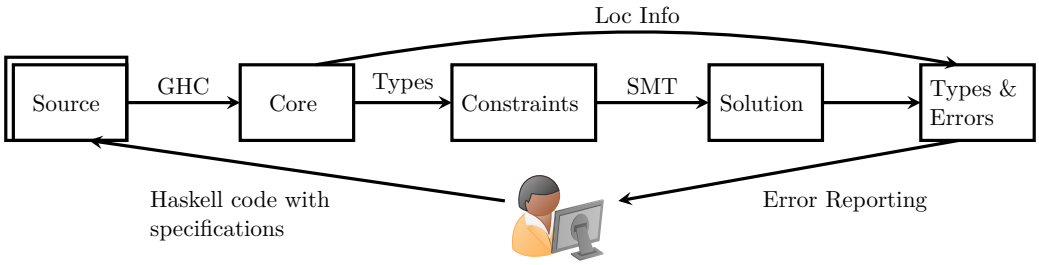

Figure 6.1: Liquid Haskell workflow [23].

The workflow of Liquid Haskell involves several procedural steps aimed at verifying the correctness of Haskell programs. Initially, GHC is utilized to transform the source code into its Core representation, as depicted in Figure 6.1. Additionally, a mapping from Core expressions to corresponding locations in the Haskell source is established to facilitate simpler error reporting at the source level. Subsequently, the abstract interpretation framework of Liquid Typing [18], adapted for soundness under lazy evaluation, is employed to derive logical constraints from the Core Intermediate Language (IL). These constraints are then resolved using a fixpoint algorithm in conjunction with an SMT solver, resulting in the deduction of a valid refinement typing for the program [22].

[2]https://hackage.haskell.org/package/liquidhaskell
[3]http://goto.ucsd.edu/liquid/haskell/demo

In terms of output, Liquid Haskell categorizes the program as SAFE if the set of constraints is satisfiable, signifying successful verification. Conversely, if the constraints prove unsatisfiable, it outputs UNSAFE. Invalid constraints are instrumental in identifying and reporting refinement type errors, pinpointing the source positions that contributed to their creation. The process leverages location information to establish a mapping between the invalid constraints and the corresponding source positions. In either outcome, Liquid Haskell generates a source map containing inferred types for each program expression, a crucial component for debugging code and specifications [23].

6.2 Refinement Types

Refinement types, also called liquid types or refinements, enable programmers to specify complex invariants by extending type systems with logical predicates [23]. We intuitively employed refinements in Code Snippet 3.1 and Code Snippet 3.2 to refine a binomial heap, and in Code Snippet 4.1 and Code Snippet 4.2 for Fibonacci heaps. These refinements allow us to specify properties like the order of ranks. Previously, when refining Fibonacci trees, we assumed refinement types like **Pos** without formally defining them. In this section, we delve into the basics of refinements. Liquid types complement the standard Haskell type. Typically, they are written as comments in the form of `{-@ annotation @-}`. While GHC ignores these comments, Liquid Haskell processes them. For simplicity, we omitted this notation in previous examples, but we use it from this point to better distinguish between the Haskell type and the refinement type.

To illustrate the concept, we start by defining the previously mentioned type **Pos** shown in Line 1 of Code Snippet 6.1. The **Pos** type encompasses all positive integers, as the logical predicate specifies that values `v` of type **Int** must be greater than zero. We provide additional examples of refinement types in Code Snippet 6.1, defining non-empty tree lists and non-empty heaps. In these cases, the length of the list or heap must be greater than zero. This showcases how setting a refinement type in such cases is straightforward.

```
{-@ type Pos = {v: Int | 0 < v} @-}
{-@ type NEBiTreeL a = {xs:[BiTree a] | len xs > 0} @-}
{-@ type NEFibTreeL a = {xs:[FibTree a] | len xs > 0} @-}
{-@ type NEBiHeap a = {ts:BiHeap a | len ts > 0} @-}
```

Code Snippet 6.1: Refinement types.

Refinement types serve to enrich the types of functions, offering to specify more precise constraints. Take, for instance, the `insTree` function of binomial heaps described in Code Snippet 3.3. Here, we can specify that the input is not just a list of binomial trees but actually a heap. Furthermore, we can refine the output of the function to be of the refinement type **NEBiHeap a** because we guarantee that at least one tree is

inserted. We can even express that the length of the output is exactly one tree larger than the input heap. In the following code snippet (Code Snippet 6.2), we provide both the Haskell type annotation as well as the Liquid Haskell type.

```
{-@ insTree :: BiTree a -> {ts : BiHeap a}
            -> {zs : NEBiHeap a | length zs ≤ length ts + 1} @-}
insTree :: Ord a => BiTree a -> [BiTree a] -> [BiTree a]
```

Code Snippet 6.2: Refinement `insTree`.

In situations, where we forget to define the refined type to the function, Liquid Haskell raises a totality error due to non-exhaustive pattern matches. This scenario might arise, for example, in the `removeMinTree` function because the case for an empty list as input is not defined. Thus, calling `removeMinTree []` would trigger a pattern error if we neglect to refine the input to be of type **NEBiHeap a**. Therefore, Liquid Haskell ensures that such errors are caught at compile-time rather than manifesting at runtime.

Liquid Haskell also enables functions to refine types, as demonstrated in Code Snippet 4.1, where we utilized the function `equalDepth` to refine the type of the subtrees of a Fibonacci tree. If we intend to incorporate such a function in the type annotation, we must lift it into the refinement logic, meaning the function must be integrated into the logical framework of the refinement type system using either **measure** or **reflect**. Measure replicates a Haskell function into the refinement logic, assigns an inferred refinement type to the constructor of the function's first argument, and generates an inferred global invariant associated with the refinement [23]. Reflect generates an uninterpreted function with identical name in the refinement logic. It duplicates the implementation to a refinement type alias and appends a refinement to the type of the uninterpreted function that specifies the type alias as a post-condition [24]. In both cases, all parts of the function definition must already be available to the refinement logic. While **measure** restricts lifting functions to those with a single argument and those that may structurally recurse on the single argument, **reflect** is less restrictive, permitting recursive functions and multiple arguments [23].

For example, to use the Haskell Prelude function `length` for the type **NEBiHeap a** or **NEHeap a**, we must lift it. Alternatively, we can employ the function `len`, which is already defined as a measure in the Liquid Haskell Prelude.

In Code Snippet 6.3, we demonstrate the process of lifting the function `emptyFibHeap`. Once we add `emptyFibHeap` as a measure, we can seamlessly utilize it in the annotation to refine the types **NEFibHeap** and **EFibHeap** without encountering any issues.

Furthermore, Liquid Haskell offers options set as pragmas. To activate theorem proving, the `"- - reflection"` option is employed, as depicted in Line 1 of Code Snippet 6.4. To enable proof by logical evaluation (PLE), we specify the `"- - ple"` option. This flag operates globally, unfolding and symbolically evaluating all terms in the specifications. It automatically provides all the resulting equations of the executed functions to the SMT

```
{-@ measure emptyFibHeap @-}
emptyFibHeap :: FibHeap a -> Bool
emptyFibHeap E = True
emptyFibHeap _ = False

{-@ type NEFibHeap = {v : FibHeap a | not emptyFibHeap v} @-}
{-@ type EFibHeap = {v : FibHeap a | (emptyFibHeap v)} @-}
```

Code Snippet 6.3: Measure `emptyFibHeap`.

solver [25]. Given the potential verbosity of refinements, we can establish predicate aliases and incorporate them within the refinements. For example, we can define a less-than relation as `Lt`, as illustrated in Line 3 of Code Snippet 6.4.

```
{-@ LIQUID "--reflection" @-}
{-@ LIQUID "--ple" @-}
{-@ predicate Lt X Y = X < Y @-}
```

Code Snippet 6.4: Pragmas and predicates.

Now that we have a grasp of the basics of writing type specifications, we can delve deeper into refining our binomial heaps and Fibonacci heaps functions, which is explored in Section 6.4.

6.3 Proving Extrinsic Theorems

If certain refinements fail to proceed automatically, one can resort to manually proving theorems in Liquid Haskell. This involves unfolding the definitions of functions and applying mathematical induction. To craft a manual proof of extrinsic theorems, we utilize the `ProofCombinators` module. This module provides access to the **Proof** type, the **QED** datatype, and the infix operator `(***)`, which converts any term x into a proof.

Furthermore, it loads some equational reasoning operators, but for our purposes, it suffices to understand the infix operator `(===)`, which denotes an implicit equality, as illustrated in Code Snippet 6.5. The expression e_1 `===` e_2 verifies whether e_1 equals e_2. To demonstrate equality, the `(===)` operator unfolds functions at most once.

Whenever we need the result of a separate proof or require an induction hypothesis, we can introduce those facts as needed using the infix operator `(?)`.

By employing these proof combinators, we can, for example, prove the property that the $\log_2$ of a number n raised to the power of two equals n. We illustrate this example in Code Snippet 6.6, assuming that we have already defined `log2` and `pow2`. Here, we

```
type Proof = ()

data QED = QED

x *** QED = ()

{-@ (===) :: x:a -> y:{a | y == x} -> {v:a | v == x && v == y} @-}
_ === y = y

{-@ (?) :: a -> Proof -> a @-}
x ? _ = x
```

Code Snippet 6.5: `ProofCombinators`.

execute a proof by induction, starting with the base case where $n = 0$ in Line 3. Step by step, we unfold the definitions of the logarithm and power of two until we arrive at the result that `log2 (pow2 0) = 0` and convert it into a proof. Subsequently, we proceed with the step case of our induction, beginning in Line 5. By unfolding the definition, we reach the step in Line 7 where we can apply our induction hypothesis with the `(?)` operator. After performing arithmetic simplification, we complete the proof.

```
{-@ logPowProp :: n:Nat -> {log2 (pow2 n) == n} @-}
logPowProp :: Int -> Proof
logPowProp 0 = log2 (pow2 0)
         === log2 1 === 0 *** QED
logPowProp n = log2 (pow2 n)
         === log2 (2 * pow2 (n - 1))
         === 1 + log2 (pow2 (n - 1)) ? logPowProp (n - 1)
         === 1 + (n - 1) === n *** QED
```

Code Snippet 6.6: Proof `logPowProp`.

Since performing all proofs manually can become tedious, we can activate the `"--ple"` flag, which conducts intermediate steps such as unfolding functions, on our behalf. However, if a proof fails with `"--ple"`, understanding Liquid Haskell's proof combinators can be helpful for reconstructing the proof and identifying where it encountered an issue.

Moreover, we can facilitate proofs and aid Liquid types in succeeding by stating verified properties using the `(??)` function, defined in Line 2 of Code Snippet 6.7. This self implemented function shown by Niki Vazou works similar to assertions but it additionally helps Liquid Haskell to succeed with proofs. Here, the first argument of the function is disregarded, but it signals to Liquid Haskell which property it might need to complete the proof. Therefore, we can simplify the proof of `logPowProp` as depicted in Line 4 and Line 5. In this case, we do not require unfolding or arithmetic simplification since those

are handled by `"- - ple"`. We just provide Liquid Haskell with the information to utilize the induction hypothesis to finalize the proof.

```
{-@ (??) :: a -> y:b -> {v:b | v == y} @-}
x ?? y = y

logPowProp 0 = ()
logPowProp n = logPowProp (n-1) ?? ()
```

Code Snippet 6.7: Function `(??)` and simplified `logPowProp`.

Although the final project employs simplified proofs, during the development process, we conducted numerous step-by-step evaluations to identify suitable invariants.

6.4 Formalization of Refinements

In Section 3.3 and Section 4.2, we presented implementations of functions for binomial heaps and Fibonacci heaps, respectively. However, we merely assumed that these functions correctly implement the respective heap structures. In practice, implementation mistakes can occur, potentially resulting in functions that return data structures that do not adhere to the intended heap properties. To mitigate this risk, we leverage Liquid Haskell's capability to specify and verify properties of Haskell programs. In this section, we refine our heap functions to ensure they adhere to the desired heap properties, using the refinement types introduced in Section 6.2.

6.4.1 Binomial Heap Refinements

We begin by examining the refinements for the functions related to binomial heaps. The binomial tree was previously annotated with Liquid Haskell refinements, as depicted in Code Snippet 3.1. In this section, we refine the heap functions to ensure they meet the specifications of a binomial heap.

An illustrative example is the `singleton` function in Code Snippet 6.8. Here, we specify that the newly created binomial tree must have rank zero, the root element must be the element x, and the list of subtrees must be empty. This refinement guarantees that the `singleton` function indeed produces a valid binomial tree with one node. While

```
{-@ singleton :: x:a -> {v:BiTree a | rank v == 0 &&
        root v == x && subtrees v == []} @-}
```

Code Snippet 6.8: Liquid Type of `singleton`.

such refinements could be applied extensively, we focus on specifications relevant to our subsequent amortized cost analysis.

Next, we consider the `insert` function. As both the input and output of `insert` are expected to be heaps, we refine the function accordingly. Additionally, since `insert` modifies the heap, we ensure that the output list of binomial trees remains ordered by rank. This requirement is expressed in the `insTree` function illustrated in Code Snippet 6.9 by stipulating that the rank of the first tree in the output heap is greater than the rank of either the tree in the first argument or the first tree in the second argument. This property, combined with Lemma 6.1 (`ordRankHProp`), guarantees that the output of `insert` remains a valid heap. Moreover, since the function inserts a tree, the output heap always contains at least one element. The ordered rank property is proven separately within Liquid Haskell.

```
{-@ insTree :: t:BiTree a -> ts:BiHeap a
        -> {zs: NEBiTreeL a | (rank (head zs) ≥ rank t ||
        rank (head zs) ≥ rank (head ts))} @-}
```

Code Snippet 6.9: Liquid Type of `insTree`.

Lemma 6.1 (`ordRankHProp`)**.** *Consider a binomial tree t and a binomial heap ts as defined in Code Snippet 3.2. If the heap is empty or the rank of t is smaller than the first tree in ts, then $(t : ts)$ is ordered by its rank:*

```
{-@ ordRankHProp :: t:BiTree a -> ts:BiHeap a -> {(length ts == 0 ||
        rank t < rank (head ts)) = ordRankH (t:ts)} @-}
```

Proof. We proof the lemma in Liquid Haskell:

```
ordRankHProp :: BiTree a -> [BiTree a] -> Proof
ordRankHProp t [] = ()
ordRankHProp t [t'] = ()
ordRankHProp t (t':ts) = ()
```

□

Incorporating refinements into the `link` function is crucial since it may be invoked when inserting a tree into our heap. It is necessary to ensure that the output of `link` indeed adheres to the properties of a binomial tree. Namely, it should be sorted, with its subtrees ordered by rank. To facilitate Liquid Haskell in verifying these properties, we provide the properties `ordRankProp` (Lemma 6.2) and `sortedProp` (Lemma 6.3).

As previously mentioned during the function's definition, `link` is exclusively called on trees with identical ranks. To enforce this condition, we refine its type and assert that the rank of the first tree is equal to that of the second one. Additionally, we specify that the rank of the output tree must be one higher than the rank of the input tree. The Liquid Haskell type specification for `link` is depicted in Code Snippet 6.10.

```
{-@ link :: t1:Bitree a -> {t2:Bitree a | rank t1 = rank t2}
         -> {v:Bitree a | rank v = rank t1 + 1} @-}
```

Code Snippet 6.10: Liquid Type of `link`.

In the `insTree` function, when invoking `link`, it inherently fulfills the precondition that both trees have equal rank. This condition is ensured by the explicit check in the otherwise case. However, in the event of an implementation error, such as mistakenly invoking `link` when the rank of the first tree is less than that of the second, Liquid Haskell detects this discrepancy and raises a liquid type mismatch error. This error indicates that the inferred type of the binomial trees does not conform to the required type, where the ranks should be equal.

Lemma 6.2 (`ordRankProp`)**.** *Consider a binomial tree t and a list of binomial trees ts which ranks are decreasing by one. Let the rank of t be one larger than the rank of the first tree in ts. Then the list $(t : ts)$ is decreasing by one:*

```
{-@ ordRankProp :: t:BiTree a -> {ts:[BiTree a] | ((length ts == 0
        || rank t == rank (head ts) +1) && ordRank ts}
        -> {ordRank (t:ts)} @-}
```

Proof. We proof the lemma in Liquid Haskell:

```
ordRankProp :: BiTree a -> [BiTree a] -> Proof
ordRankProp t [] = ()
ordRankProp t (t':ts) = ()
```

□

While certain properties may seem trivial and obvious, Liquid Haskell relies on them to prove the types.

Lemma 6.3 (`sortedProp`)**.** *Let t_1 and t_2 be a binomial trees where the root of t_1 is smaller than or equal to the root of t_2. Then the root of t_1 is smaller or equal than the root of t_2 and smaller than or equal to all roots of the subtrees of t_1.*

Proof. Straightforward by the definition of binomial trees. □

Next, we examine the `merge` function, which takes two heaps as input and returns a heap whose first tree has a rank greater than or equal to the minimum rank of the input heaps. This function is formalized as shown in Code Snippet 6.11. We require refinements of the rank to be able to use the `ordRankHProp` lemma within the function to assist Liquid Haskell in proving that the output is indeed a binomial heap.

Similarly, we refine the types of `removeMinTree` and support Liquid Haskell with the previously defined lemmas. However, for the input of the `removeMinTree` function,

```
{-@ merge :: ts1:BiHeap a -> ts2:BiHeap a
        -> {zs:BiHeap a | (length ts1 == 0 || length ts2 == 0
         || (length zs > 0 && (rank (head zs) ≥ rank (head ts1)
         || rank (head zs) ≥ rank (head ts2))))} @-}
```

Code Snippet 6.11: Liquid Type of `merge`.

we demand a non-empty heap since our intention is to delete a tree. To articulate this requirement, we utilize the pre-existing types **NEBiTreeL a** and **NEBiHeap a** to prevent totality errors stemming from non-exhaustive pattern matching. These types are indispensable for the functions `removeMinTree`, `findMin`, and `deleteMin` as they require the presence of at least one element to facilitate the retrieval or deletion process. We illustrate those liquid types in Code Snippet 6.12.

```
{-@ removeMinTree :: NEBiHeap a-> (BiTree a, BiHeap a) @-}

{-@ findMin :: NEBiHeap a-> a @-}

{-@ deleteMin :: NEBiHeap a-> BiHeap a @-}
```

Code Snippet 6.12: Liquid Type of `removeMinTree`, `findMin` and `deleteMin`.

For the `deleteMin` function, an additional lemma is required. The challenge lies in ensuring that merging reversed subtrees with a heap results in another binomial heap. This assertion is valid because the subtrees maintain a descending rank order, and when reversed, they exhibit an ascending order akin to a binomial heap. Therefore, we define Lemma 6.4, which converts the `ordRank` property to `ordRankH`.

Lemma 6.4 (`oRtoORHProp`)**.** *Consider a list of binomial trees ts in descending rank order. Then the reversed list is of ascending rank order:*

```
{-@ oRtoORHProp :: {ts:[BiTree a] | ordRank ts}
        -> {ordRankH (reverse ts)} @-}
```

Proof. The proof is straightforward by the definition of `reverse`. □

With these adjustments, we have refined the functions to meet the requirements of binomial heaps.

6.4.2 Fibonacci Heap Refinements

Next, we examine whether we need to modify the functions of Fibonacci heaps to satisfy the refinement specification. Once again, our goal is to eliminate totality errors in Liquid Haskell by refining functions with non-exhaustive pattern matching. The crucial functions

in this regard are `merge`, `consolidate`, `extractMin`, and `deleteMin`, where we need to specify that the input heap is not empty or that the input list contains at least one element and hence are of type **NEFibHeap** or **NEFibTreeL a**. The liquid types of those functions are shown in Code Snippet 6.13.

```
{-@ merge :: FibHeap a -> NEFibHeap -> NEFibHeap @-}

{-@ consolidate :: NEFibTreeL a -> [NEFibTreeL a] @-}

{-@ extractMin :: NEFibTreeL a -> (FibTree a, [FibTree a]) @-}

{-@ deleteMin :: NEFibHeap -> FibHeap a @-}
```

Code Snippet 6.13: Liquid Types of `merge`, `consolidate`, `extractMin` and `deleteMin`.

When investigating the refinement type of Fibonacci trees, we encounter stringent depth requirements that must be adhered to across all functions. However, in the `link` function, although we append one tree to another, we fail to increase the depth of the appended tree. Liquid Haskell immediately alerts us to this error, ensuring that we do not overlook the necessity of increasing the depth of the subtrees. Consequently, we introduce a function called `increaseDepth` that increments the depth of every node in the appended tree by one. However, ensuring termination can be challenging, as we must remind Liquid Haskell that we have a finite number of nodes. The termination criterion is not initially evident to Liquid Haskell because we continuously transition from one tree to many subtrees, which may appear to be incrementing. Nevertheless, all subtrees combined contain fewer nodes than their parent tree. By providing Liquid Haskell with this information, it successfully verifies the termination condition for this case. We establish the number of nodes in a tree as a termination metric in several auxiliary functions. Additionally, after increasing the depth, we must verify that the `equalDepth` property still holds. To facilitate this, we assist Liquid Haskell with Lemma 6.5, ensuring that the added tree has the same depth as the subtree of the other one. Furthermore, we ensure that the `link` function is only called when both trees have the same degree, which we explicitly include in the liquid type of the function. The other functions of Fibonacci heaps meet the specification of the liquid type with minimal alteration.

Lemma 6.5 (`eqDepthProp`)**.** *Consider a Fibonacci tree t and a list of Fibonacci trees ts, where every root of ts has the same depth. Additionally, the root node of first element of ts has the same depth as the root node of t. Then, also every root node in the list $(t : ts)$ has equal depth:*

```
{-@ eqDepthProp :: t:FibTree a -> {ts:NEFibTreeL a |
        equalDepth ts && depth t == depth (head ts)}
        -> {equalDepth (t:ts)} @-}
```

Proof. We prove the lemma in Liquid Haskell, which is straightforward:

```
eqDepthProp :: FibTree a -> [FibTree a] -> Proof
eqDepthProp t (t':ts) = ()
```

□

6.5 Program Efficiency

In addition to verifying program correctness, Liquid Haskell can also analyze program efficiency. For a cost analysis, we utilize the datatype **Tick a** from the `RTick` library, which comprises an integer tcost to track resource usage and a value tval of type `a`.

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

The **Tick a** datatype operates as a monad, and we provide a brief explanation of the main monadic methods displayed in Code Snippet 6.14 and 6.15 that we apply to the binomial and Fibonacci heap code. The `pure` function generates a **Tick** with zero cost, while `wait` accomplishes the same with a cost of one. The `step` method increments the cost of the **Tick** monad by a specified amount m.

```
{-@ pure :: x:a -> {t:Tick a | x == tval t && 0 == tcost t} @-}
pure x = Tick 0 x

{-@ wait :: x:a -> {t:Tick a | x == tval t && 1 == tcost t} @-}
wait x = Tick 1 x

{-@ step :: m:Int -> t1:Tick a -> {t:Tick a |
        tval t1 == tval t && m + tcost t1 == tcost t} @-}
step m (Tick n x) = Tick (m + n) x
```

Code Snippet 6.14: Operators of the **Tick a** datatype.

The datatype also includes functions for aggregating the costs of subexpressions. The `(<*>)` operator combines the costs of two expressions and applies the function contained in the value of the first argument to the value of the second one. Similarly, the `(</>)` operator performs the same aggregation but increments the costs by one. Additionally, the bind operator `(>>=)` aggregates the costs of subexpressions but accepts a function as an input.

We utilize `(<*>)`, `(</>)`, and `(>>=)` as infix operators. Furthermore, a function can be lifted to be of type **Tick a** without increasing the costs using the `liftM` operation.

It is crucial to avoid direct usage of the Tick data constructor. Instead, each datatype should be implicitly accessed using the aforementioned operators to prevent potential

```
{-@ (<*>) :: t1:Tick (a -> b) -> t2:Tick a
        -> {t:Tick b | (tval t1) (tval t2) == tval t &&
         tcost t1 + tcost t2 == tcost t}
@-}
Tick m f <*> Tick n x = Tick (m + n) (f x)

{-@ (</>) :: t1:Tick (a -> b) -> t2:Tick a
        -> { t:Tick b | (tval t1) (tval t2) == tval t &&
         1 + tcost t1 + tcost t2 == tcost t}
@-}
Tick m f </> Tick n x = Tick (1 + m + n) (f x)

{-@ (>>=) :: t1:Tick a -> f:(a -> Tick b)
        -> { t:Tick b | tval (f (tval t1)) == tval t &&
         tcost t1 + tcost (f (tval t1)) == tcost t }
@-}
Tick m x >>= f = let Tick n y = f x in Tick (m + n) y

{-@ liftM :: f:(a -> b) -> t1:Tick a
        -> { t:Tick b | tcost t1 == tcost t}
@-}
liftM f (Tick m x) = Tick m (f x)
```

Code Snippet 6.15: Operators of the **Tick a** datatype.

errors where costs might be inadvertently omitted. In Example 6.6, we provide both a direct and implicit usage of the data constructor.

Example 6.6. Consider the function `(++)`, which concatenates two lists. We track the number of recursive calls using the **Tick a** datatype. Initially, as depicted in Code Snippet 6.16, we employ the safe method with implicit access to the data structure through the described operators.

```
[] ++ ys = pure ys
(x:xs) ++ ys = pure (x :) </> (xs ++ ys)
```

Code Snippet 6.16: Implicit access of the Tick data constructor.

Conversely, in Code Snippet 6.17, we directly access the Tick data constructor by using tval and tcost. While this example correctly counts the desired operations, direct access to the constructor introduces potential mistakes. For instance, tcost could be assigned any value, such as 0, without triggering any Liquid Haskell warnings. Therefore, direct access poses a risk of errors and should be avoided in favor of accessing it via the

provided operators.

```
[] ++ ys = Tick 0 ys
(x:xs) ++ ys = Tick (1 + tcost (xs ++ ys)) (x : tval (xs ++ ys))
```

Code Snippet 6.17: Direct access of the Tick data constructor.

There are several other operators available, but for our purposes, the described ones are sufficient to conduct our analysis. When performing a cost analysis of our code, the objective is to tally the number of recursive calls made by the functions. This information is stored in tcost and utilized as the total cost in the amortized cost analysis. This analysis is declared within the type signature of the function and automatically verified by Liquid Haskell [10]. We provide examples of amortized cost analysis using the **Tick a** datatype in the upcoming chapter.

7 Formalization of Amortized Analysis

In this chapter, we conduct an amortized cost analysis for our implemented data structures. To accomplish this, we need to augment the existing code with the type **Tick a** as explained in Section 6.5 to monitor the number of recursive calls. This enables us to express the amortized cost within the refinement type of a function. We define a potential function for binomial heaps as well as for Fibonacci heaps. Then, we utilize the number of recursive calls and the defined potential function to specify the amortized cost within the signatures of the functions. We split this chapter into Section 7.1 that formalizes an amortized cost analysis of binomial heaps and Section 7.2 that does the same for Fibonacci heaps.

Before we delve into our fomalization, we begin by explaining a smaller formalization, namely an amortized cost analysis of a stack with the functions `pop` and `push`. Here, we simply define the potential function `pot` as length of the input list. The function `push` puts an element x first in the stack list ls. Since there is no recursive call and our costs correspond to the number of recursive calls, we have zero cost which we get by the `pure` function illustrated in Code Snippet 7.1. We analyze the amortized cost which is the actual cost zero, plus the output potential, namely the length of ls plus one, and subtract the input potential which is the length of the list ls. Hence, the amortized costs are constantly bound by one.

```
{-@ push :: ls:[a] -> x:a -> {ti:Tick [a] |
        tcost ti + pot (tval ti) - pot ls ≤ 1}
@-}
push ls x = pure (x : ls)
```

Code Snippet 7.1: Amortized cost analysis of `push`.

We perform the same analysis for `push` in Code Snippet 7.2, where we have a recursive call. This function takes as input a natural number k and a list ls, where k indicates how many first elements of ls should be removed from the stack. For totality we refine the input k to be smaller or equal than the length of the list. In the case where k equals zero, we return the list with with zero costs. Otherwise, whenever k is greater than zero, we apply `step` to increase the current costs by one for every recursive call. Again, we can directly show the amortized cost analysis within the refinement, which is illustrated in Line 2 of Code Snippet 7.2.

```
{-@ pop :: k:Nat -> {x:[a] | k ≤ len xs} -> {ti:Tick [a] |
        tcost ti + pot (tval ti) − pot ls < 1}
@-}
pop :: Int -> [a] -> Tick [a]
pop 0 ls = pure ls
pop k (x : ls) = step 1 (pop (k − 1) ls)
```

Code Snippet 7.2: Amortized cost analysis of `pop`.

7.1 Amortized Cost Analysis of Binomial Heaps

The potential function of binomial heaps corresponds to the length of the list, as defined in Definition 5.5. We employ `pot` to implement the potential function of a list and `pott` for that of a tuple as depicted in Code Snippet 7.3. Both potential functions are lifted because we require them in the refinement logic to reason about the amortized cost in the type signature of the binomial heap functions. In addition to implementing the potential function as the length, we explicitly state in the refinement type that it equals the length of the input heap. As observed in Lemma 5.9, we know that the length of a binomial heap is bounded by $\log_2 n$ plus one, where n is the number of nodes in the heap. After performing the proof of Lemma 5.9, that is directly convertible into Liquid Haskell, we can refine the output of the potential functions to be bound by the logarithm which is stated in Line 3 of Code Snippet 7.3. Here, `treeListSize` returns the number of nodes of the heap.

```
{-@ reflect pot @-}
{-@ pot :: xs:BiHeap a -> {v:Nat | v == length xs
        && v ≤ 1 + log2 (treeListSize xs)} @-}
pot :: [BiTree a] -> Int
pot [] = 0
pot (x:xs) = 1 + (pot xs)

{-@ reflect pott @-}
{-@ pott :: tup:(BiTree a, BiHeap a)
        -> {v:Nat | v == pot (snd tup)) + 1} @-}
pott :: (BiTree a, [BiTree a]) -> Int
pott (x,xs) = 1 + (pot xs)
```

Code Snippet 7.3: Potential function binomial heap.

To demonstrate the amortized cost of binomial heaps, we count the cost of each binomial heap function, add the output potential, and subtract the input potential. To track the costs, we need to slightly modify the existing functions such that they return a type **Tick a**. In the tval component, we return the same value as before, but tcost

additionally tracks the number of recursive calls.

7.1.1 Insertion

We commence by establishing that the `insert` function has constant amortized time. To achieve this, we also require `insTree` to operate in constant time. We modify the code such that it incurs zero costs whenever there are no recursive calls. This is accomplished by utilizing `pure` in the non-recursive cases. In cases where recursion occurs, we increment the cost by one.

```
{-@ insTree :: t:BiTree a -> ts:BiHeap a
        -> {ti:Tick {zs:NEBiTreeL a | ...} | ordRankH (tval ti) &&
         tcost ti + pot (tval ti) - pot ts <= 1}
@-}
insTree t [] = pure [t]
insTree t ts@(t':ts')
        | rank t < rank t' = pure (t : ts)
        | otherwise = step 1 (insTree (link t t') ts')
```

Code Snippet 7.4: Amortized cost analysis of `insTree`.

Next, we delve into the annotation of `insTree` as depicted in Code Snippet 7.4. Here, "..." serves as an abbreviation, corresponding to the same refinement of the function as before (Code Snippet 6.9). Additionally, we extend the ordered rank property to the **Tick a** level because otherwise, Liquid Haskell encounters difficulties in proving the property. With `insTree` now of type **Tick a**, we can utilize its cost and value for the amortized analysis.

As illustrated in Code Snippet 6.2, we observe that the length of the output list zs is less than or equal to the length of the input list ts plus one. This relationship must also hold for the potential, as `pot` is defined as the length of the list. Consequently, the difference between the input and output potential is less than or equal to one. Whenever this difference diminishes to less than one, signifying a reduction in the length of the output list due to tree linking, we deduce that we have entered the otherwise case (Line 8 of the implementation in Code Snippet 7.4). This implies that costs increase by one while the length of the output list decreases by one. As the costs offset the loss of output potential, the amortized cost of `insTree` remains constant.

Liquid Haskell is able to verify the amortized cost in the refinement automatically. Furthermore, we present a manual amortized cost proof in Code Snippet 7.5, which is useful for debugging purposes and to understand the intermediate steps of Liquid Haskell. We commence by examining the cost plus the output potential minus the input potential of the `insTree` function and aim to demonstrate its equivalence to one. To substantiate our proof, we look at each case of the function individually. Initially, we verify it for an empty heap. Unfolding the `insTree` function for its input by its definition, we omit the

potential of the empty list as it evaluates to zero. Subsequently, we unfold the definition of `pure` to obtain the Tick constructor. We then apply the definitions of the costs and values. Given that the potential of a list with one element equals one, we can successfully prove the base case.

We proceed in Code Snippet 7.5 by dissecting the step case into scenarios where the rank of the first input is less than the ranks of the list (Line 12) and where it is not (Line 17). In the former case, the function evaluates to `pure` $(t : ts)$, which can be further simplified to one. Conversely, in the latter case, we increment the costs by one. As the list ts' contains exactly one element less than ts, we can invoke the induction hypothesis in Line 23, and through arithmetic evaluation, we substantiate the constant amortized cost of this case. Consequently, we conclude the proof, denoted in Liquid Haskell by *** QED.

```
1  {-@ insTreePot :: t:BiTree a -> ts:BiHeap a ->
2          {tcost (insTree t ts) + pot (tval (insTree t ts))
3          - pot ts == 1}
4  @-}
5  insTreePot :: BiTree a -> [BiTree a] -> Proof
6  insTreePot t [] =
7          tcost (insTree t []) + pot (tval (insTree t [])) - pot []
8          === tcost (pure [t]) + pot (tval (pure [t]))
9          === tcost (Tick 0 [t]) + pot (tval (Tick 0 [t]))
10         === 0 + pot [t]
11         === 1
12 insTreePot t ts@(t':ts')
13     | rank t < rank t' =
14         tcost (insTree t ts) + pot (tval (insTree t ts)) - pot ts
15         === tcost (pure (t:ts)) + pot (tval (pure (t:ts))) - pot ts
16         === 0 + pot (t:ts) - pot ts
17         === 1
18     | otherwise =
19         tcost (insTree t ts) + pot (tval (insTree t ts)) - pot ts
20         === tcost (step 1 (insTree (link t t') ts'))
21             + pot (tval (step 1 (insTree (link t t') ts'))) - pot ts
22         === 1 + tcost (insTree (link t t') ts'))
23             + pot (insTree (link t t') ts')) - pot ts' - 1
24         ? insTreePot (link t t') ts'
25         === 1
26 *** QED
```

Code Snippet 7.5: Proof of amortized cost analysis of `insTree`.

The amortized cost of the `insert` function is similar to that of `insTree`, remaining

constant. To refine the `insert` function, we also extend the ordered rank property to the **Tick a** level, refining the amortized cost akin to the `insTree` function with the cost and potential difference. No alterations are required in the implementation.

```
{-@ insert :: x:a -> ts:BiHeap a -> {ti:Tick ([BiTree a]) |
        ordRankH (tval ts) &&
        tcost (tval ti) + pot (tval ti) - pot ts ≤ 2} @-}
```

Code Snippet 7.6: Amortized cost analysis of `insert`.

7.1.2 Merge

In analyzing the `merge` function of binomial heaps, we integrate the type **Tick a** in the signature. If either of the two input heaps is empty, there exists no recursive call, and we merely return the other heap, additionally returning a cost of zero using the `pure` function. Interestingly, we must aid Liquid Haskell in checking the properties by specifying that the potential of the list is non-negative.

All other scenarios entail a recursive call. We denote the `(:)` operator to possess zero costs with `pure`. Notably, the `(:)` operator must be used as a prefix operator, as otherwise, Liquid Haskell will raise an error, stating its inability to transform lambda abstraction to logic. We aggregate the costs of `(:)` and the recursive call, incrementing them by one with the function `(</>)`. We encounter an obstacle where we cannot directly

```
{-@ merge :: ts1:BiHeap a -> ts2:BiHeap a
        -> {ti:(Tick {zs:[BiTree a] | ...}) | ordRankH (tval ti) &&
        tcost ti + pot (tval ti) - pot ts1 - pot ts2 ≤ pot ts1
        + pot ts2}
@-}
merge ts1 [] = pure ts1
merge [] ts2 [] = pure ts2
merge ts1@(t1:ts1') ts2@(t2:ts2')
        | rank t1 < rank t2 = pure ((:) t1) </> merge ts1' ts2
        | rank t2 < rank t1 = pure ((:) t2) </> merge ts1 ts2'
        | otherwise =
                let Tick m x = merge ts1' ts2'
                Tick n y = insTree (link t1 t2) x in
                Tick (n + m + 1) y
```

Code Snippet 7.7: Amortized cost analysis of `merge`.

apply `merge` to the `insTree` function due to its **Tick a** type. Consequently, a function akin to `(>>=)` is necessitated to execute the function and appropriately augment the costs. However, `(>>=)` does not inherit types, posing a challenge. Given that the desired output

must be a list of binomial trees ordered by rank, the application of `(>>=)` is unsuitable. Regrettably, no other function in the `RTick` library accommodates our types.

One potential solution involves crafting a custom function, which we integrate into the trusted library. However, this trusted function must encompass all refinements of `merge` and `insTree` in its signature, necessitating a comprehensive and complex function. Given its specificity, the utility of this separate trusted function is limited, it would not be modular and its extensive nature renders it vulnerable to oversight. Consequently, we opted to entrust smaller segments within our code rather than large separate functions.

For `merge`, the segments we trust are the final three lines in Code Snippet 7.7. These lines correspond directly to the `(>>=)` operator, albeit with the slight variation of increasing the cost by one due to the recursive call.

With these adaptations, we can incorporate the amortized analysis into the refinement of **Tick a**. Here, we deduct the potential of both input heaps, ensuring that the amortized cost remains less than or equal to the length of both input heaps.

7.1.3 Deletion and FindMin

The amortized cost of finding the minimum tree is constant. It relies on the analysis of the `removeMinTree` function. Demonstrating that the cost of this function is bounded by the length of the list allows us to easily establish a constant amortized bound for `findMin`. We can then utilize `liftM` twice to apply the functions `first` and `getRoot`. As their names suggest, `first` extracts the first element of the tuple, and `getRoot` retrieves the root element of the binomial tree. Since these two functions do not incur any costs, the expense of finding the minimum element precisely corresponds to the cost of removing the minimum tree from the heap.

```
{-@ findMin :: ts:NEBiHeap a -> {ti:Tick a |
        tcost ti + 1 - pot ts <= 1} @-}
findMin ts = liftM getRoot (liftM first (removeMinTree ts))
```

Code Snippet 7.8: Amortized cost analysis of `findMin`.

Conducting the amortized cost analysis, we increment the costs by one since the potential of the output elements is one, and subtract the potential of the input heap. As the costs and potential of the input heap offset each other, we obtain one as a constant amortized bound for `findMin`.

We undertake the amortized cost analysis of the `removeMinTree` function in Line 2 of Code Snippet 7.9. Here, we augment the cost with the potential of the output tuple and deduct the potential of the input heap. We establish the length of the input heap as the amortized bound. Line 3 is essential for the amortized cost analysis of `findMin`, as previously described.

When lifting the `removeMinTree` function to the type **Tick a**, caution is warranted as we directly access the Tick constructor in the code. When we receive a heap with only one tree as input, we promptly return it as the minimum tree with zero costs. In the alternative case, we count the number of function calls by incrementing the costs by one each time. The return value remains unchanged. We must trust that we have not made a mistake and inadvertently omitted any costs in the lines where we directly access the Tick constructor.

```
{-@ removeMinTree :: ts:NEBiHeap a -> {ti:Tick (BiTree a, BiHeap a) |
        tcost ti + pott (tval ti) - pot ts ≤ pot ts &&
        tcost ti ≤ pot ts}
@-}
removeMinTree [t] = pure (t,[])
removeMinTree (t:ts)
        | root t < root t' =
                Tick (cc + 1) (t,ts)
        | otherwise =
                Tick (cc + 1) (t',t:ts)
        where (Tick cc (t',ts')) = removeMinTree ts
```

Code Snippet 7.9: Amortized cost analysis of `removeMinTree`.

Finally, we perform the amortized cost analysis of the `deleteMin` function. For brevity, we denote the lists ts_1 and ts_2 in the refinement type. While we do not have direct access to them, we access ts_1, for instance, by calling the subtrees of the first entry of the tuple of the value returned by the `removeMinTree` function. The amortized bound is then twice the potential of ts_1 and ts_2. As `merge` and `removeMinTree` are already of type **Tick a**, we simply add both costs.

```
{-@ deleteMin :: ts:NEBiHeap a -> {ti:Tick (BiHeap a) |
        tcost ti + pot (tval ti) - pot ts ≤ 2·(pot ts_1 + pot ts_2)}
@-}
deleteMin ts =
        let Tick cc (Node _ x ts_1 _, ts_2) = removeMinTree ts in
        step cc (merge (reverse ts_1) ts_2)
```

Code Snippet 7.10: Amortized cost analysis of `deleteMin`.

7.2 Amortized Cost Analysis of Fibonacci Heaps

Similarly to binomial heaps, we establish the amortized cost analysis for Fibonacci heaps, albeit with a distinct potential function. The potential function, as implemented in Code Snippet 7.11, corresponds to with its definition in Section 5.3. As previously

noted, monitoring the marked nodes is crucial for our analysis. To this end, the function `markedNodes` retrieves all marked nodes of the Fibonacci heap. Additionally, we once again tally the number of trees in the heap, which precisely corresponds to the potential function of binomial heaps, denoted as `pot`, applied to the trees in the heap, augmented by one for the tree with the minimum root, stored separately within the heap.

```
{-@ measure poth @-}
{-@ poth :: h:FibHeap a -> {v:Nat | v == 0 && emptyFibHeap h
          || v == lenght (trees h) + 1 + 2 · markedNodes h}
@-}
poth :: [a] -> Int
poth E = 0
poth h = pot (trees h) + 1 + 2 · markedNodes h
```

Code Snippet 7.11: Potential function Fibonacci heap.

Once more, we must adapt all functions to be of type **Tick a** to retain the actual costs alongside the value.

7.2.1 Insertion

We commence by demonstrating that the insertion operation of a Fibonacci heap can be executed in constant amortized time. Since insertion relies on the functions `merge` and `singleton`, we first establish that these operations are also accomplished in constant time. The creation of a singleton heap is straightforward, involving the construction of a Fibonacci heap with a single node. This process does not entail any recursive calls. Similarly, merging two Fibonacci heaps involves simply combining them and identifying the minimum tree of the resulting heap. Consequently, these operations can be lifted to the type **Tick a** using the `pure` function, which assigns zero costs.

Subsequently, we conduct an amortized cost analysis within the refinement types of the functions, as depicted in Code Snippet 7.12. For the `singleton` operation, the function incurs an actual cost of zero. We augment this with the potential of the singleton heap, which is one, as it contains only the minimum tree with all nodes unmarked. Subtracting one for the input element yields an amortized cost of zero.

Moreover, in the case of merging Fibonacci heaps, the number of trees and marked nodes remains unchanged. Consequently, the input and output potentials cancel each other out, resulting in the amortized cost of the `merge` function equating to its actual cost of zero.

We can now proceed with the analysis of the `insert` function. Here, the application of the `singleton` function to the `merge` function poses a challenge due to the requirement of the bind operator `(>>=)`, which leads to type inheritance problems. To address this issue, instead of defining a separate trusted bind function with the desired types, we

```
{-@ singleton :: x:a -> {ti:Tick NEFibHeap |
        tcost ti + poth (tval ti) - 1 < 1} @-}

{-@ merge :: h1:FibHeap a -> h2:NEFibHeap -> {ti:Tick NEFibHeap |
        tcost ti + poth (tval ti) - (poth h1 + poth h2) < 1}
@-}

{-@ insert :: h:FibHeap a -> x:a -> {ti:Tick NEFibHeap |
        tcost ti + poth (tval ti) - (poth h + 1) < 1}
@-}
```

Code Snippet 7.12: Amortized cost analysis of `singleton`, `merge` and `insert`.

opted to consider the original non-lifted `singleton` function, which is not of type **Tick a**. Consequently, we can conduct the analysis of the `insert` function similarly to the `merge` function. Alternatively, we could have cautiously used `tval` to access the type of singleton. However, this approach could introduce the risk of making mistakes and dropping costs as already mentioned in Section 6.5. Therefore, our chosen approach demonstrates that the amortized cost of insertion remains constant in our program, where we only need to trust the given library.

7.2.2 Deletion

Moving on to the deletion operation, the amortized cost analysis depends on the cost of `extractMin`, `consolidate`, and `meld`. We commence by tallying the recursive calls of the `meld` function, as illustrated in Code Snippet 7.13. In the first case, where the degrees of both compared trees are equal, a recursive call is made, resulting in an increase in costs by one. Similarly, the second case also involves recursion, with the costs incremented by one using the `(<*>)` operator. Here, we denote `(:)` as having zero costs and specify it as a prefix operator. We can conduct a cost analysis of the tree list, where the actual costs are bounded by the length of the input list, as it represents the maximum number of recursive calls that this function can make. Additionally, we need to stipulate that the length of the output list is smaller than the input list, as this information is required by Liquid Haskell to prove the type of `deleteMin`.

We can also analyze the costs associated with `consolidate`. However, due to the absence of pointers or arrays in Haskell, the `consolidate` function needs to call the `meld` function with every element until no tree with the same degree remains. With an array implementation, we could store the trees at the array position of their degree. If a tree is already stored at that position, we could directly link both trees to form a new one. Consequently, with an array implementation, we would need to iterate through the list at most once. However, we cannot achieve this level of efficiency with our current implementation. Since the actual costs would be greater in this scenario, the amortized

```
{-@ meld :: xs:[FibTree a] -> t:FibTree a ->
        {ti:Tick (NEFibTreeL a]) | tcost ti ≤ length xs
        && length (tval ti) ≤ length xs + 1}
@-}
meld [] t = pure [t]
meld (x:xs) t =
        | getDegree x == getDegree t =
                step 1 (meld xs (tval (link t x)))
        | otherwise = pure ((:) x) <*> meld xs t
```

Code Snippet 7.13: Cost analysis of `meld`.

cost of `deleteMin` cannot be strictly bound by the length of the list. Therefore, we intentionally do not account for the additional costs of `consolidate` when performing the amortized cost analysis of deletion.

```
{-@ extractMin :: ts:NEFibTreeL a -> {ti:Tick (FibTree a,[FibTree a])
        | tcost ti == pott (tval ti) && tcost ti ≤ length ts}
@-}
```

Code Snippet 7.14: Cost analysis of `extractMin`.

For the process of deleting the minimum element from the heap, we must extract the minimum element from the remaining trees. This operation is performed by the `extractMin` function, whose cost refinement is displayed in Code Snippet 7.14. As we check whether the root of each tree is smaller, we iterate through the entire list. In each recursive call, we increment the actual costs by one. Consequently, the cost is bound by the length of the list. To specify this in the refinement, we assist Liquid Haskell by providing the information, that the length of the output tuple equals the length of the input list. To reason about the length of the tuple, we reuse the functions `pott` and `pot` from the binomial heap analysis.

Since all functions in `deleteMin` are bound by the length of the list, the actual costs of deleting the minimum element are similarly bound by it. Regarding the difference in potential, we can analyze that the number of marked nodes remains the same, and the length of the tree might increase by the number of subtrees of the minimum element. Hence, the amortized cost is bound by the number of trees in the heap, as shown in the refinement of the function illustrated in Code Snippet 7.15. As we merge all trees with the same degree during deletion, our Fibonacci heap satisfies the property of a binomial heap. Therefore, we can assume that after deletion, the length of the tree list is bound by the logarithm of the number of nodes in the heap. However, this assumption does not hold universally for Fibonacci heaps. For example, after inserting five singletons into an empty heap, our Fibonacci heap consists of five trees with one node each. Obviously, in this case, the length of the heap is not bound by the logarithm.

```
{-@ deleteMin :: h:NEFibHeap -> {ti:Tick (FibHeap a) |
        (emptyFibHeap (tval ti) || tcost ti + poth (tval ti)
        - poth h <= 2 * length (trees (tval ti)) + 1)}
@-}
```

Code Snippet 7.15: Amortized cost analysis of `deleteMin`.

8 Conclusion and Future Work

In conclusion, this thesis has made significant achievements in formalizing the behavior and performance of binomial and Fibonacci heaps using Liquid Haskell. By leveraging the expressive type system and refinement capabilities of Liquid Haskell, we have provided a rigorous analysis of these classical data structures. Consequently, we elaborate on two Liquid Haskell modules: `ProofCombinators` and `RTick`. The former module delineates combinators designed to encode steps of equational reasoning concerning both the values and resource usage of annotated expressions, facilitating the execution of extrinsic proofs. Meanwhile, the latter module introduces the **Tick a** datatype and associated functions tailored for recording and modifying resource usage [10].

Throughout the thesis, we have defined the structure and behavior of binomial and Fibonacci heaps in Haskell, encoding fundamental heap operations such as insertion, merging, finding and deleting the minimum element. These definitions were then subjected to correctness proofs, where we established and verified various invariants to ensure the heaps maintain their intended properties after each operation.

Moreover, we conducted an in-depth amortized cost analysis of the heap operations, providing insights into their efficiency over a sequence of actions. Therefore, we defined the actual costs and potential functions. Additionally, we proved that the amortized cost of a binomial heap is not only bound by the length of the heap but also the logarithm of its number of nodes. The analysis was shown on paper and directly converted in Liquid Haskell which automatically checks the correctness of our proofs.

One potential direction for future work related to the thesis could involve extending the formal verification to cover additional operations or variations of binomial and Fibonacci heaps. This could include verifying operations such as `delete` in Fibonacci heaps, or exploring alternative implementations and their performance characteristics. Additionally, the formalization could be extended to include more complex data structures or algorithms, building upon the foundational work laid out in this thesis. Furthermore, exploring optimizations or refinements to the existing formalization process could also be an area of interest for future research. This could include analyzing time efficiency in an extended parallelized version in Liquid Haskell as Daiki et al. suggest in a recently published paper [4].

Bibliography

[1] G. Brodal, G. Lagogiannis, and R. Tarjan. Strict Fibonacci heaps. In *Proceedings of the forty-fourth annual ACM symposium on theory of computing*, pages 1177–1184, 2012. doi: 10.1145/2213977.2214082.

[2] B. Cherkassky, A. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical programming*, 73(2):129–174, 1996. doi: 10.1007/BF02592101.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 3rd edition, 2009. ISBN 978-0-262-03384-8.

[4] Y. Daiki and S. Nishizaki. Time efficiency analysis of parallel programs on Liquid Haskell. In *Workshop on Computation: Theory and Practice (WCTP 2023)*, pages 155–192. Atlantis Press, 2024. doi: 10.2991/978-94-6463-388-7_11.

[5] S. Das and C. Pinotti. Parallel priority queues based on binomial heaps. *Parallel Computing*, 26(11):1411–1428, 2000. doi: 10.1016/S0167-8191(00)00064-8.

[6] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987. doi: 10.1145/28869.28874.

[7] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986. doi: 10.1007/BF01840439.

[8] M. Goodrich, R. Tamassia, and M. Goldwasser. *Data structures and algorithms in Java*. John Wiley & Sons, 2014. ISBN 978-1-119-27802-3.

[9] S. Griebel. Verification of the decrease-key operation in Fibonacci heaps in Imperative HOL. Master's thesis in Informatics, Technical University of Munich, May 2020.

[10] M. Handley, N. Vazou, and G. Hutton. Liquidate your assets: Reasoning about resource usage in Liquid Haskell. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–27, 2019. doi: 10.1145/3371092.

[11] C. Josh and P. Palmer. Verified binomial and skew heaps using Liquid Haskell. `https://github.com/pzp1997/liquid-heaps/`, April 2020. Online; accessed 26 March 2024.

[12] H. Kaplan, R. Tarjan, and U. Zwick. Fibonacci heaps revisited. *arXiv preprint arXiv:1407.5750*, 2014. doi: 10.48550/arXiv.1407.5750.

[13] D. King. Functional binomial queues. In *Functional Programming*, pages 141–150. Springer, 1994. doi: 10.1007/978-1-4471-3573-9_10.

[14] D. Knuth. Big omicron and big omega and big theta. *ACM Sigact News*, 8(2):18–24, 1976. doi: 10.1145/1008328.1008329.

[15] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999. ISBN 0 521 63124 6.

[16] R. Peña. An introduction to Liquid Haskell. *preprint arXiv:1701.03320*, 2017. doi: 10.48550/arXiv.1701.03320.

[17] C. Pranjal and P. Renjith. Amortized analysis, June 2017.

[18] P. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–169, 2008. doi: 10.1145/1375581.1375602.

[19] R. Roopesh and P. Renjith. Lecture notes: Fibonacci heap, June 2017.

[20] D. Stüwe. Verification of Fibonacci heaps in Imperative HOL. Master's thesis in informatics, Technical University of Munich, April 2019.

[21] R. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985. doi: 10.1137/0606031.

[22] N. Vazou. *Liquid Haskell: Haskell as a theorem prover*. PhD thesis, University of California, San Diego, 2016.

[23] N. Vazou, E. Seidel, and R. Jhala. Liquid Haskell: Experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, pages 39–51, 2014. doi: 10.1145/2775050.2633366.

[24] N. Vazou, A. Tondwalkar, V. Choudhury, R. Scott, R. Newton, P. Wadler, and R. Jhala. Refinement reflection: Complete verification with SMT. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–31, 2017. doi: 10.48550/arXiv.1711.03842.

[25] N. Vazou, J. Breitner, W. Kunkel, D. Van Horn, and G. Hutton. Theorem proving for all: Equational reasoning in Liquid Haskell (functional pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, pages 132–144, 2018. doi: 10.48550/arXiv.1806.03541.