Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

Verified Functional Data Structures: Priority Queues in Liquid Haskell

Master's Thesis

by

MohammadMehran Shahidi

August 28, 2025

University of Kaiserslautern-Landau
Department of Computer Science
67663 Kaiserslautern
Germany

Examiner: Prof. Dr. Ralf Hinze
Michael Youssef, M.Sc.

Declaration of Independent Work

I hereby declare that I have written the work I am submitting, titled “Verified Functional Data Structures: Priority Queues in Liquid Haskell”, independently. I have fully disclosed all sources and aids used, and I have clearly marked all parts of the work — including tables and figures — that are taken from other works or the internet, whether quoted directly or paraphrased, as borrowed content, indicating the source.

Kaiserslautern, den 28.8.2025

MohammadMehran Shahidi

Abstract

Formal program verification is a powerful approach to ensuring the correctness of software systems. However, traditional verification methods are often tedious, requiring significant manual effort and specialized tools or languages [RKJ08].
This thesis explores `LiquidHaskell`, a refinement type system for Haskell that integrates SMT (Satisfiability Modulo Theories) solvers to enable automated verification of program properties [Vaz+18]. We demonstrate how `LiquidHaskell` can be used to verify correctness of priority queue implementations in Haskell. By combining type specifications with Haskell's expressive language features, we show that `LiquidHaskell` allows for concise and automated verification with minimal annotation overhead.

Contents

1. Introduction

1.1. Motivation

Data structures are fundamental in computer science, providing efficient ways to organize, store, and manipulate data. However, correctness of these structures is vital, especially in safety-critical systems such as aviation, finance, or healthcare, where software bugs can lead to catastrophic consequences. Traditional testing techniques are often insufficient to cover all execution paths or edge cases, particularly for complex data invariants.

Priority queues are one such data structure used widely in scheduling, pathfinding algorithms (e.g., Dijkstra's), and operating systems. Their correctness is essential to ensure minimal elements are accessed as expected, and operations like insertion, deletion, and merging preserve the heap property [Oka98].

Formal verification provides a promising avenue for ensuring correctness, but mainstream adoption is hindered by the complexity of existing tools. This thesis explores an approach that brings verification closer to the developer: integrating verification directly into the Haskell programming language via `LiquidHaskell`. By embedding logical specifications into types, developers can catch invariant violations at compile time—without leaving their programming environment [RKJ08].

1.2. Problem Statement

Program verification is the process of proving that a program adheres to its intended specifications. For example, verifying that the result of a `splitMin` operation on a priority queue indeed removes the minimum element and preserves the heap invariant.

While powerful tools like Coq, Agda, and Dafny enable formal proofs, they often require switching to a new language or proof assistant environment, a deep understanding of dependent types or interactive theorem proving, and significant annotation and proof overhead. These barriers limit adoption in day-to-day software development.

`LiquidHaskell` offers an alternative: a lightweight refinement type system that integrates seamlessly into Haskell. It leverages SMT solvers to check properties like invariants, preconditions, and postconditions automatically, thus reducing manual proof effort [Vaz+18].

1.3. Goals and Contributions

This thesis aims to bridge the gap between practical programming and formal verification by demonstrating how Liquid Haskell can be used to verify the correctness of priority queue implementations.

The key contributions are as follows. First, we implement multiple functional priority queue variants (e.g., Leftist Heap, Binary Heap) in Haskell. Second, we encode structural and behavioral invariants using refinement types in `LiquidHaskell`. Third, we demonstrate verification of correctness properties directly in Haskell with minimal annotation. Finally, we evaluate the ease, limitations, and effort required in this approach compared to traditional theorem provers.

This integrated approach allows both implementation and verification to happen in the same language and tooling ecosystem, making verified software development more accessible.

1.4. Structure of the Thesis

The rest of this thesis is structured as follows:

- **Chapter 2** provides background on functional data structures, priority queues, and program verification techniques, along with related work.
- **Chapter 3** describes the design and implementation of different priority queue variants in Haskell.
- **Chapter 4** introduces Liquid Haskell, its syntax, verification pipeline, and its strengths and limitations.
- **Chapter 5** demonstrates the verification of priority queue operations using Liquid Haskell, including encoding invariants, use of refinement types, and example proofs.
- **Appendices** contain the complete verified code and additional implementation insights.

2. Background and Related Work

2.1. Functional Data Structures

Functional data structures are *immutable*, meaning their state cannot be changed after creation, and *persistent*, allowing access to previous versions of the structure. Combined with recursive algebraic data types (ADTs), this enables efficient and elegant implementations that are often easier to reason about compared to their imperative counterparts [Oka98].

In contrast, imperative and mutable data structures permit in-place modifications, which can introduce side effects such as data races or unintended state changes in concurrent environments. By ensuring that each operation produces a new version of the structure without altering the original, functional data structures provide strong guarantees of referential transparency and purity. These properties not only improve modularity and composability but also facilitate formal reasoning and verification, since invariants are preserved across all versions of the structure [Oka98].

This thesis focuses on functional data structures, specifically priority queues, due to their suitability for formal verification and the rich body of existing research in this area.

2.2. Program Verification Techniques

Overview of Hoare logic, model checking, interactive theorem proving, etc.

2.3. Related Work

Comparison with Coq, Agda, Dafny, or other tools verifying similar structures.

3. Priority Queue Implementations

3.1. Specification of Priority Queue Interface

Priority queues are multisets with an associated priority for each element, allowing efficient retrieval of the element with the highest (or lowest) priority. To avoid confusion with FIFO queues, we will refer to them as "heaps" throughout this thesis.

Typical operations include:

- **insert**: Add a new element with a given priority.
- **findMin**: Retrieve the element with the minimum key (in the min-heap variant).
- **splitMin**: Return a pair consisting of the minimum key and a heap with that minimum element removed.
- **merge**: Combine two priority queues into one.

Below is the specification of the priority queue interface, defined as a Haskell type class. `MinView` is a utility type to represent the result of the `splitMin` operation, which returns the minimum element and the remaining heap.

```haskell

data MinView q a =
EmptyView | Min {minValue :: a, restHeap :: q a}
deriving (Show, Eq)

class PriorityQueue pq where
empty :: (Ord a) => pq a
isEmpty :: (Ord a) => pq a -> Bool
findMin :: (Ord a) => pq a -> Maybe a
insert :: (Ord a) => a -> pq a -> pq a
splitMin :: (Ord a) => pq a -> MinView pq a
```

Listing 3.1: *Leftist Heap Implementation in Haskell*

Priority queues are widely used in computer science and engineering. They play a central role in *operating systems* for task scheduling, in *graph algorithms* such as Dijkstra's shortest path and Prim's minimum spanning tree, and in *discrete event simulation*, where events are processed in order of occurrence time. Other applications include data compression (e.g., Huffman coding) and networking (packet scheduling).

In this thesis, we focus on the *min-priority queue*, where elements with lower keys are considered higher priority. We will study and verify functional implementations of *Leftist Heaps* priority queues.

3.2. Leftist Heap Implementation

Leftist heaps, introduced by Crane [Cra72] and discussed extensively by Knuth [Knu73], are a variant of binary heaps designed to support efficient merging. They are defined by two key invariants:

- **Heap Property** – For every node, the stored key is less than or equal to the keys of its children. This ensures that the minimum element is always found at the root.
- **Leftist Property** – For every node, the rank (also called the *right spine length*, i.e., the length of the rightmost path from the node in question to an empty node) of the left child is greater than or equal to that of the right child. This property ensures that the right spine of the heap is kept as short as possible, which in turn guarantees logarithmic time complexity for merging operations.

We represent leftist heaps using a recursive algebraic data type in Haskell, as described by Okasaki [Oka98]:

```haskell
data LeftistHeap a
  = EmptyHeap
  | HeapNode
    { value :: a
      , left   :: LeftistHeap a
      , right :: LeftistHeap a
      , rank   :: Int
    }
```

Listing 3.2: *Leftist Heap data type*

Each node contains a value, its left subtree, right subtree, and its rank.

The merge operation merges the right subtree of the heap with the smaller root value with the other heap. After merging, it adjusts the rank by swapping the left and right subtrees if necessary using the function `makeHeapNode`.

```haskell

heapMerge :: (Ord a) => LeftistHeap a -> LeftistHeap a
    -> LeftistHeap a
heapMerge EmptyHeap EmptyHeap = EmptyHeap
heapMerge EmptyHeap h2@(HeapNode _ _ _ _) = h2
heapMerge h1@(HeapNode _ _ _ _) EmptyHeap = h1
heapMerge h1@(HeapNode x1 l1 r1 _) h2@(HeapNode x2 l2
    r2 _)
| x1 <= x2 = makeHeapNode x1 l1 (heapMerge r1 h2)
| otherwise = makeHeapNode x2 l2 (heapMerge h1 r2)
```

Listing 3.3: *Leftist Heap merge*

Because the the right spine is kept short by the leftist property and at most is logarithmic, the merge operation runs in $O(\log n)$ time.

```

```

```haskell
makeHeapNode :: a -> LeftistHeap a -> LeftistHeap a ->
    LeftistHeap a
makeHeapNode x h1 h2
| rrank h1 >= rrank h2 = HeapNode x h1 h2 (rrank h2 + 1)
| otherwise = HeapNode x h2 h1 (rrank h1 + 1)
```

Listing 3.4: *Leftist Heap helper functions*

Other functions are straightforward to implement.

```haskell
heapEmpty :: (Ord a) => LeftistHeap a
heapEmpty = EmptyHeap

heapFindMin :: (Ord a) => LeftistHeap a -> Maybe a
heapFindMin EmptyHeap = Nothing
heapFindMin (HeapNode x _ _ _) = Just x

heapIsEmpty :: (Ord a) => LeftistHeap a -> Bool
heapIsEmpty EmptyHeap = True
heapIsEmpty _ = False

heapInsert :: (Ord a) => a -> LeftistHeap a ->
    LeftistHeap a
heapInsert x h = heapMerge (HeapNode x EmptyHeap
    EmptyHeap 1) h

heapSplit :: (Ord a) => LeftistHeap a -> MinView
    LeftistHeap a
heapSplit EmptyHeap = EmptyView
heapSplit (HeapNode x l r _) = Min x (heapMerge l r)
```

In the chapter 5, we will verify that these implementations satisfy the priority queue interface and maintain the leftist heap invariants.

4. Liquid Haskell

4.1. Overview of Liquid Haskell

In this chapter, we focus on `LiquidHaskell`, a tool that extends Haskell with refinement types.

4.2. Refinement Types

Refinement types extend conventional type systems by attaching logical predicates to types. This allows for more precise type specifications and can potentially detect more errors at compile time [**vazou2014**].

Consider the following function:

```haskell
divide :: Int -> Int -> Int
```

The standard type system ensures that the function `divide` takes two integers and returns an integer. For example, if we call `divide` with arguments of type `Bool`, the type system will show the error at compile time. However, it does not detect the error if the function is called with the second argument being zero.

In a refinement type system, we can define more precise types as follows:

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

These are refinements of the basic `Int` type, where the logical predicates state that `v` is strictly positive (`Pos`) and non-negative (`Nat`), respectively. We can use these refinement types to annotate functions with preconditions and postconditions. For instance:

```haskell
divide :: Nat -> Pos -> Int
```

This type signature specifies that the function `divide` takes a non-negative integer as its first argument and a positive integer as its second argument. Consequently, if we call `divide`, the type checker will verify if the specifications meet. For instance, the following function is rejected by the type checker:

```haskell
bad :: Nat -> Nat -> Int
bad x y = x `div` y
```

To be able to verify this, the refinement type system translates the annotation into a so-called *subtyping* query as follows [**vazou2014**]:

$$\begin{matrix} x : \{\, x : \mathrm{Int} \mid x \geq 0\}, \\ y : \{\, y : \mathrm{Int} \mid y \geq 0\} \end{matrix} \vdash \{\, y : \mathrm{Int} \mid y \geq 0\} \preceq \{\, v : \mathrm{Int} \mid v > 0\}.$$

The notation $\Gamma \vdash \tau_1 \preceq \tau_2$ means that in the type environment Γ, τ_1 is a subtype of τ_2. The subtype query states, given the type environment in which x and y have type `Nat`, the type of y should be a subtype of `divide`'s second parameter v where v is a positive integer. The type system then translates this query into a verification condition (VC)- logical formulas whose validity ensures that the type specification is satisfied [**vazou2014**]. The translation of the subtyping query to VCs is shown in Figure 4.1. Based on this translation we would have the following VC:

$$(x \geq 0) \land (y \geq 0) \Rightarrow (v \geq 0) \Rightarrow (v > 0) \tag{4.1}$$

$$\begin{aligned}
(\!|\Gamma \vdash b_1 \preceq b_2|\!) &\doteq (\!|\Gamma|\!) \Rightarrow (\!|b_1|\!) \Rightarrow (\!|b_2|\!) \\
(\!|\{x : \text{Int} \mid r\}|\!) &\doteq r \\
(\!|x : \{v : \text{Int} \mid r\}|\!) &\doteq \text{“x is a value“} \Rightarrow r[x/v] \\
(\!|x : (y : \tau_y \rightarrow \tau)|\!) &\doteq \text{true} \\
(\!|x_1 : \tau_1, \ldots, x_n : \tau_n|\!) &\doteq (\!|x_1 : \tau_1|\!) \land \cdots \land (\!|x_n : \tau_n|\!)
\end{aligned}$$

Figure 4.1.: *Notation: Translation to VCs [***vazou2014***]*

This VC is meant to express that, under the environment where x and y are non-negative, the property “if v is non-negative then v is strictly positive" must hold. This is unsatisfiable since 0 is non-negative but not strictly positive, which is what the verifier should detect for the `bad` function.

Refinement type systems are designed to exclude any arbitrary functions and only include formulas from decidable logics[**vazou2014**]. These VCs are then passed to an SMT solver to check their satisfiability. In this case, the SMT solver would reject the `bad` function as the VC is unsatisfiable. In the next section, we provide a brief introduction to SMT solvers and how they can be used in the context of `LiquidHaskell`.

4.3. Specification Language

How to write and interpret refinement types.

4.4. Verification Workflow

From writing Haskell code to getting verified guarantees via Liquid Haskell.

4.5. Strengths and Limitations

Where it excels and where it struggles (e.g., termination proofs, higher-order functions).

4.6. Related Work

Comparison with Coq, Agda, Dafny, or other tools verifying similar structures.

5. Verification in Liquid Haskell

5.1. Encoding Invariants

How structural and behavioral invariants are expressed in refinement types.

5.2. Use of Measures and Predicates

Defining custom properties over data.

5.3. Example Proofs

Walkthroughs of insert and deleteMin correctness.

5.4. Dealing with Termination and Recursion

How Liquid Haskell checks termination.

5.5. Challenges and Workarounds

What was hard to prove and how you solved it.

List of Figures

A. My Code

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

B. My Ideas

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Bibliography

[Cra72] Clark Crane. *Linear Lists and Priority Queues as Balanced Binary Trees.* Technical Report. Carnegie Mellon University, 1972.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* 1st. Reading, Massachusetts: Addison-Wesley, 1973. ISBN: 0-201-03803-X.

[Oka98] Chris Okasaki. *Purely Functional Data Structures.* Cambridge: Cambridge University Press, 1998. ISBN: 978-0-521-63124-2. DOI: `10.1017/CBO9780511530104`. URL: `https://www.cambridge.org/core/books/purely-functional-data-structures/0409255DA1B48FA731859AC72E34D4` (visited on 04/06/2025).

[RKJ08] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. "Liquid Types". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '08. New York, NY, USA: Association for Computing Machinery, June 7, 2008, pp. 159–169. ISBN: 978-1-59593-860-2. DOI: `10.1145/1375581.1375602`. URL: `https://dl.acm.org/doi/10.1145/1375581.1375602` (visited on 03/03/2025).

[Vaz+18] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. "Refinement Reflection: Complete Verification with SMT". In: *Proceedings of the ACM on Programming Languages* 2 (POPL Jan. 2018), pp. 1–31. ISSN: 2475-1421. DOI: `10.1145/3158141`. URL: `https://dl.acm.org/doi/10.1145/3158141` (visited on 12/15/2024).