Verified Functional Data Structures: Priority Queues in Liquid Haskell

Master's Thesis

by

MohammadMehran Shahidi

September 21, 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 21.9.2025

_____

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

Todo list

1. Introduction

1.1. Motivation

Data structures are fundamental in computer science, providing efficient ways to organize, store, and manipulate data. However, ensuring the 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, widely used in scheduling, pathfinding algorithms (e.g., Dijkstra's), and operating systems. Their correctness is essential to ensure that minimal elements are accessed as expected, and that 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 while preserving 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 effort [Vaz+18]. 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 such as 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 `LiquidHaskell` can be used to verify the correctness of priority queue implementations.

The key contributions are as follows. First, we implement a priority queue variant, *Leftist Heap*, in Haskell. Second, we encode structural and behavioral invariants using refinement types in `LiquidHaskell`. Third, we demonstrate the 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 take place 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 *Leftist Heap* in Haskell.

- **Chapter 4** introduces `LiquidHaskell`, its syntax, verification pipeline, and its strengths and limitations.

- **Chapter 5** demonstrates the verification of priority queue operations using `LiquidHaskell`, 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

This chapter provides the necessary background for understanding the contributions of this thesis. We begin by introducing functional data structures, focusing on the principles of immutability and persistence that make them amenable to formal verification. We then survey program verification techniques, from foundational concepts like Hoare logic to modern automated methods based on refinement types and SMT solvers. Finally, we present a detailed comparison of related work, evaluating how other verification frameworks like Coq, Agda, and Dafny have been used to verify the correctness of similar data structures.

2.1. Functional Data Structures

Functional data structures are fundamentally **immutable**, meaning their state cannot be changed after creation [Oka98]. This design choice is crucial for ensuring **referential transparency**—a property where an expression can be replaced by its value without altering the program's behavior. In-place modifications, common in imperative data structures, introduce side effects that violate this property and complicate reasoning, especially in concurrent environments [Oka98].

To work with immutable structures, operations do not modify them directly. Instead, they produce a new, updated version while preserving the original. This approach is known as **persistence**. The seminal work in this area, Chris Okasaki's *Purely Functional Data Structures*, provides the foundation for implementing these structures efficiently [Oka98]. The key to their performance is **structural sharing**. For example, when adding an element to a tree, only the nodes on the path to the new element are recreated. The remainder of the tree is unchanged and shared between the old and new versions, minimizing memory overhead.

These combined properties—immutability to guarantee referential transparency, and persistence made efficient through structural sharing—not only improve modularity but also greatly facilitate formal reasoning and verification, as invariants are preserved across all versions of the structure [Oka98].

In the following chapter, we will explore priority queues, a specific type of functional data structure that benefits from these principles.

2.2. Program Verification Techniques

The goal of program verification is to formally prove that a program behaves according to its specification. A foundational approach is **deductive veri-**

fication, which uses logical reasoning to establish correctness. This is often based on the principles of Hoare logic, where programs are annotated with pre-conditions and post-conditions [Hoa69].

Traditionally, proving these properties required significant manual effort using interactive theorem provers. However, modern techniques increasingly focus on automation by leveraging **Satisfiability Modulo Theories (SMT) solvers**. These solvers are powerful engines that can automatically determine the satisfiability of logical formulas over various background theories (e.g., integers, arrays, bit-vectors). The groundwork for combining logical theories, which is central to SMT solvers, was laid by seminal works such as the Nelson-Oppen procedure [NO79] and Shostak's method [Sho84].

This thesis focuses on **refinement types**, a lightweight verification technique that extends a language's type system to encode logical predicates. The concept was notably advanced in the paper "Liquid Types" by Rondon, Jhala, and Kawaguci, who proposed a system to automatically infer and check refinement types using an SMT solver [RKJ08]. Their primary motivation was to reduce the significant annotation and proof burden associated with full dependent type systems, making formal verification more accessible to programmers. `LiquidHaskell`, the tool used in this thesis, is a direct evolution of this work, integrating refinement type checking seamlessly into the Haskell development environment [Vaz+14]. By embedding specifications directly into types, developers can catch invariant violations at compile time with a high degree of automation.

2.3. Related Work

The verification of data structure invariants has been a long-standing goal in the formal methods community. While this thesis uses `LiquidHaskell`, other powerful tools exist, each with different trade-offs regarding automation, expressiveness, and proof effort. We compare our approach with verification in Coq, Agda.

2.3.1. Verification in Interactive Theorem Provers (Coq and Agda)

Interactive theorem provers like Coq and Agda represent the gold standard for formal verification, offering a very high degree of assurance. Type-level computation is employed to enable rigorous reasoning about the termination of user-defined functions. This approach requires users to provide lemmas or rewrite hints to assist in proving properties within decidable theories [Vaz+18].

In **Coq**, data structures are commonly verified using a model-based approach. The process involves defining a logical model—a multiset of elements, represented as a list where order is proven irrelevant—and then proving that a concrete implementation correctly simulates the abstract operations on the multiset [ATC]. This involves defining the data structure, its representation invariants, and its operations within Coq's logic (the Calculus of Inductive Constructions), and then interactively proving the correspondence using tac-

tics. While this method is powerful for ensuring correctness, it can be labor-intensive and require significant expertise in formal methods.

Agda is a dependently-typed functional programming language where proofs are programs and propositions are types (the Curry-Howard correspondence). This allows properties to be encoded directly in the types of the data structures themselves. This approach ensures that any well-typed implementation is correct by construction. However, like Coq, it demands a deep understanding of dependent type theory and can lead to complex type definitions and proof terms that require significant manual development.

Compared to `LiquidHaskell`, these systems provide stronger guarantees (as the entire logic is verified within a trusted kernel), but at the cost of automation. `LiquidHaskell`, by contrast, outsources proof obligations to an external SMT solver, automating large parts of the verification process and requiring less interactive guidance from the user.

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.

```
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 [Dij59] and Prim's minimum spanning tree [Pri57], and in *discrete event simulation*, where events are processed in order of occurrence time. Other applications include data compression (e.g., Huffman coding [Huf52]) 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[Oka98].

Figure 3.1.: *Visualization of Leftist Heap Properties*
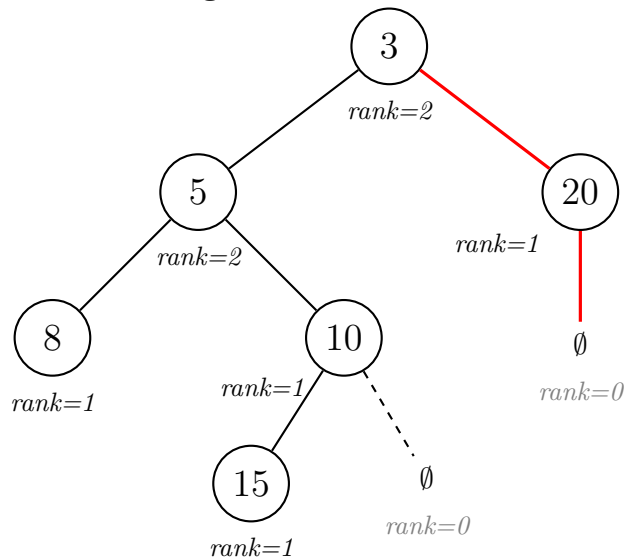
Heap Property: Parent $\leq$ Child (e.g., $3 \leq 5$, $3 \leq 20$, $5 \leq 8$)

Leftist Property: rank(left) $\geq$ rank(right)

Right Spine: Path highlighted in red. The *rank* of a node is the length of its right spine. For the root, the path is $3 \rightarrow 20 \rightarrow \emptyset$, so its rank is 2.

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. LiquidHaskell Overview

`LiquidHaskell` is a static verification tool that extends Haskell with *refinement types*. In essence, it augments Haskell's type system with logical predicates that are automatically checked by an SMT (Satisfiability Modulo Theories) solver [Vaz+14]. This combination makes it possible to verify properties of Haskell programs in a lightweight and automated way.

`LiquidHaskell` is implemented as a GHC plugin and works directly on standard Haskell code. Programmers can enrich type signatures with logical refinements, such as bounds on integers, shape properties of data structures, or functional invariants. During compilation, `LiquidHaskell` generates *subtyping queries* from these annotations and delegates them to an SMT solver. If the queries are valid, the program is accepted as verified; otherwise, Liquid Haskell produces verification errors.

Compared to traditional interactive theorem provers, `LiquidHaskell` emphasizes automation and minimal annotation overhead. Its design philosophy is to preserve Haskell's expressiveness while enabling program verification as a natural extension of the type system. This makes it particularly suitable for verifying properties of functional data structures, where invariants such as ordering, balance, or size constraints can be expressed concisely at the type level.

In the remainder of this chapter, we present the specification language of `LiquidHaskell` (Section 4.1), and discuss its strengths, limitations, and relation to other verification frameworks.

4.1. Type Refinement

Refinement types extend conventional type systems by attaching logical predicates to base types. This enables more precise specifications and allows certain classes of errors to be detected statically at compile time [Vaz+14].

Consider the following function:

```
lookup :: Int -> [Int] -> Int
lookup 0 (x : _) = x
lookup x (_ : xs) = lookup (x - 1) (xs)
```

The Haskell type system ensures that `lookup` takes an integers, a list of intergers and returns an integer. For example, an application `lookup True [3]` is rejected because the first argument has type `Bool`. However, the standard type system does not rule out the erroneous call `lookup -1 [3]`.

Of course, we could use Haskell's `Maybe` type to indicate that the function returns `Nothing` for out-of-bounds indices. However, this merely shifts the

handling of invalid inputs to the caller, who must remember to check for the `Nothing`.

With refinement types, we can express stronger specifications. In `LiquidHaskell`, refinements are written inside comments marked by `-@` and `@-`. For instance, we can define non-negative integers as:

```
{-@ x :: {v:Int | v >= 0} @-}
x :: Int
x = 2
```

A refinement type has the general form

$$\{\, v : T \mid e \,\},$$

where T is a Haskell type and e is a logical predicate over the distinguished value variable v. The type denotes the set of all values $v : T$ for which e holds [Vaz+14].

For example, the type $\{v : \text{Int} \mid v \geq 0\}$ describes all non-negative integers. For brevity, one can use type aliases or predicates to define commonly used refinements or predicates:

```
predicate Btwn Lo N Hi = Lo <= N && N < Hi
type Nat =  {v:Int | v >= 0}
```

Additionally, Holes can be used for Haskell types, as those types can be infereded from the regualar Haskell type signature or via GHC's type inference [VSJ14].

In `LiquidHaskell`, Constants such as integers and booleans are given singleton types, i.e., types that describe precisely one value [Vaz24]. The typing rule for integer literals is:

$$\frac{}{\Gamma \vdash i : \{\, v : \text{Int} \mid v = i \,\}} \quad (\textit{T-Int})$$

Here, the environment Γ contains bindings of program variables to their refinement types. One important aspect of refinement types is that expressions can be assigned to multiple types. For instance, the integer literal 3 has type $\{v : \text{Int} \mid v = 3\}$, but also any supertype, such as $\{v : \text{Int} \mid v \geq 0\}$. Crucially, refinement type systems support *subtyping*: if τ_1 is a subtype of τ_2, then any expression of type τ_1 may safely be used where τ_2 is expected:

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_1 \preceq \tau_2}{\Gamma \vdash e : \tau_2} \quad (\textit{SUBTYPE})$$

As an illustration, consider the invalid binding:

```
{-@ x :: Nat @-}
x = -1
```

By rule *T-Int*, the literal -1 has type $\{v : \text{Int} \mid v = -1\}$. To assign it to `x` of type `Nat`, the checker must show:

$$\emptyset \;\vdash\; \{v : \text{Int} \mid v = -1\} \;\preceq\; \{v : \text{Int} \mid v \geq 0\}.$$

This so called *subtyping query* is then translated into a logical implication, known as a *verification condition (VC)*:

$$(v = -1) \;\Rightarrow\; (v \geq 0).$$

These logical formula then passed to an SMT solver for validation. Since the formula is unsatisfiable, the assignment is rejected.

Figure 4.1 summarizes the notation used to translate subtyping queries into VCs [Vaz+14].

$$
\begin{aligned}
(|\Gamma \vdash b_1 \preceq b_2|) &\doteq (|\Gamma|) \Rightarrow (|b_1|) \Rightarrow (|b_2|) \\[6pt]
(|\{x : \mathrm{Int} \mid r\}|) &\doteq r \\[6pt]
(|x : \{v : \mathrm{Int} \mid r\}|) &\doteq \text{``x is a value''} \Rightarrow r[x/v] \\[6pt]
(|x : (y : \tau_y \to \tau)|) &\doteq \mathrm{true} \\[6pt]
(|x_1 : \tau_1, \ldots, x_n : \tau_n|) &\doteq (|x_1 : \tau_1|) \wedge \cdots \wedge (|x_n : \tau_n|)
\end{aligned}
$$

Figure 4.1.: *Notation: Translation to VCs [Vaz+14]*

4.1.1. Function Contracts

Refinements can also be used to specify function contracts, i.e., pre- and post-conditions. For lookup, we can require that the index is non-negative and less than the length of the list:

```
lookup :: i : Nat -> xs : {[a] | i < len xs} -> a
```

The type of second argument states that the list `xs` must have length greater than `i`. `len` is a function defined by `LiquidHaskell` in the refinement logic that returns the length of the list. In Section 4.1.3, we will show how to define and use user-defined functions in the refinement logic.

4.1.2. Refined Data Types

In the previous examples, we saw how refinements of input and output of function allow us to have stronger arguments about our program. We can take this further by refining the data types. We use the following example as an illustration, following [JSV20]:

```
data Slist a = Slist {
size :: Nat,
elems :: {v:[a] | len v == size}
}
```

This refined *Slist* data type ensures the stored 'size' always matches the length of the 'elems' list, as formalized in the refinement annotation. This ensures that the size of the list is always correct.

In the following section, we show how can we use reflection or measure directives to reason about user-defined Haskell function in the refinement logic.

4.1.3. Lifting Functions to the Refinement Logic

When our programs become more complex, we need to define our own functions in the refinement logic and reason about a function within another function. Refinement Reflection allows deep specification and verification by reflecting the code implementing a Haskell function into the function's output refinement type [Vaz+18]. That means we are able to reason about the function's behavior directly in the refinement logic. There are two ways to do this: `reflection` and `measure`.

`Measure` can be used on a function with one argument which is a Algebraic Data Type (ADT), like a list [Vaz24]. Consider the data type of a bag (multiset) defined as a map from elements to their multiplicities:

```
data Bag a = Bag { toMap :: M.Map a Int } deriving Eq
```

Now we can define a measure `bag` that computes the bag of elements for a list:

```
{-@ measure bag @-}
{-@ bag :: Ord a => [a]-> Bag a @-}
bag :: (Ord a) => [a] -> Bag a
bag [] = B.empty
bag (x : xs) = B.put x (bag xs)
```

`LiquidHaskell` lifts the Haskell function to the refinement logic, by refining the types of the data constructors with the definition of the function[Vaz24]. For example, `bag` measure definition refines the type of the `List`'s constructor to be:

```
Nil  :: {v:List a | bag v = B.empty}
Cons :: x:a -> l:List a -> {v:List a | bag v = B.put x
    (bag l)}
```

Thus, we can use the `bag` function in the refinement logic to reason about invariants of the `List` data type. For instance, in the following example:

```
{-@ equalBagExample1 :: { bag(Cons 1 (Cons 3 Nil)) ==
    bag( Cons 2 Nil) } @-}

>>     VV : {v : () | v == GHC.Tuple.Prim.()}
>>      .
>>     is not a subtype of the required type
>>       VV : {VV##2465 : () | bag (Cons 1 (Cons 3 Nil))
    == bag (Cons 2 Nil)}
```

The {x = y} is shorthand for {v :() | x = y}, where x and y are expressions. This formulation is motivated by the fact that the equality predicate $x = y$ is a condition that does not depend on any particular value. Note that equality for bags is defined as the equality of the underlying maps that already have a built-in equality function.

Reflection is another useful feature that allows the user to define a function in the refinement logic, providing the SMT solver with the function's behavior [Vaz+18]. This has the advantage of allowing the user to lift in the logic functions with more than one argument, but the verification is no more automated

[Vaz24]. Additionally, with the use of a library of combinators provided by `LiquidHaskell`, we can leverage the existing programming constructs (e.g. pattern-matching and recursion) to prove the correctness of the program and use the principle of programs-as-proofs. (known as Curry-Howard isomorphism)[Vaz+18; Wad15].

To illustrate the use of reflection, we define the (`++`) function in the refinement logic as follows:

```
{-@ LIQUID "--reflection" @-}
{-@ infixr ++ @-}
{-@ reflect ++ @-}

{-@ (++) :: xs:[a] -> ys:[a] -> { zs:[a] | len zs ==
    len xs + len ys } @-}
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

The `{-@ LIQUID "-reflection" @-}` annotation used to activate the reflection feature in `LiquidHaskell`. The `reflect` annotation, lift the (`++`) into the logic in three steps [Vaz+18]:

1. **Definition**: The annotation creates an *uninterpreted function* (`++`) `:: [a] -> [a] -> [a]` in the refinement logic. By uninterpreted, we mean that the logical (`++`) is not connected to the program function (`++`); in the logic, (`++`) only satisfies the *congruence axiom.*

2. **Reflection**: In this step, `LiquidHaskell` reflects the definition of (`++`) into its refinement type by automatically strengthening the defined function type for (`++`) to:

   ```
   (++) :: xs:[a] -> ys:[a]
   -> { zs:[a] | len zs == len xs + len ys
   && zs = xs ++ ys
   && ppProp xs ys }
   ```

   where `ppProp` is an alias for the following refinement, derived from the function's definition:

   ```
   ppProp xs ys = if xs == [] then ys
   else cons (head xs) (ppProp (tail xs) ys)
   ```

3. **Application**: With the reflected refinement type, each application of (`++`) in the code automatically unfolds the definition of (`++`) only *once* in the logic. In the next section, we will look into `PLE` that allows to unfold the definition of the function multiple times.

we can now reason about properties of (`++`) in the refinement logic that requires unfolding its definition, as opposed to treating it only as an uninterpreted function. In the following subsection, we will show how to use `LiquidHaskell` to verify that the (`++`) function is associative.

4.1.4. Refinement Abstraction

In addition to reflection and measures, `LiquidHaskell` provides powerful abstraction mechanisms for refinement types. Suppose we want to define a list where elements satisfy a relation with their neighbors. We can use `Refinement Abstraction` to define a new data type that abstracts over the refinement predicate [VSJ14]:

```
data PList a <p :: a -> a -> Bool> =
Nil
| Cons { phd :: a, ptl :: PList <p> a <p phd> }
```

Here, the type `a<p>` is a refinement type, equivalent to `{v:a | p v}`. The abstraction `<p :: a -> a -> Bool>` allows us to parameterize the list by a binary predicate `p`.

We can now use this abstraction to define a list of integers where each element is less than or equal to the next:

```
type SortedList = PList <{\x y -> x <= y}> Int
```

LiquidHaskell can verify that the following list is sorted:

```
{-@ ok :: SortedList @-}
ok :: PList Int
ok = Cons 1 (Cons 2 (Cons 3 Nil))
```
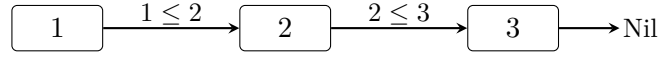

$$\text{SortedList} = \text{PList}\langle \lambda x\, y.\ x \leq y\rangle\ \text{Int}$$

Figure 4.2.: *Example* `SortedList`*: each neighbor pair respects* $\leq$*.*

4.1.5. Equational Proofs

`LiquidHaskell` allows formulation of proofs following the style of calculational or equational reasoning popularized in classic texts and implemented in proof assistants like Coq and Agda [Vaz+18]. It comes with the proof combinators library that allows to make the proofs more readable.

```
type Proof = ()
```

The alias `Proof` is defined as the unit type `()`, representing the result of a completed proof.

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

The (===) function proves equality. It takes `x:a` and `y:{a | y == x}`, returning a value refined as `{v:a | v == x && v == y}`.

```
data QED = QED

(***) :: a -> QED -> Proof
_ *** _ = ()
```

The `QED` data type is used to signal the end of a proof. The (***) operator takes a value and a `QED`, returning `Proof` (i.e., `()`).

```
{-@ (?) :: forall a b <pa :: a -> Bool, pb :: b ->
    Bool>. a<pa> -> b<pb> -> a<pa> @-}
(?) :: a -> b -> a
x ? _ = x
```

The (?) combinator preserves refinements. With type `a<pa> -> b<pb> -> a<pa>`, it maintains the refinement `pa` of the first argument, allowing properties to be carried across proof steps.

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

The `withProof` combinator enforces equality between input and output. With type `x:a -> b -> {v:a | v = x}`, it asserts that the returned value is exactly `x`, making it useful for chaining with (===) in equational reasoning.

In the following example, we show how to use these combinators to verify that the (++) function is associative:

```
{-@ assoc :: xs:[a] -> ys:[a] -> zs:[a]
-> { (xs ++ ys) ++ zs = xs ++ (ys ++ zs) } @-}
assoc :: [a] -> [a] -> [a] -> ()
assoc [] ys zs = ([] ++ ys) ++ zs
=== ys ++ zs
=== [] ++ (ys ++ zs)
*** QED

assoc (x : xs) ys zs = ((x : xs) ++ ys) ++ zs
===  x : (xs ++ ys) ++ zs
=== x : ((xs ++ ys) ++ zs) ? assoc xs ys zs
=== (x : xs) ++ (ys ++ zs)
*** QED
```

As you can see, we use proof by induction and in the induction step we use recursive call in the last step.

4.1.6. Totality

Ensuring *total* functions—functions defined for all possible inputs—is essential in program verification. In Haskell, however, many heap operations are naturally partial. Consider the definition of `findMin` for leftist heaps:

```
findMin :: Heap a -> a
findMin (Node x _ _) = x
```

This works for non-empty heaps but fails on `Empty`, resulting in a runtime exception. In GHC's Core, the missing case is made explicit through a call to `patError`:

```
findMin d = case d of
Node x _ _ -> x
Empty -> patError "findMin"
```

Here, `patError` is technically total, but it has an uninhabited type:

```
patError :: {v:String | false} -> a
```

`LiquidHaskell` eliminates such dead code by requiring refinements that ensure `findMin` is only applied to non-empty heaps. This can be achieved by defining a measure and a corresponding predicate:

```
measure isNonEmpty :: Heap a -> Prop
isNonEmpty (Node _ _ _) = true
isNonEmpty Empty = false

predicate NonEmp H = isNonEmpty H
```

Using this predicate, we refine the type of `findMin`:

```
findMin :: {h:Heap a | NonEmp h} -> a
```

Now `LiquidHaskell` verifies that the `Empty` branch is unreachable. When pattern matching on `d = Empty`, the environment is refined to:

```
Empty :: {v:Heap a | NonEmp v && v = Empty}
```

This condition is contradictory: `v = Empty` implies the heap is empty, while `NonEmp v` requires it to be non-empty. Since no such value exists, `LiquidHaskell` concludes that the call to `patError` is infeasible. Thus, `findMin` is total under its precondition. The burden shifts to clients: they must prove statically that any heap passed to `findMin` is non-empty.

The same reasoning applies to `deleteMin`, which is unsafe on empty heaps. By refining its type, we likewise guarantee totality:

```
deleteMin :: {h:Heap a | NonEmp h} -> Heap a
```

4.1.7. Termination

Another crucial aspect of `LiquidHaskell` is termination checking. Ensuring that all functions terminate is necessary for the soundness of the refinement type system, since non-termination can undermine logical consistency [Vaz24].

`LiquidHaskell` enforces this by associating a well-founded termination metric with a function's parameters and proving—via refinement checking—that the metric decreases with each recursive call [VSJ14].

For example, consider the factorial function:

```
fac :: n:Nat -> Nat / [n]
```

The termination metric `[n]` specifies that `n` must decrease in each recursive call.

Metrics need not be limited to single arguments. For instance, the `range` function uses an expression as the metric:

```
{-@ range :: lo:Int -> hi:Int -> [Int] / [hi - lo] @-}
range :: Int -> Int -> [Int]
range lo hi
| lo < hi = lo : range (lo + 1) hi
| otherwise = []
```

Although neither `lo` nor `hi` decreases alone, their difference `hi - lo` does.

Some functions require more general metrics. When multiple arguments may decrease, lexicographic ordering is used. Consider the greatest common divisor (GCD):

```
gcd :: Int -> Int -> Int
gcd 0 b = 0
gcd a 0 = a
gcd a b | a == b = a
| a > b = gcd (a - b) b
| a < b = gcd a (b - a)
```

Its refined type uses lexicographic ordering:

```
gcd :: a:Nat -> b:Nat -> Nat / [a, b]
```

The same technique applies to mutually recursive functions such as `isEven` and `isOdd` [VSJ14]:

```
isEven 0 = True
isEven n = isOdd (n - 1)

isOdd n = not (isEven n)
```

Here, `isEven` decreases on its argument, while `isOdd` does not. By specifying metrics as:

```
isEven :: n:Nat -> Bool / [n, 0]
isOdd :: n:Nat -> Bool / [n, 1]
```

`LiquidHaskell` checks that each recursive call reduces the lexicographic metric: e.g., `[n, 0]` is greater than `[n-1, 1]` in the call from `isEven` to `isOdd`.

Termination checking also extends to finite data structures by using their size. For example, the `bag` function can be checked using list length:

```
bag :: Ord a => xs:[a] -> Bag a / [len xs]
```

In practice, `LiquidHaskell` assumes by default that the first argument with a size measure (e.g., `len` for lists) decreases [VSJ14].

In some cases, Haskell functions are deliberately non-terminating. For such functions, termination checking can be disabled locally with the `lazy` annotation, or globally with the directive:

```
{-@ LIQUID "--no-termination" @-}
```

4.1.8. Proof by Logical Evaluation

In our proof in code 4.1.5, we primarily relied on straightforward unfoldings of the (`++`) function definition. However, `LiquidHaskell` provides a directive known as `Proof by Logical Evaluation` (PLE), which offers two significant advantages [Vaz+18]. First, PLE is guaranteed to construct an equational proof whenever one can be derived solely from unfoldings of function definitions, provided the user supplies necessary lemmas and induction hypotheses [Vaz+18]. Second, under practical conditions that are commonly satisfied, PLE is guaranteed to terminate [Vaz+18]. We can activate PLE by adding `{-@ LIQUID "--ple" @-}` annotation to automate the most parts of the proof for associativity of (`++`):

```
{-@ LIQUID "--ple" @-}
{-@ assoc :: xs:[a] -> ys:[a] -> zs:[a]
-> { (xs ++ ys) ++ zs = xs ++ (ys ++ zs) } @-}
assoc :: [a] -> [a] -> [a] -> ()
assoc [] ys zs = ()
assoc (x : xs) ys zs = assoc xs ys zs
```

In the above code, we only need to provide the base case and induction hypotheses, and `LiquidHaskell` will automatically unfold the definition of (`++`) to prove the associativity of the function. In the following chapter, we learn how to use `LiquidHaskell` to verify properties of functional data structures.

5. Verification in LiquidHaskell

This chapter details the formal verification of the leftist heap implementation using `LiquidHaskell`. We demonstrate how refinement types can be used to enforce complex structural and behavioral invariants, ensuring the correctness of our priority queue implementation.

5.1. Encoding Invariants

The cornerstone of the verification is the refined data type for the leftist heap itself. We encode the core invariants of the data structure directly into its type definition.

```
data LeftistHeap a = EmptyHeap
     | HeapNode { value :: a
        , left  :: LeftistHeapBound a value
        , right :: {v : LeftistHeapBound a value  |
           rrank v <= rrank left }
        , rank :: {r : Nat | r == 1 + rrank right}
        }
```

This refined definition enforces three key invariants:

1. **Heap Property:** The value at any node is the minimum in its subtree. This is captured by the `LeftistHeapBound a X` refinement type on the `left` and `right` children. `isLowerBound` is a recursively defined predicate that checks if a given value is less than or equal to all elements in a heap.

   ```
   {-@ type LeftistHeapBound a X = { h : LeftistHeap a
       | isLowerBound X h} @-}
   {-@ reflect isLowerBound @-}
   isLowerBound :: (Ord a) => a -> LeftistHeap a ->
       Bool
   isLowerBound _ EmptyHeap = True
   isLowerBound v (HeapNode x l r _) =
       v <= x && isLowerBound v l && isLowerBound v r
   ```

2. **Leftist Property:** For any node, the rank of its right child is less than or equal to the rank of its left child. This is expressed by `rrank v <= rrank left`. This property is what ensures that the right spine of the heap is short, leading to logarithmic time complexity for merge operations.

3. **Rank Property:** The rank of a node is defined as one plus the rank of its right child. This is specified by `rank :: {r : Nat | r == 1 + rrank right}`. The rank of an `EmptyHeap` is 0.

By embedding these invariants directly into the type, `LiquidHaskell`'s verifier will ensure that any function constructing or modifying a `LeftistHeap` respects them.

5.2. Use of Measures and Predicates

To express these invariants and reason about the behavior of our functions, we use several features of `LiquidHaskell`.

   Measures (see Section 4.1.3) are functions from Haskell's term-level to the refinement logic's domain. We define several measures:

- `size`: Computes the total number of nodes in the heap, useful for termination metrics (see Section 4.1.7).

- `rrank`: Returns the rank of a heap, which is crucial for the leftist property.

- `bag`: Converts the heap into a multiset (or bag) of its elements. This is invaluable for proving that operations like `heapMerge` do not lose or duplicate elements.

   Reflected Functions (see Section 4.1.3) allow us to use standard Haskell functions within the refinement logic. We use this for `isLowerBound`, `heapMerge`, and `makeHeapNode`. This allows us to reason about their behavior during verification.

   Predicates are aliases for refinement types that improve readability. For instance, we define predicates to describe the post-conditions of the `heapMerge` function:

```
predicate HeapMergeMin H1 H2 H =
   ((not (heapIsEmpty H1) && not (heapIsEmpty H2)) =>
   isLowerBound (min (heapFindMin H1) (heapFindMin H2))
      H )
predicate BagUnion H1 H2 H =
   (bag H == B.union (bag H1) (bag H2))
```

`HeapMergeMin` asserts that the resulting heap `H` respects the heap property relative to the minimum elements of the input heaps `H1` and `H2`. `BagUnion` asserts that the elements in the merged heap are the union of the elements from the input heaps.

5.3. Example Proofs

The most critical operation for a leftist heap is `heapMerge`. Its correctness is fundamental to the correctness of `insert` and `deleteMin`.

   The type signature for `heapMerge` specifies its behavior:

```
heapMerge :: h1 : LeftistHeap a
       -> h2: LeftistHeap a
       -> {h : LeftistHeap a | (HeapMergeMin h1 h2 h)
                                && (BagUnion h1 h2 h)}
       / [size h1, size h2, 0]
```

This signature guarantees that merging two valid leftist heaps results in a new valid leftist heap, that the heap property is maintained, and that the set of elements is preserved.

The implementation of `heapMerge` involves a recursive call. To help the SMT solver prove that the invariants hold through this recursion, we provide helper lemmas. For example, in the case where `x1 <= x2`, we merge the right child of the first heap (`r1`) with the second heap (`h2`). We must prove to `LiquidHaskell` that the root value `x1` is a lower bound for this newly merged heap.

```
... makeHeapNode x1 l1 ((heapMerge r1 h2)
    `withProof` lemma_merge_case1 x1 x2 r1 h2)
```

The lemma `lemma_merge_case1` provides this proof.

```
lemma_merge_case1 :: x1 : a
       -> x2 : { v : a |  x1  <= v}
       -> r1 : {h : LeftistHeap a | isLowerBound x1 h}
       -> h2 : {h : LeftistHeap a | not (heapIsEmpty h)
                                && isLowerBound x2 h}
       -> {isLowerBound x1 (heapMerge r1 h2)}
```

This lemma states that if `x1` is a lower bound for `r1` and `x1 <= x2` where `x2` is a lower bound for `h2`, then `x1` must also be a lower bound for the result of merging `r1` and `h2`.

With `heapMerge` verified, the correctness of other operations follows more easily:

- `heapInsert x h` is simply `heapMerge (HeapNode x EmptyHeap EmptyHeap 1) h`.

- `heapDeleteMin h` (or `splitMin`) is `heapMerge` of the left and right children of the root.

The properties proven for `heapMerge` are thus inherited by these functions.

5.4. Dealing with Termination and Recursion

`LiquidHaskell` must ensure that all recursive functions terminate. For `heapMerge`, we provide a termination metric:

```
/ [size h1, size h2, 0]
```

> Explain the metric why 0 is used

This specifies a lexicographically ordered tuple. `LiquidHaskell` verifies that for every recursive call within `heapMerge`, this metric decreases. In our case,

one of the heaps is replaced by its right child, which is strictly smaller, thus decreasing the total size and ensuring termination. The lemmas also have termination metrics that are checked by the verifier.

5.5. Challenges and Workarounds

The primary challenge in verifying the leftist heap was proving that the structural invariants are preserved by the recursive `heapMerge` function. The SMT solver, while powerful, cannot always deduce these properties on its own, especially when they rely on inductive reasoning over a recursive data structure.

The main "workaround" was to guide the solver by providing explicit proofs in the form of lemmas (`lemma_merge_case1` and `lemma_heapMerge_case2`). These lemmas essentially break down the proof of correctness into smaller, more manageable steps that the solver can verify. They act as explicit induction steps. For example, `lemma_merge_case1` proves a key property about the recursive call, which `LiquidHaskell` can then use to prove the property for the function as a whole. This pattern of using helper lemmas to discharge proof obligations is common in formal verification with refinement types.

List of Figures

A. My Code

```haskell
module PriorityQueue.LeftistHeap (PriorityQueue (..),
    MinView (..), lemma_merge_case1 , LeftistHeap (..),
    isLowerBound) where

import Language.Haskell.Liquid.ProofCombinators

import qualified Language.Haskell.Liquid.Bag as B
import PriorityQueue.Base
import Prelude hiding (min)

{-@ LIQUID "--reflection" @-}
{-@ LIQUID "--ple" @-}

data LeftistHeap a = EmptyHeap | HeapNode {value :: a,
    left :: (LeftistHeap a), right :: (LeftistHeap a),
    rank :: Int}
  deriving (Show , Eq)

{-@ data LeftistHeap a = EmptyHeap
      | HeapNode { value :: a
          , left   :: {h : LeftistHeap a | isLowerBound
              value h}
          , right  :: {v : LeftistHeap a  | isLowerBound
              value v
                    && rrank v <= rrank left }
          , rank   :: {r : Nat | r == 1 + rrank right}
        }
@-}

{-@ type Nat = {v:Int | 0 <= v} @-}

{-@ measure size @-}
{-@ size :: LeftistHeap a -> Nat @-}
size :: LeftistHeap a -> Int
size EmptyHeap = 0
size (HeapNode _ l r _) = 1 + size l + size r

{-@ reflect bag @-}
{-@ bag :: Ord a => LeftistHeap a -> Bag a @-}
bag :: (Ord a) => LeftistHeap a -> B.Bag a
bag EmptyHeap = B.empty
bag (HeapNode x l r _) = B.put x (B.union (bag l) (bag r))
```

```
{-@ measure rrank @-}
{-@ rrank :: LeftistHeap a -> Nat @-}
rrank :: LeftistHeap a -> Int
rrank EmptyHeap = 0
rrank h@(HeapNode _ _ _ r) = r

{-@ reflect min @-}
{-@ min :: x : a -> y : a -> {v : a |  v <= x && v <= y}
   @-}
min :: (Ord a) => a -> a -> a
min x y
  | x < y = x
  | otherwise = y

{-@ reflect makeHeapNode @-}
{-@ makeHeapNode :: x : a -> {h : LeftistHeap a |
   isLowerBound x h} -> {h : LeftistHeap a | isLowerBound
   x h} -> {h : LeftistHeap a | isLowerBound x h}@-}
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)

{-@ measure heapIsEmpty @-}
{-@ heapIsEmpty :: LeftistHeap a -> Bool @-}
heapIsEmpty :: (Ord a) => LeftistHeap a -> Bool
heapIsEmpty EmptyHeap = True
heapIsEmpty _ = False

{-@ measure heapFindMin @-}
{-@ heapFindMin :: h : { h : LeftistHeap a | not
   heapIsEmpty h } -> {v : a | isLowerBound v h} @-}
heapFindMin :: (Ord a) => LeftistHeap a -> a
heapFindMin (HeapNode x l r rank) = x

{-@ reflect isLowerBound @-}
{-@ isLowerBound :: Ord a => a -> h : LeftistHeap a ->
   Bool @-}
isLowerBound :: (Ord a) => a -> LeftistHeap a -> Bool
isLowerBound _ EmptyHeap = True
isLowerBound v (HeapNode x l r _) = v <= x &&
   isLowerBound v l && isLowerBound v r

{-@ predicate HeapMergeMin H1 H2 H = ((not (heapIsEmpty
   H1) && not (heapIsEmpty H2)) => isLowerBound (min
   (heapFindMin H1) (heapFindMin H2)) H ) @-}
{-@ predicate BagUnion H1 H2 H = (bag H == B.union (bag
   H1) (bag H2)) @-}

{-@ bogusUnion :: {v: ()| not BagUnion (HeapNode 1
   EmptyHeap EmptyHeap 1) (HeapNode 1 EmptyHeap EmptyHeap
```

```
   1) (HeapNode 1 EmptyHeap (HeapNode 2 EmptyHeap
      EmptyHeap 1) 1)} @-}
bogusUnion :: ()
bogusUnion = ()

-- heapMerge two heaps
{-@ reflect heapMerge @-}
{-@ heapMerge :: h1 : LeftistHeap a -> h2: LeftistHeap a
   -> { h : LeftistHeap a | (HeapMergeMin h1 h2 h) &&
   (BagUnion h1 h2 h) } / [size h1, size h2, 0] @-}
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)
     `withProof` lemma_merge_case1 x1 x2 r1 h2)
  | otherwise = makeHeapNode x2 l2 ((heapMerge h1 r2)
     `withProof` lemma_heapMerge_case2 x2 x1 r2 h1)

-- Transitivity lemma for isLowerBound
{-@ lemma_isLowerBound_transitive :: x : a -> y : {v : a
   | x <= v} -> h : {h : LeftistHeap a | isLowerBound y
   h} -> {isLowerBound x h} @-}
lemma_isLowerBound_transitive :: (Ord a) => a -> a ->
   LeftistHeap a -> Proof
lemma_isLowerBound_transitive x y EmptyHeap = ()
lemma_isLowerBound_transitive x y (HeapNode z l r _) =
   lemma_isLowerBound_transitive x y l &&&
   lemma_isLowerBound_transitive x y r *** QED

{-@ lemma_merge_case1 :: x1 : a  -> x2 : { v : a |  x1
   <= v}-> r1 : {h : LeftistHeap a | isLowerBound x1 h}
   -> h2 : {h : LeftistHeap a | not (heapIsEmpty h) &&
   isLowerBound x2 h}  -> {isLowerBound x1 (heapMerge r1
   h2)} / [size r1, size h2 , 1]@-}
lemma_merge_case1 :: (Ord a) => a -> a -> LeftistHeap a
   -> LeftistHeap a -> Proof
lemma_merge_case1 x1 x2 EmptyHeap h2 =
  isLowerBound x1 (heapMerge EmptyHeap h2)
    ? lemma_isLowerBound_transitive x1 x2 h2
    *** QED
lemma_merge_case1 x1 x2 r1@(HeapNode _ _ _ _)
   h2@(HeapNode _ _ _ _) =
  isLowerBound x1 (heapMerged)
    ? (lemma_isLowerBound_transitive x1 (min (heapFindMin
       r1) (heapFindMin h2)) (heapMerged))
    *** QED
 where
  heapMerged = heapMerge r1 h2
```

```
{-@ lemma_heapMerge_case2 :: x2 : a -> x1 : { v : a |  x2
    <= v} -> r1 : {h : LeftistHeap a | isLowerBound x2 h}
    -> h2 : {h : LeftistHeap a | not (heapIsEmpty h) &&
    isLowerBound x1 h}  -> {isLowerBound x2 (heapMerge h2
    r1)} / [size h2 , size r1, 1] @-}
lemma_heapMerge_case2 :: (Ord a) => a -> a -> LeftistHeap
    a -> LeftistHeap a -> Proof
lemma_heapMerge_case2 x2 x1 EmptyHeap h2 =
  isLowerBound x2 (heapMerge EmptyHeap h2)
    === isLowerBound x2 h2
    ? lemma_isLowerBound_transitive x2 x1 h2
    *** QED
lemma_heapMerge_case2 x2 x1 h1 EmptyHeap = ()
lemma_heapMerge_case2 x2 x1 r1@(HeapNode _ _ _ _)
    h2@(HeapNode _ _ _ _) =
  isLowerBound x2 (heapMerged)
    ? (lemma_isLowerBound_transitive x2 (min (heapFindMin
        h2) (heapFindMin r1)) (heapMerged))
    *** QED
 where
  heapMerged = heapMerge h2 r1

{-@ heapInsert :: Ord a => x : a -> h1 : LeftistHeap a ->
    {h : LeftistHeap a | not (heapIsEmpty h1) =>
    isLowerBound (min x (heapFindMin h1)) h} @-}
heapInsert :: (Ord a) => a -> LeftistHeap a ->
    LeftistHeap a
heapInsert x h = heapMerge (HeapNode x EmptyHeap
    EmptyHeap 1) h

heapDeleteMin :: (Ord a) => LeftistHeap a -> LeftistHeap a
heapDeleteMin EmptyHeap = EmptyHeap
heapDeleteMin h@(HeapNode _ l r _) = heapMerge l r

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

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

instance PriorityQueue LeftistHeap where
  empty = EmptyHeap
  findMin = safeHeapFindMin
  isEmpty = heapIsEmpty
  insert = heapInsert
  splitMin = heapSplit
```

Listing A.1: *Leftist Heap*

Bibliography

[ATC]   Andrew W. Appel, Andrew Tolmach, and Michael Clarkson. *Verified Functional Algorithms*. URL: https://softwarefoundations.cis.upenn.edu/vfa-current/index.html (visited on 09/16/2025).

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

[Dij59]   E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1 (1959), pp. 269–271. DOI: 10.1007/BF01386390.

[Hoa69]   C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[Huf52]   David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE*. Vol. 40. 9. 1952, pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[JSV20]   Ranjit Jhala, Eric Seidel, and Niki Vazou. *Programming With Refinement Types*. en. July 2020. URL: https://ucsd-progsys.github.io/liquidhaskell-tutorial/.

[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.

[NO79]   Greg Nelson and Derek C. Oppen. "Simplification by Cooperating Decision Procedures". In: *ACM Transactions on Programming Languages and Systems* 1.2 (1979), pp. 245–257. DOI: 10.1145/357073.357079.

[Oka98]   Chris Okasaki. *Purely Functional Data Structures*. Cambridge: Cambridge University Press, 1998. ISBN: 978-0-521-63124-2. DOI: 10.1017/CBO9780511530104. (Visited on 04/06/2025).

[Pri57]   R. C. Prim. "Shortest connection networks and some generalizations". In: *The Bell System Technical Journal* 36.6 (1957), pp. 1389–1401. DOI: 10.1002/j.1538-7305.1957.tb01515.x.

[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).

[Sho84]   Robert E. Shostak. "Deciding Combinations of Theories". In: *Journal of the ACM* 31.1 (1984), pp. 1–12. DOI: 10.1145/2422.322411.

[Vaz+14]   Niki Vazou, Eric Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. "Refinement Types For Haskell". In: *ACM SIGPLAN Notices* 49 (Aug. 19, 2014). ISSN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628161.

[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).

[Vaz24]   Niki Vazou. *Programming with Refinement Types Lecture*. Mar. 15, 2024. URL: https://nikivazou.github.io/lh-course.

[VSJ14]   Niki Vazou, Eric L. Seidel, and Ranjit Jhala. "LiquidHaskell: experience with refinement types in the real world". In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*. Haskell '14. New York, NY, USA: Association for Computing Machinery, Sept. 2014, pp. 39–51. ISBN: 978-1-4503-3041-1. DOI: 10.1145/2633357.2633366. URL: https://dl.acm.org/doi/10.1145/2633357.2633366 (visited on 11/11/2024).

[Wad15]   Philip Wadler. "Propositions as types". In: *Commun. ACM* 58.12 (Nov. 23, 2015), pp. 75–84. ISSN: 0001-0782. DOI: 10.1145/2699407. URL: https://dl.acm.org/doi/10.1145/2699407 (visited on 02/03/2025).