Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

Verified Functional Data Structures: Priority Queues in Liquid Haskell

Master's Thesis

by

MohammadMehran Shahidi

November 17, 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 17.11.2025

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 the 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, 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 two different priority queue variants, *Leftist Heaps* and *Binomial Heaps*, 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 Heaps* and *Binomial Heaps* 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.
- **Chapter 6** concludes the thesis by summarizing the findings, discussing limitations, and suggesting directions for future work.
- **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 such as Coq, Agda, and Dafny have been used to verify the correctness of similar data structures.

2.1 Functional Data Structures

Functional data structures are **immutable**, meaning their state cannot be changed after creation [Oka98]. This immutability guarantees **referential transparency**: an expression can always be replaced by its value without altering program behavior. By contrast, in-place modifications, common in imperative data structures, introduce side effects that complicate reasoning, especially in concurrent environments [Oka98].

To support updates without mutation, functional data structures rely on **persistence**: operations return a new version of the structure while preserving the old one. The seminal work in this area, Okasaki's *Purely Functional Data Structures*, shows how persistence can be implemented efficiently through **structural sharing**. For example, when inserting into a tree, only the nodes along the update path are recreated, while the rest of the structure is reused. This design minimizes memory overhead while preserving previous versions.

These properties—immutability, persistence, and structural sharing—not only improve modularity but also make functional data structures particularly well suited to formal verification. Invariants remain stable across all versions, allowing correctness properties to be reasoned about compositionally. This thesis builds on these principles, focusing on priority queues as a case study of verified functional data structures.

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 verification**, which uses logical reasoning to establish correctness. This is often

based on the principles of Hoare logic, where programs are annotated with preconditions and postconditions [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 Kawaguchi, 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 and Agda.

2.4 Verification in Interactive Theorem Provers (Coq and Agda)

Interactive theorem provers such as 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 [Vaz+18]. 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 tactics. 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) [Tea24]. 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 [Vaz+18].

Compared to `LiquidHaskell`, these systems provide stronger guarantees (as the entire logic is verified within a trusted kernel), but at the cost of automation [Vaz+18]. `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 [Vaz+14].

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.

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).

Typical operations include:

- **insert**: Add a new element with a given priority.
- **merge**: Combine two heaps into one.
- **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.
- **empty and isEmpty**: Create an empty heap and check if a heap is empty.

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

Listing 3.1: *Leftist Heap Implementation in Haskell*

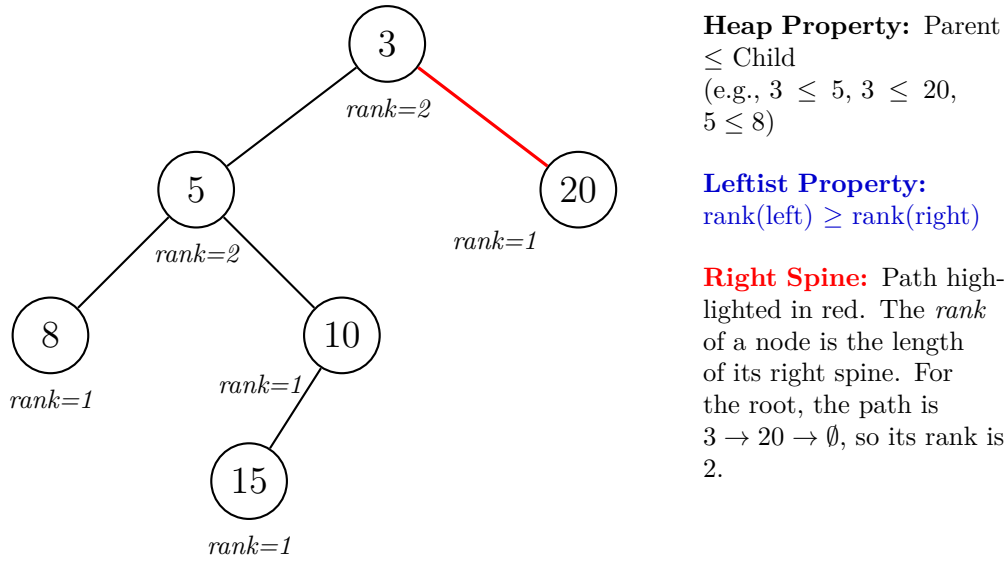
In this thesis, we focus on the *min-priority queue*, where elements with lower keys are considered higher priority. We will study and verify functional implementations of *Leftist Heaps* and some parts of *Binomial Heaps*.

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*



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

```
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`.

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

```
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.

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

3.3 Binomial Heap Implementation

For our verified implementation of binomial heaps, we follow the approach of *binary binomial heaps* as described by Hinze [Hin99]. This approach represents a heap as a list of trees, analogous to a binary number, which allows for a clean and efficient implementation of the merge operation.

Instead of classical binomial trees, this implementation uses *pennants*[SS90]. Pennants consist of a root node and a perfect binary tree.

```
data BinTree a
  = Empty
  | Bin { value :: a
        , left :: BinTree a
        , right :: BinTree a
        , height :: Int
        }

data Pennant a = P { root :: a
                   , pheight :: Int
                   , bin :: (BinTree a)
                   }
```

Listing 3.5: *Pennant Data Type*

A pennant's `pheight` corresponds to its height h. The `root` holds the minimum element, and `bin` is the internal `BinTree` structure containing the other $2^h - 1$ elements.

There are three invariants that pennants must satisfy:

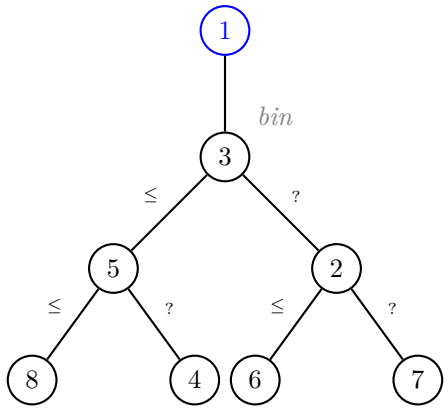
- **Minimum Property**: The root of the pennant is less than or equal to all other elements in the pennant.
- **Left-ordering Property**: For every node in the internal binary tree, the value of the left child is less than or equal to the value of the right child.
- **Perfect Binary Tree Property**: The internal binary tree is a perfect binary tree, meaning all internal nodes have two children and all leaves are at the same level.

The primary operation on pennants is `link`, which combines two pennants of equal height h into a single pennant of height $h+1$. The root of the resulting pennant is the minimum of the roots of the two original pennants.

```
link :: (Ord a) => Pennant a -> Pennant a -> Pennant a
link p1@(P r1 h t1) p2@(P r2 _ t2)
  | r1 <= r2  = P r1 (h + 1) (Bin r2 t2 t1 h)
  | otherwise = P r2 (h + 1) (Bin r1 t1 t2 h)
```

Listing 3.6: *Linking two Pennants*

Figure 3.2: *Visualization of Binomial Heap Properties*



Minimum Property: 1 is minimum element in the pennant

Left-ordering Property: Parent ≤ Left Child (e.g., $3 \leq 5$, $5 \leq 8$)

Perfect binary tree: Internal perfect tree of size $2^3 - 1$

A binomial heap is then represented as a list of "bits", where each bit can be either `Zero` or `One`. A `One` bit at position (`rank`) i signifies the presence of a pennant of height i.

```
data BinomialBit a =
       Zero { rank :: Int }
     | One  { rank :: Int, pennant :: Pennant a }

type BinomialHeap a = [BinomialBit a]
```

Listing 3.7: *Binomial Heap Representation*

Merging two binomial heaps is analogous to adding two binary numbers. The addition is performed by a ripple-carry adder that processes the lists of bits. For instance, adding two `One` bits of the same rank involves linking their pennants to form a carry bit of the next higher rank.

Following the ripple-carry adder approach, we define a `bHalfAdder` function that takes two bits and produces a sum bit and a carry bit.

```
bHalfAdder :: (Ord a) => BinomialBit a
                      -> BinomialBit a
                      -> (BinomialBit a, BinomialBit a)
bHalfAdder b1 b2 = (bSum b1 b2, bCarry b1 b2)
```

Listing 3.8: *Half Adder for Binomial Bits*

Additionally, we define a `bFullAdder` function that takes two bits and an incoming carry bit, producing a sum bit and an outgoing carry bit.

```
bFullAdder :: (Ord a) => BinomialBit a
                      -> BinomialBit a
                      -> BinomialBit a
                      -> (BinomialBit a, BinomialBit a)
bFullAdder b1 b2 cin =
  let (s, c1) = bHalfAdder b1 b2
      (s', c2) = bHalfAdder s cin
   in (s', bSum c1 c2)
```

Listing 3.9: *Full Adder for Binomial Bits*

The `bSum` and `bCarry` functions handle the logic for combining bits of the same rank according to the rules of binary addition.

```
bSum :: (Ord a) => BinomialBit a
                -> BinomialBit a
                -> BinomialBit a
bSum (Zero r) (Zero _) = Zero r
bSum (Zero r) (One _ p) = One r p
bSum (One _ p) (Zero r) = One r p
bSum (One r1 p1) (One r2 p2) = Zero r1
bCarry :: (Ord a) => BinomialBit a
                  -> BinomialBit a
                  -> BinomialBit a
bCarry (Zero r1) (Zero r2) = Zero (r1 + 1)
bCarry (Zero r1) (One _ p2) = Zero (r1 + 1)
bCarry (One _ p1) (Zero r2) = Zero (r2 + 1)
bCarry (One r1 p1) (One _ p2) = One (r1 + 1) (link p1 p2)
```

Listing 3.10: *bSum and bCarry Functions*

Finally, we can implement the addition of two binomial heaps using the full adder logic.

```
add :: (Ord a) => BinomialHeap a
               -> BinomialHeap a
               -> BinomialHeap a
add xs ys = addWithCarry xs ys (Zero 0)

addWithCarry :: (Ord a) => BinomialHeap a
                        -> BinomialHeap a
                        -> BinomialBit a
                        -> BinomialHeap a
addWithCarry Nil Nil c
  | c == (Zero 0) = Nil
  | otherwise = Cons c Nil
addWithCarry (Cons x xs) Nil (Zero r) = Cons x xs
addWithCarry (Cons x xs) Nil c@(One r _) =
  let z = Zero (rank x)
      (s, c') = bFullAdder x z c
   in
    Cons s (addWithCarry xs Nil c')
addWithCarry Nil (Cons y ys) c =
  let z = Zero (rank y)
      (s, c') = bFullAdder z y c
    in
      Cons s (addWithCarry Nil ys c')
addWithCarry (Cons x xs) (Cons y ys) c =
  let (s, c') = bFullAdder x y c
  in Cons s (addWithCarry xs ys c')
```

Listing 3.11: *Merging Two Binomial Heaps*

The main priority queue operations are implemented as follows:

- **merge**: Two heaps are merged by adding their corresponding lists of bits using the full adder logic. This takes $O(\log n)$ time.

```
merge :: (Ord a) => BinomialHeap a
                 -> BinomialHeap a
                 -> BinomialHeap a
merge h1 h2 = add h1 h2
```

Listing 3.12: *Merging Two Binomial Heaps*

- **insert**: A new element is treated as a pennant of rank 0. It is "added" to the heap, which may cause a series of carries. This is an $O(\log n)$ operation, with amortized $O(1)$ complexity.

```
insert :: (Ord a) => a
                  -> BinomialHeap a
                  -> BinomialHeap a
insert x heap = merge [One 0 (P x 0 (Empty))] heap
```

Listing 3.13: *Inserting an Element into Binomial Heap*

- **findMin**: This requires finding the minimum among the roots of all pennants in the heap. Since there are at most $\log n$ pennants, this takes $O(\log n)$ time. The implementation recursively scans through the list of bits, skipping `Zero` bits and comparing the roots of `One` bits to find the minimum.

```
findMin :: (Ord a) => BinomialHeap a -> Maybe a
findMin Nil = Nothing
findMin (Cons (Zero _) bs) = findMin bs
findMin (Cons (One _ (P r _ _)) Nil) = Just r
findMin (Cons (One _ (P r _ _)) bs) =
  case findMin bs of
    Nothing -> Just r
    Just r' -> Just (min r r')
```

Listing 3.14: *Finding Minimum in Binomial Heap*

- **extractMin**: This helper function scans through the heap to find the pennant with the minimum root value, removes it from the heap, and returns both the pennant and the remaining heap. It recursively compares roots and maintains heap structure by preserving `Zero` bits and selecting the pennant with the smaller root.

```
extractMin :: (Ord a) => BinomialHeap a
                      -> (Pennant a, BinomialHeap a)
extractMin (Cons (Zero r) bs) =
  let (p, bs') = extractMin bs
   in (p, Cons (Zero r) bs')
extractMin (Cons (One r p) Nil) = (p, Nil)
extractMin (Cons (One r p@(P m _ _)) bs) =
  case extractMin bs of
    (p'@(P m' _ _), bs')
      | m <= m' -> (p, Cons (Zero r) bs)
      | otherwise -> (p', Cons (One r p) bs')
```

Listing 3.15: *Extract Minimum Pennant from Binomial Heap*

- **splitMin**: This is the most involved operation. First, `extractMin` finds and removes the pennant with the minimum root from the list of pennants. Let's say it has rank k. After removing the minimum root from this pennant, the remaining perfect binary tree of height $k-1$ must be converted back into a binomial heap. This is accomplished by the `dismantle` function, which recursively splits the perfect binary tree into a list of pennants of ranks 0 to $k-1$. These pennants are then reversed and merged back with the remaining heap. The overall complexity is $O(\log n)$.

```
splitMin :: (Ord a) => BinomialHeap a
                    -> MinView BinomialHeap a
splitMin Nil = EmptyView
splitMin heap =
  let (minPennant, restHeap) = extractMin heap
      converted = case minPennant of
        P _ 0 _ -> restHeap
        P _ _ bt ->
          let dismantled = reverseToBinomialHeap
              (dismantle bt)
           in add restHeap dismantled
   in Min (root minPennant) converted
```

Listing 3.16: *Splitting Minimum from Binomial Heap*

To facilitate the dismantling process, we introduce an auxiliary data structure `ReversedBinomialHeap`, which is a list of bits with *decreasing* ranks (as opposed to the standard `BinomialHeap` where ranks increase).

```
data ReversedBinomialHeap a =
    RNil
  | RCons { rhd :: BinomialBit a
          , rtl :: ReversedBinomialHeap a
          }
```

Listing 3.17: *Reversed Binomial Heap*

The `dismantle` function takes a perfect binary tree and converts it into a reversed list of pennants with decreasing ranks. For a tree of height h, it produces pennants of ranks $h, h-1, \ldots, 0$. The function works by recursively processing the right spine of the tree: at each node, it creates a pennant from the current node's value and left subtree, then recursively dismantles the right subtree.

```
dismantle :: (Ord a) => BinTree a
                     -> ReversedBinomialHeap a
dismantle Empty = RNil
dismantle (Bin m l r h) = case r of
  Empty -> RCons (One h (P m h l)) RNil
  Bin _ _ _ hr ->
    let rest = dismantle r
     in RCons (One h (P m h l)) rest
```

Listing 3.18: *Dismantle Perfect Binary Tree*

The reversed heap is then converted back to a standard binomial heap using `reverseToBinomialHeap`, which reverses the list so that ranks increase from left to right. This conversion is necessary because the standard heap operations expect pennants ordered by increasing rank.

This representation of binomial heaps is both elegant and well-suited for formal verification, as we will discuss in Chapter 5. Rather than verifying every operation, we concentrate on the main operations, which serve as the fundamental building blocks from which other operations can be derived.

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 [JSV20a]. 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 [VSJ14]. 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` (§ 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 integer, a list of integers, 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 `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 inferred from the regular 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:

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

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:

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

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 formulas are 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].

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

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

4.2 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 the 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 §4.4, we will show how to define and use user-defined functions in the refinement logic.

4.3 Refined Data Types

In the previous examples, we saw how refinements of the input and output of functions allow us to make stronger arguments about our programs. We can take this further by refining the data types. We use the following example as an illustration, following [JSV20b]:

```
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.

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

4.4 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 an Algebraic Data Type (ADT), such as 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 into the logic functions with more than one argument, but the verification is no longer 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 is used to activate the reflection feature in `LiquidHaskell`. The `reflect` annotation lifts 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 unfolding the definition of the function multiple times.

We can now reason about properties of `(++)` in the refinement logic that require 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.

4.5 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> }
```

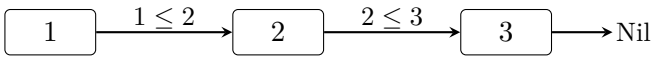
Here, the type `a<p>` is a refinement type, equivalent to `{v:a | p v}`. The abstraction `<p :: a -> a -> Bool>` allows us to parameterize the list by a binary predicate `p`.

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))
```



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

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

4.6 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 can be seen, we use proof by induction, and in the induction step we use the recursive call in the last step.

4.7 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
```

4.8 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`. Similarly, `[n, 1]` is greater than `[n, 0]` in the call from `isOdd` to `isEven`.

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" @-}
```

4.9 Proof by Logical Evaluation

In our proof in Code 4.6, 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 the `{-@ LIQUID "-ple" @-}` annotation to automate 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 verifies our priority queue implementations using LiquidHaskell. We first factor out the common logical infrastructure used by both heaps (§5.1), then present the leftist-heap development (§5.2), and finally *only* those parts of the binomial heap for which we currently have mechanized proofs (§5.3). Throughout, we refer back to Chapter 4 for the LiquidHaskell features we rely on (reflection, measures, PLE, and termination).

5.1 Shared Logical Infrastructure

We reuse LiquidHaskell features introduced in Chapter 4:

- **Measures and refined data types** (§4.4–4.3) to expose structural invariants to the refinement logic.
- **Reflection and equational reasoning** (§4.4, §4.6) for unfolding definitions during proofs.
- **Proof by Logical Evaluation (PLE)** (§4.9) to automate unfolding-driven proofs.
- **Termination metrics** (§4.8) using size/height and lexicographic tuples.

We also standardize the following predicate, reused by both heaps:

$$\mathsf{isLowerBound}\ v\ t\ \triangleq\ v \leq \text{every element of } t,$$

so that lower-bound obligations can be phrased uniformly for tree- and heap-shaped structures.

5.2 Verification of Leftist Heaps

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. In LiquidHaskell notation, the refined `LeftistHeap` type is defined as follows:

```
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.

To illustrate how LiquidHaskell enforces these invariants, consider a valid leftist heap:

```
validHeap :: LeftistHeap Int
validHeap = HeapNode 1
              (HeapNode 2 EmptyHeap EmptyHeap 1)
              EmptyHeap
              1
```

This heap passes verification because the heap property holds ($1 \leq 2$), the leftist property holds (`rrank right` $= 0 \leq 1 =$ `rrank left`), and the rank is correct ($1 = 1 + 0$).

In contrast, the following example fails LiquidHaskell verification:

```
invalidHeap :: LeftistHeap Int
invalidHeap = HeapNode 5
                (HeapNode 2 EmptyHeap EmptyHeap 1)
                EmptyHeap
                1
```

LiquidHaskell rejects this because the heap property is violated: the parent value 5 is greater than the left child value 2, contradicting the refinement `left :: LeftistHeapBound a value` which requires `isLowerBound 5 (HeapNode 2 ...)`.

Following the interface defined in Chapter 3.1, we refine the types of the heap operations to ensure they maintain the invariants of the leftist heap. Unfortunately, due to limitations in `LiquidHaskell`'s current support for type classes, we cannot directly define the invariants for the `PriorityQueue` type class. Instead, we provide refined type signatures for each operation individually. To express these invariants and reason about the behavior of our functions, we use several features of `LiquidHaskell`.

Measures (see §4.4) 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 §4.8).
- `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 §4.4) 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.

In the following sections, we present the refined type signatures and implementations of the key heap operations, along with explanations of how they maintain the invariants of the leftist heap.

5.2.1 Heap isEmpty

This function checks if the heap is empty. There is no invariant to maintain here, but we define a measure to help with other proofs.

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

5.2.2 Heap findMin

To retrieve the minimum element from a non-empty heap, we define `heapFindMin`. We restrict its input to non-empty heaps and specify that the returned value is a lower bound for the heap. As per the heap property, the minimum element is always at the root of the heap. This can be directly extracted from the `HeapNode`, and `LiquidHaskell` can verify that this value is indeed a lower bound for the entire heap.

```
{-@ heapFindMin :: h : {h : LeftistHeap a | not
    (heapIsEmpty h)}
```

```
      -> {v : a | isLowerBound v h} @-}
heapFindMin :: (Ord a) => LeftistHeap a -> a
heapFindMin (HeapNode x _ _ _) = x
```

5.2.3 Heap Merge

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.

To express these properties, we have defined two predicates:

```
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.

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 provide the proof for `LiquidHaskell` that the root value `x1` is a lower bound for this newly merged heap.

```
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)
```

The `makeHeapNode` requires that its first argument is a lower bound for both its left and right children.

```
makeHeapNode :: x : a
 -> {h : LeftistHeap a | isLowerBound x h}
 -> {h : LeftistHeap a | isLowerBound x h}
 -> {h : LeftistHeap a | isLowerBound x h}
```

In this context, `LiquidHaskell` can automatically infer that the root value (`x1` or `x2`) is a lower bound for the left child, since it is inherited from the parent heap. However, for the right child, which is obtained from a recursive call to `heapMerge`, the proof must be supplied explicitly. This proof obligation is discharged by auxiliary lemmas such as `lemma_merge_case1` and `lemma_merge_case2`.

The first lemma, `lemma_merge_case1`, handles the case where `x1 <= x2`. It states that if `x1` is a lower bound for `r1` and `x2` is a lower bound for `h2`, then `x1` is also a lower bound for the result of merging `r1` and `h2`.

```
lemma_merge_case1 :: x1 : a
  -> x2 : {a |  x1  <= x2}
  -> r1 : LeftistHeapBound a x1
  -> h2 : {LeftistHeapBound a x2 | not (heapIsEmpty h2)}
  -> {isLowerBound x1 (heapMerge r1 h2)}
  / [size r1, size h2, 1]
```

The proof proceeds by case analysis on the structure of `r1`.

```
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
```

In the base case, when `r1` is empty, the merge simply returns `h2`. Since `x1 <= x2` and `x2` is a lower bound for `h2`, it follows by transitivity that `x1` is also a lower bound for `h2`. In the inductive case, both heaps are non-empty. The result of `heapMerge r1 h2` again satisfies the lower-bound property, which is established through the transitive lemma below.

Transitivity of Lower Bounds

The lemma `lemma_isLowerBound_transitive` expresses the fundamental transitivity of the lower-bound relation across heaps. It is used repeatedly throughout the verification of heap merge.

```
{-@ 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
```

This lemma formalizes the intuitive notion that if `x <= y` and `y` is a lower bound for all elements of a heap `h`, then `x` must also be a lower bound for `h`. It is a small but essential proof component that supports most of the recursive heap reasoning.

Symmetric Case

A symmetric argument applies to the case where `x1 > x2`. The second lemma, `lemma_merge_case2`, follows the same structure as `lemma_merge_case1` but exchanges the roles of the input heaps and their bounds.

```
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]
```

The reason we cannot simply reuse `lemma_merge_case1` is that the order of the arguments to `heapMerge` is swapped in this case. Additionally, the termination metric must reflect which arguments are reduced in the recursive call. We discuss this in the next section.

Dealing with Termination and Recursion

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

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

This specifies a lexicographically ordered tuple. `LiquidHaskell` verifies that for every recursive call within `heapMerge`, either the size of the first argument or the size of the second argument decreases. In our case, one of the heaps is replaced by its right child, which is strictly smaller, thereby decreasing the total size and ensuring termination. The trailing `0` acts as a tie-breaker: it allows us to extend the tuple later when reasoning about mutually recursive functions.

The lemma `lemma_merge_case1` is mutually recursive with `heapMerge`, which requires us to make explicit which arguments of `heapMerge` are reduced in the body of the function (for instance, `r1` compared to `h1`). To capture this, we use the following termination metric for `lemma_merge_case1`:

```
/ [size r1, size h2, 1]
```

Here the final `1` ensures that

$$[size\ r1, size\ h2, 0] \ < \ [size\ r1, size\ h2, 1]$$

in lexicographic order. This indicates that when `lemma_merge_case1` calls `heapMerge`, it is making progress towards termination. In the same way, other supporting lemmas are also assigned termination metrics, which are automatically checked by the verifier.

Together, these lemmas ensure that `heapMerge` preserves the heap property and terminates correctly for all valid inputs. They form the core of the mechanized proof that merging two leftist heaps yields a well-formed, correctly ordered, and element-preserving result.

5.2.4 Heap Insert

Heap insertion is implemented using heap merging. The refined type signature for `heapInsert` specifies that inserting an element into a heap produces a new heap where the inserted element is a lower bound for the resulting heap if it is smaller than the minimum of the original heap (when non-empty), and that the multiset of elements is updated accordingly.

```
heapInsert :: x : a
  -> h1 : LeftistHeap a
  -> {h : LeftistHeap a |
        not (heapIsEmpty h1)
          => isLowerBound (min x (heapFindMin h1)) h
        && bag h = B.put x (bag h1) }
```

`LiquidHaskell` can automatically verify that the properties hold, given the correctness of `heapMerge`. Therefore, no additional lemmas are required here.

5.2.5 Heap SplitMin

The `SplitMin` operation decomposes a heap into its minimum element and the remaining heap. If the heap is empty, the result is an empty view. This operation can be expressed elegantly in LiquidHaskell using a single refinement predicate that encodes both cases, the empty and non-empty heap, guarded by logical conditions.

```
{-@ predicate SplitOK H S =
  (heapIsEmpty H => isEmptyView S)
    && (not (heapIsEmpty H) => not (isEmptyView S)
      && getMinValue S == heapFindMin H
      && bag H == B.put (getMinValue S) (bag (getRestHeap
          S)))
@-}

{-@ heapSplit :: (Ord a)
              => h:LeftistHeap a
              -> { s:MinView LeftistHeap a | SplitOK h s
                 } @-}
```

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

The refinement predicate `SplitOK` expresses the relationship between the input heap `H` and the result `S` of the `heapSplit` operation in a case-distinguishing manner:

- **Empty heap case.** When `heapIsEmpty H` holds, the resulting view `S` must be empty, i.e., `isEmptyView S` is true. This ensures that the field selectors `getMinValue` and `getRestHeap` are never applied to an empty structure, maintaining totality of the specification.
- **Non-empty heap case.** When `H` is non-empty, the result must be of the form `Min` with fields `minValue` and `restHeap`. In this branch, the following properties must hold:
  1. The minimum value of the view matches the minimum of the input heap:
     `getMinValue S = heapFindMin H.`
  2. The multiset of elements is preserved:

     `bag H = B.put (getMinValue S) (bag (getRestHeap S)).`

The implementation follows the specification precisely. For an empty heap, `heapSplit` simply returns `EmptyView`. For a non-empty heap of the form `HeapNode x l r _`, the result is `Min x (heapMerge l r)`. The correctness of this implementation relies on two key facts:

1. The root element `x` of a non-empty leftist heap is its minimum, establishing the equality `getMinValue S == heapFindMin H`.
2. The `heapMerge` operation preserves the multiset of elements, ensuring the bag equality in the refinement holds.

Thus, the `heapSplit` function correctly decomposes any leftist heap into its minimal element and the remainder, while preserving both structural and content invariants as captured by `SplitOK`.

5.3 Verification of Binomial Heaps

Building upon the verification techniques established for leftist heaps (§5.2), this section presents the verification of binomial heaps. The binomial heap structure, described in Chapter 3 (§3.3), is more intricate than leftist heaps, requiring a compositional approach across three data structure layers. We verify: (i) the *pennant* and its internal tree invariants, (ii) the *link* operation for merging two pennants, (iii) rank-correctness of bit-level addition (`bSum`,

`bCarry`, `bHalfAdder`, `bFullAdder`), and (iv) the termination and rank consistency of `addWithCarry`. As with leftist heaps, we leverage the shared logical infrastructure (§5.1), particularly the `isLowerBound` predicate for expressing heap properties.

5.3.1 Refined Data Structures

Following the pattern established for leftist heaps (§5.2), we encode the binomial heap's structural invariants directly into refined data types. The binomial heap implementation (Chapter 3, §3.3) is built from three nested structures: binary trees, pennants, and bit-level heaps. Each layer is verified through appropriate refinements.

Binary Trees with Left-Ordering

The foundation is a perfect binary tree with a left-ordering property rather than a full heap property. We reuse the `isLowerBound` predicate from the shared infrastructure (§5.1), but apply it only to the left child. The refined `BinTree` type ensures both left-ordering and perfect balance:

```
type BinTreeBound a X = {b : BinTree a | isLowerBound X b}
data BinTree a = Empty
  | Bin { value  :: a
        , left   :: BinTreeBound a value
        , right  :: BinTreeHeight a (bheight left)
        , height :: {h : Nat | h == 1 + bheight right}
        }
```

`bheight` is a measure that computes the height of a `BinTree`:

```
{-@ measure bheight @-}
{-@ bheight :: BinTree a -> {v: Int | v >= -1} @-}
bheight :: BinTree a -> Int
bheight Empty = -1
bheight (Bin _ _ _ h) = h
```

Pennants

A pennant wraps a binary tree with its root element. The refined `Pennant` type enforces the **Minimum Property** from Chapter 3—the root is less than or equal to all elements in the internal tree:

```
data Pennant a
  = P { root    :: a
      , pheight :: Nat
      , bin     :: {b : BinTreeBound a root | bheight b +
         1 = pheight}
      }
```

The refinement `bin :: {b : BinTreeBound a root | ...}` ensures `isLowerBound root bin`, capturing the **Minimum Property**. The height

constraint `bheight b + 1 = pheight` connects the pennant's advertised height to its internal tree structure.

Together with the `BinTree` definition, the three invariants from Chapter 3 are enforced:

1. **Minimum Property:** The `Pennant` refinement `bin :: BinTreeBound a root` ensures that the pennant's root is a lower bound for all elements in its internal tree (`isLowerBound root bin`).

2. **Left-ordering Property:** Within the internal `BinTree`, the refinement `left :: BinTreeBound a value` ensures that each node's value is a lower bound for its left subtree (`isLowerBound value left`). This means for every node, the left child's value is less than or equal to the right child's value. Note that this is *weaker* than the full heap property used in leftist heaps (§5.2)—we only enforce ordering between the parent and left child, not both children.

3. **Perfect Binary Tree Property:** The `BinTree` refinement `right :: BinTreeHeight a (bheight left)` enforces that both subtrees have identical height. Combined with the height correctness constraint `height :: {h:Nat | h == 1 + bheight right}`, this ensures the tree is perfectly balanced: all internal nodes have two subtrees of equal height, and all leaves are at the same level.

5.3.2 Pennant Operations

Singleton Pennant

Creating a singleton pennant, which represents a single element, is a simple operation. The refined type ensures that it correctly forms a pennant of size 0.

```
{-@ singleton :: Ord a => a -> {v : Pennant a | pheight v
    == 0} @-}
singleton :: (Ord a) => a -> Pennant a
singleton x = P x 0 Empty
```

The implementation creates a pennant with the given element `x` as the root, a size of 0, and an `Empty` tree as its `bin`. LiquidHaskell verifies this is valid because `bheight Empty` is 0, and `isLowerBound x Empty` is trivially true.

Link Pennants

The core operation on pennants is merging two pennants of the same height h into one of height $h+1$. The refined signature guarantees the height increase, ensures that all elements of the merging pennants exist, and, more importantly, that the resulting pennant preserves the invariants.

```
{-@ reflect link @-}
{-@ link :: Ord a =>
      t1 : Pennant a
```

```
    -> t2 : {Pennant a | pheight t2 == pheight t1}
    -> {v : Pennant a | pheight v == pheight t1 + 1 &&
       BagUnion t1 t2 v} @-}
link :: (Ord a) => Pennant a -> Pennant a -> Pennant a
link (P x1 h1 t1) (P x2 h2 t2)
  | x1 <= x2
    = P x1 (h1 + 1) (Bin x2 t2 t1 (h1))
       `withProof` lemma_isLowerBound_transitive x1 x2 t2
  | otherwise
    = P x2 (h1 + 1) (Bin x1 t1 t2 (h1))
       `withProof` lemma_isLowerBound_transitive x2 x1 t1
```

The implementation chooses the smaller of the two roots (`x1` or `x2`) as the new root. The other pennant's root and tree are used to form a new `BinTree` node that becomes the `bin` of the resulting pennant.

Verification of this function hinges on proving that the new pennant satisfies the *Minimum Property*. For example, in the `x1 <= x2` case, the new root is `x1`. The new internal tree is `Bin x2 t2 t1 ...`. To prove `isLowerBound x1 (Bin x2 t2 t1 ...)`, we must show:

1. `x1 <= x2`, which is given by the conditional.
2. `isLowerBound x1 t1`, which is true because `t1` was the `bin` of the pennant rooted at `x1`.
3. `isLowerBound x1 t2`, which requires a proof. We know `x1 <= x2` and `isLowerBound x2 t2` (from the input pennant `t2`). The property follows from transitivity.

This final step is not obvious to the SMT solver, so we provide an explicit proof using the `lemma_isLowerBound_transitive`, which is identical in function to the one used for Leftist Heaps (§5.2). This lemma proves that if `x <= y` and `y` is a lower bound for a tree, then `x` is also a lower bound.

5.3.3 Binomial Heap Representation and Bit-level Operations

Following Chapter 3, a binomial heap is represented as a list of bits, where each bit at position i either contains `Zero` or `One` pennant of height i. This representation is analogous to the binary representation of a number, enabling efficient merge operations through bit-level addition.

Refined Bit Structure

Each bit is tagged with its rank (order) and refined to ensure rank-height consistency:

```
data BinomialBit a
  = Zero { zorder :: Nat }
  | One  { oorder :: Nat
         , pennant :: {p : Pennant a | pheight p ==
           oorder}
         }
```

The refinement `pennant :: {p : Pennant a | pheight p == oorder}` ensures that a `One` bit at rank i contains a pennant of height i.

A `BinomialHeap` is a list of these bits with ranks strictly increasing by one at each step. Rather than using LiquidHaskell's parameterized `PList` type, we define a custom refined data type that directly encodes the rank-incrementing invariant:

```
data BinomialHeap [len] a
  = Nil
  | Cons { hd :: BinomialBit a
         , tl :: {bs : BinomialHeap a |
                  not (heapIsEmpty bs) =>
                    rank (bhead bs) = rank hd + 1}
         }
```

This refinement maintains the binomial heap's structural invariant: if the tail is non-empty, its first bit's rank must be exactly one more than the current bit's rank. The measure `len` provides a termination metric for recursive functions over heaps.

To illustrate the rank-incrementing invariant, consider a valid binomial heap:

```
validBH :: BinomialHeap Int
validBH = Cons (Zero 0)
                       (Cons (One 1 (P 3 1 (Bin 5 Empty
                          Empty 0)))
                         (Cons (Zero 2) Nil))
```

This heap satisfies the invariant: the bits have ranks 0, 1, and 2, increasing by exactly one at each step. The `One` bit at rank 1 contains a pennant of height 1, satisfying `pheight p == oorder`.

In contrast, the following example fails verification:

```
invalidBH :: BinomialHeap Int
invalidBH =
  Cons (Zero 0)
    (Cons (One 2 (P 3 2 (Bin 5 (Bin 7 Empty Empty 0)
                               (Bin 6 Empty Empty 0) 1)))
      Nil)
```

LiquidHaskell rejects this because the rank-incrementing invariant is violated: the first bit has rank 0, but the second bit has rank 2, contradicting the refinement `rank (bhead bs) = rank hd + 1` which requires the rank to increase by exactly one.

We define `heapIsEmpty` and `bhead` as measures to check emptiness and access the first bit:

```
{-@ measure heapIsEmpty @-}
heapIsEmpty :: BinomialHeap a -> Bool
heapIsEmpty Nil = True
heapIsEmpty _   = False

{-@ measure bhead @-}
```

```
{-@ bhead :: {b : BinomialHeap a | not (heapIsEmpty b)}
          -> BinomialBit a @-}
bhead :: BinomialHeap a -> BinomialBit a
bhead (Cons a _) = a
```

Additionally, the measure `bRank` extracts the rank of a heap's first bit (or 0 for empty heaps), which is crucial for specifying preconditions on operations like `bAdd` that require heaps starting at rank 0:

```
{-@ measure bRank @-}
{-@ bRank :: BinomialHeap a -> Nat @-}
bRank :: BinomialHeap a -> Int
bRank Nil = 0
bRank (Cons b bs) = rank b
```

Bit Arithmetic Operations

Following the ripple-carry adder approach from Chapter 3, the merge operation is implemented through bit-level addition. The `bSum` and `bCarry` functions form the basis of this arithmetic, with refined types guaranteeing rank-correctness:

```
bSum :: b1 : BinomialBit a
  -> b2 : {BinomialBit a | rank b2 == rank b1}
  -> {b : BinomialBit a | rank b == rank b1}

bCarry :: Ord a =>
     b1 : {BinomialBit a | rank b1 >= 0}
  -> b2 : {BinomialBit a | rank b2 == rank b1}
  -> {b : BinomialBit a | rank b == rank b1 + 1} @-}
```

The `bSum` function produces a bit of the same rank as its inputs, while `bCarry` produces a carry bit of the next higher rank. Both functions are reflected into the refinement logic, enabling LiquidHaskell to reason about their behavior during verification.

These primitives are composed into half-adders and full-adders:

```
bHalfAdder :: b1 : BinomialBit a
  -> b2 : {BinomialBit a | rank b2 == rank b1}
  -> ({s : BinomialBit a | rank s == rank b1},
     {c : BinomialBit a | rank c == rank b1 + 1})

bFullAdder :: b1 : BinomialBit a
  -> b2 : {BinomialBit a | rank b2 == rank b1}
  -> c  : {BinomialBit a | rank c == rank b1}
  -> ({s : BinomialBit a | rank s == rank b1},
     {co : BinomialBit a | rank co == rank b1 + 1})
```

The half-adder takes two bits at rank r and produces a sum at rank r and a carry at rank $r + 1$. The full-adder additionally accepts a carry-in bit at rank r, producing the same outputs. These rank-preserving refinement types ensure that the composed operations maintain heap rank consistency without requiring explicit rank-preservation lemmas—LiquidHaskell's PLE (§4.9) automatically unfolds the reflected definitions to verify correctness.

5.3.4 Heap Merging with `addWithCarry`

The function `addWithCarry` implements the ripple-carry addition recursively over the lists of bits. Its refined type ensures that the carry bit passed between recursive calls has the expected rank, and that the output maintains heap rank consistency.

The top-level `bAdd` function initiates the merge by calling `addWithCarry` with an initial carry of `Zero 0`:

```
bAdd :: h1 : {BinomialHeap a | bRank h1 == 0}
     -> h2 : {BinomialHeap a | bRank h2 == 0}
     -> BinomialHeap a
```

The precondition requires both input heaps to start at rank 0, which is maintained by all priority queue operations through the `padWithZeros` helper.

The `addWithCarry` function has a more complex refinement type:

```
addWithCarry :: h1 : BinomialHeap a
             -> h2 : {BinomialHeap a |
                       (bRank h2 == bRank h1 ||
                          heapIsEmpty h1)
                       || heapIsEmpty h2}
             -> carry : {BinomialBit a |
                          ((not (heapIsEmpty h1)) => rank
                             carry == bRank h1)
                          && ((not (heapIsEmpty h2)) =>
                             rank carry == bRank h2)}
             -> {b : BinomialHeap a |
                 (not (heapIsEmpty b)) => rank (bhead b)
                    == rank carry}
             / [len h1, len h2]
```

The key properties verified by LiquidHaskell are:

- **Rank Alignment:** The precondition ensures that when both heaps are non-empty, they have the same starting rank. Additionally, the carry bit's rank must match the starting rank of any non-empty input heap.
- **Output Rank:** The postcondition guarantees that if the result is non-empty, its first bit has the same rank as the input carry. This ensures that the output heap's structure is consistent with the input.
- **Termination:** The lexicographic metric `/ [len h1, len h2]` proves termination, as each recursive call consumes at least one bit from one of the input heaps.

The verification relies on the refined types of `bFullAdder`, which guarantee that the new sum bit has the correct rank for prepending to the recursive result, and that the new carry bit has rank $r + 1$ for matching the next bits in the lists. The `BinomialHeap` data type refinement ensures that consecutive bits differ by exactly one in rank, so when processing bit at rank r, the tail's first bit (if it exists) has rank $r + 1$, matching the new carry's rank.

LiquidHaskell's PLE automatically unfolds the reflected definitions of `bSum`, `bCarry`, and `bFullAdder` to verify these rank invariants hold throughout the recursion, eliminating the need for explicit rank-preservation lemmas.

5.3.5 Finding and Extracting Minimum Elements

The binomial heap's list-of-bits representation requires searching across multiple pennants to find the global minimum. Unlike leftist heaps where the minimum is always at the root, binomial heaps must examine the root of each non-empty pennant (each `One` bit) in the heap.

Checking for Empty Heaps

A binomial heap is effectively empty when it contains only `Zero` bits. The `hasOnlyZeros` predicate captures this property:

```
{-@ reflect hasOnlyZeros @-}
{-@ hasOnlyZeros :: BinomialHeap a -> Bool @-}
hasOnlyZeros :: BinomialHeap a -> Bool
hasOnlyZeros Nil = True
hasOnlyZeros (Cons (Zero _) bs) = hasOnlyZeros bs
hasOnlyZeros (Cons (One _ _) _) = False
```

This predicate is reflected into the logic and used as a precondition for operations that require a non-empty heap.

Finding the Minimum Root

The `minRootInHeap` function searches through the list of bits to find the minimum among all pennant roots. Its refined type ensures it is only called on heaps containing at least one `One` bit:

```
{-@ reflect minRootInHeap @-}
{-@ minRootInHeap :: Ord a =>
      h : {BinomialHeap a | not (hasOnlyZeros h)}
   -> a @-}
minRootInHeap :: (Ord a) => BinomialHeap a -> a
minRootInHeap (Cons (Zero _) bs) = minRootInHeap bs
minRootInHeap (Cons (One _ (P r _ _)) Nil) = r
minRootInHeap (Cons (One _ (P r _ _)) bs@(Cons _ _)) =
  if hasOnlyZeros bs
    then r
    else min r (minRootInHeap bs)
```

The function recursively skips `Zero` bits and compares roots of `One` bits using the `min` function (see §5.1). The precondition `not (hasOnlyZeros h)` ensures the recursion encounters at least one `One` bit before reaching `Nil`.

Extracting the Minimum Pennant

The `extractMin` function locates and removes the pennant with the minimum root value, returning both the pennant and the remaining heap. Its refined type guarantees that the extracted pennant's root is indeed the minimum:

```
extractMin :: (Ord a)
   => {h : BinomialHeap a | not (hasOnlyZeros h)}
   -> ({p : Pennant a | minRootInHeap h == root p},
       {h' : BinomialHeap a |
        (not (heapIsEmpty h') => rank (bhead h') == rank
           (bhead h))
        && (bRank h == 0 => (heapIsEmpty h' || bRank h'
           == 0))})
```

The implementation traverses the bit list, comparing roots and recursively extracting from the tail when necessary. The refinement ensures:

- The extracted pennant's root equals `minRootInHeap h`.
- The remaining heap maintains rank consistency (if non-empty, its first bit has the same rank).
- If the input heap starts at rank 0, the output heap is either empty or also starts at rank 0.

The verification relies on comparing roots using the `min` function and the transitivity of the $\leq$ relation, which PLE can establish automatically.

5.3.6 Dismantling Pennants and Reversing Heaps

The `splitMin` operation requires converting the internal tree of the minimum pennant back into a binomial heap. This involves two key operations: dismantling a perfect binary tree into a reversed bit-list, and reversing that list back to standard form.

Reversed Binomial Heaps

To efficiently convert a binary tree to a binomial heap, we use an auxiliary data structure with *decreasing* ranks:

```
data ReversedBinomialHeap [rlen] a =
       RNil
     | RCons { rhd :: BinomialBit a
             , rtl :: {bs : ReversedBinomialHeap a |
                       not (isRNil bs) =>
                         rank (rbhead bs) = rank rhd - 1}
             }
```

This structure is the mirror image of `BinomialHeap`: consecutive bits differ by -1 in rank rather than $+1$. The refinement ensures that ranks decrease strictly by one at each step.

Dismantling a Binary Tree

The `dismantle` function converts a perfect binary tree into a reversed bit-list. At each level of the tree, it creates a `One` bit containing a pennant formed from the current node's value and left subtree:

```
{-@ dismantle :: Ord a => t : BinTree a
    -> {rh : ReversedBinomialHeap a | ValidDismantle t rh}
       @-}
dismantle :: (Ord a) => BinTree a -> ReversedBinomialHeap
    a
dismantle Empty = RNil
dismantle (Bin m l r h) =
  case r of
    Empty -> RCons (One h (P m h l)) RNil
    Bin _ _ _ hr ->
      let rest = dismantle r
          result = RCons (One h (P m h l)) rest
       in result `withProof` lemma_rlast_preserved (One h
          (P m h l)) rest
```

The refinement predicate `ValidDismantle` ensures structural consistency:

- If the tree has non-negative height, the result is non-empty.
- The first bit's rank equals the tree's height.
- The last bit's rank equals 0 (ensuring compatibility with `reverseToBinomialHeap`).

The proof uses `lemma_rlast_preserved`, which establishes that prepending a bit to a reversed heap preserves the property that the last bit has rank 0. This lemma is proven by structural induction on the reversed heap.

Reversing to Standard Form

The `reverseToBinomialHeap` function converts a reversed bit-list back to a standard binomial heap using an accumulator pattern:

```
{-@ reverseToBinomialHeap ::
      rh : {ReversedBinomialHeap a | ReversedEndsAtZero
         rh}
    -> {h : BinomialHeap a | ValidReversed rh h}
    / [rlen rh] @-}
reverseToBinomialHeap :: ReversedBinomialHeap a ->
    BinomialHeap a
reverseToBinomialHeap RNil = Nil
reverseToBinomialHeap rh@(RCons b bs) =
  case bs of
    RNil -> Cons b Nil
    RCons _ _ -> reverseAcc bs (Cons b Nil)
                 `withProof` lemma_rlast_tail rh
```

The precondition `ReversedEndsAtZero` requires that the last bit has rank 0, ensuring the resulting heap starts at rank 0 (as required by `ValidReversed`). The termination metric `/ [rlen rh]` proves the function terminates by consuming the reversed list.

5.3.7 The `splitMin` Operation

The `splitMin` operation combines all the previously discussed components to extract the minimum element and its remaining heap:

```
{-@ splitMin ::
      h : {BinomialHeap a | bRank h == 0}
   -> MinView BinomialHeap a @-}
splitMin :: (Ord a) => BinomialHeap a -> MinView
   BinomialHeap a
splitMin heap =
  if hasOnlyZeros heap
    then EmptyView
    else case extractMin heap of
      (minPennant, restHeap) ->
        let converted = case minPennant of
              P _ 0 _ -> restHeap
              P _ _ _ -> case restHeap of
                Nil -> reverseToBinomialHeap (dismantle
                    (bin minPennant))
                Cons _ _ -> bAdd restHeap
                              (reverseToBinomialHeap
                                 (dismantle (bin
                                 minPennant)))
         in Min (root minPennant) converted
```

The operation proceeds as follows:

1. Check if the heap is empty (only zeros). If so, return `EmptyView`.
2. Extract the minimum pennant and the remaining heap using `extractMin`.
3. If the pennant has height 0 (singleton), return it with the remaining heap.
4. Otherwise, dismantle the pennant's internal tree, reverse it to standard form, and merge it with the remaining heap using `bAdd`.

The refined return type `MinView BinomialHeap a` is defined in the shared infrastructure (§5.1). The precondition `bRank h == 0` ensures the input heap starts at rank 0, which is maintained throughout by `extractMin`, `reverseToBinomialHeap`, and `bAdd`.

5.3.8 Priority Queue Operations

The priority queue interface operations are implemented using the verified core operations, with additional rank-normalization through the `padWithZeros` helper:

```
{-@ heapEmpty :: {h : BinomialHeap a | bRank h == 0} @-}
heapEmpty :: BinomialHeap a
heapEmpty = Nil

{-@ heapInsert :: Ord a => a
```

```
    -> h : BinomialHeap a
    -> {h' : BinomialHeap a | bRank h' == 0} @-}
heapInsert :: (Ord a) => a -> BinomialHeap a ->
    BinomialHeap a
heapInsert x h = bAdd (Cons (One 0 (singleton x)) Nil)
    (padWithZeros h)

{-@ heapMerge :: Ord a =>
      h1 : BinomialHeap a
    -> h2 : BinomialHeap a
    -> {h' : BinomialHeap a | bRank h' == 0} @-}
heapMerge :: (Ord a) => BinomialHeap a -> BinomialHeap a
    -> BinomialHeap a
heapMerge h1 h2 = bAdd (padWithZeros h1) (padWithZeros h2)
```

The `padWithZeros` function ensures heaps start at rank 0 by prepending `Zero` bits as needed. While marked with `assume` in the current implementation, its correctness follows from prepending bits with strictly increasing ranks starting from 0.

The `findMin` operation uses `minRootInHeap` when the heap is non-empty:

```
safeFindMin :: (Ord a) => BinomialHeap a -> Maybe a
safeFindMin h
  | hasOnlyZeros h = Nothing
  | otherwise = Just (minRootInHeap h)
```

This implementation pattern—checking `hasOnlyZeros` to enable calling `minRootInHeap`—is typical in the verification, where predicates in guards establish preconditions for subsequent operations.

5.4 Summary

This chapter demonstrated the verification of two priority queue implementations in LiquidHaskell, each presenting distinct challenges and verification techniques.

For **leftist heaps** (§5.2), we achieved complete verification of all priority queue operations. The verification relies on refined data types that directly encode the heap property, leftist property, and rank correctness. The key challenge was proving that merge operations preserve these invariants, which required explicit lemmas for transitivity of lower bounds and bag equality preservation. The relatively simple structure of leftist heaps allowed for straightforward induction over the recursive tree structure.

For **binomial heaps** (§5.3), we verified a more complex compositional structure consisting of three layers: perfect binary trees with left-ordering, pennants with minimum roots, and bit-lists representing the heap. Each layer required distinct refinements:

- The `BinTree` layer enforces left-ordering and perfect balance through height constraints.

- The `Pennant` layer ensures the minimum property by relating the root to the internal tree.
- The `BinomialHeap` layer maintains rank consistency across the bit-list structure.
- The `ReversedBinomialHeap` auxiliary structure enables efficient tree-to-heap conversion.

The binomial heap verification required careful tracking of ranks through bit-level arithmetic (§5.3.3), management of multiple heap representations (standard and reversed, §5.3.6), and coordination of multiple operations to implement `splitMin` (§5.3.7). The `dismantle` and `reverseToBinomialHeap` operations exemplify the compositional verification approach, where structural properties are maintained through lemmas about auxiliary functions (`lemma_rlast_preserved`, `lemma_rlast_tail`).

Both developments leverage the shared logical infrastructure (§5.1), particularly the `isLowerBound` predicate and the LiquidHaskell features introduced in Chapter 4: measures, reflection, PLE, and termination metrics. The systematic use of refined data types ensures that structural invariants are maintained by construction, with the type system preventing the creation of invalid heap structures.

The contrast between the two implementations illustrates important principles in verified functional programming: leftist heaps trade structural complexity for simpler verification, while binomial heaps achieve better theoretical worst-case complexity bounds at the cost of more intricate verification involving multiple data structures and auxiliary operations. Both approaches successfully use LiquidHaskell's refinement type system to achieve machine-checked correctness proofs that closely mirror the informal reasoning about these classic data structures from Chapter 3.

6 Conclusion

This thesis explored the verification of functional priority queue implementations in Haskell using LiquidHaskell, a lightweight refinement type system that integrates formal reasoning directly into the Haskell ecosystem. Our focus was on verifying essential invariants and correctness properties of two heap variants: the *Leftist Heap* and the *Binomial Heap*, each presenting distinct verification challenges and techniques.

6.1 Summary of Contributions

We presented verified implementations of key priority queue operations within LiquidHaskell, progressively building from foundational proofs over simple structures to full heap operations. Our main contributions can be summarized as follows:

- We demonstrated how algebraic data types such as `BinTree`, `LeftistHeap`, and `Pennant` can be extended with refinement types to encode structural invariants, such as the heap-order property, rank relations, and perfect balance constraints.
- We achieved complete verification of all priority queue operations—`empty`, `insert`, `merge`, `findMin`, and `splitMin`—for the leftist heap, ensuring preservation of heap order, leftist property, and rank correctness through explicit lemmas for lower-bound transitivity and bag equality.
- We developed a comprehensive verification of the binomial heap's compositional structure, spanning three layers: perfect binary trees with left-ordering (`BinTree`), pennants with minimum roots (`Pennant`), and rank-consistent bit-lists (`BinomialHeap`). This included verification of bit-level arithmetic operations (`bSum`, `bCarry`, `bAdd`), tree dismantling (`dismantle`), heap reversal (`reverseToBinomialHeap`), and the complete `splitMin` operation with its auxiliary structures and lemmas.
- We integrated these verifications with the LiquidHaskell toolchain, demonstrating how reflection, measures, PLE, and termination metrics can encode sophisticated reasoning directly at the type level, including structural recursion over multiple data structures and compositional properties across layered abstractions.

Together, these results demonstrate that non-trivial, purely functional data structures can be reasoned about in a practical way without leaving the Haskell ecosystem.

6.2 Lessons Learned and Observations

Through the verification process, several technical and conceptual insights emerged about LiquidHaskell's capabilities and limitations.

6.2.1 Strengths of LiquidHaskell

Lightweight integration. LiquidHaskell is comparatively lightweight compared to proof assistants such as Agda or Coq. It allows reasoning directly over existing Haskell code, without the need to re-model the entire language or data structure within a new logical environment. This makes it an ideal choice for incremental verification, especially when working with real-world Haskell libraries.

Expressiveness through refinements. By attaching logical predicates directly to types, we can express invariants such as heap order, height relations, or non-emptiness in a concise and composable way. This leads to code that remains executable Haskell, while also serving as a formal specification.

Automation via PLE and SMT solving. The combination of *Proof by Logical Evaluation (PLE)* and SMT-based refinement checking often enables LiquidHaskell to automatically discharge many proof obligations, making routine verification nearly effortless once the correct refinements are in place.

6.2.2 Limitations and Practical Challenges

Despite its elegance, the verification process also revealed several limitations that made large-scale verification challenging:

Reflection and reuse. LiquidHaskell's reflection mechanism is both powerful and restrictive. Any function that is used within a reflected definition must itself be reflected, which prevents direct reuse of many Haskell standard-library functions that are not reflected into the refinement logic. This recursive dependency forces the user to re-implement or re-reflect auxiliary functions, leading to significant boilerplate and occasional code duplication.

Limited type-class support. Although type classes are a fundamental abstraction in Haskell, their interaction with LiquidHaskell's refinement logic remains fragile. In principle, one could encode general invariants for a class of data structures (e.g., a `VerifiedHeap` type class), but in practice the refinement checker struggles to generalize constraints and resolve overloaded instances. Consequently, many proofs must be carried out on a per-function basis, reducing modularity and reuse.. This limitation was particularly evident in our binomial heap verification, where the multi-layered structure (`BinTree`, `Pennant`, `BinomialHeap`) required duplicated proof patterns across layers rather than shared abstractions

Error reporting and debugging. Error messages in LiquidHaskell can be difficult to interpret. Because LiquidHaskell translates annotated Haskell code into GHC Core and then re-interprets it in the SMT-based refinement logic, the resulting errors often refer to internal Core identifiers rather than the original Haskell source. This two-layer translation complicates the debugging process and makes it hard to locate the precise cause of a failed proof.

The "butterfly effect" of verification. Another practical issue is the high interdependence of reflected functions. Small logical inconsistencies or subtle changes in one definition can propagate widely, causing unrelated proofs to fail elsewhere. This "butterfly effect" makes large proof developments fragile and demands a disciplined structure of modular proofs and minimal reflection scopes.. This was particularly evident in the binomial heap verification, where changes to bit-level operations or rank predicates required revisiting multiple layers of the compositional structure

Limited higher-order reasoning. LiquidHaskell's logic is essentially first-order, and while it supports function values at the Haskell level, reasoning about higher-order properties (e.g., monotonicity of a function parameter) is not straightforward. This limits the expressiveness for verifying generic combinators or polymorphic invariants.

6.3 Future Directions

There are several promising directions to extend this work:

- **Improved abstraction mechanisms.** A richer integration of type classes or modular refinement signatures could improve code reuse across verified data structures, particularly for multi-layered compositional structures like binomial heaps.
- **Interoperability with other provers.** Linking LiquidHaskell with external proof assistants (e.g., exporting obligations to Coq or Lean) could allow deeper reasoning when needed, while keeping most proofs lightweight.
- **Enhanced error reporting and tooling.** Improvements to LiquidHaskell's diagnostic layer—mapping errors more directly to source expressions—would greatly improve usability for larger projects, especially when working with complex compositional structures involving multiple data types and auxiliary operations.
- **Verification of additional priority queue variants.** Future work could explore other heap structures such as Fibonacci heaps, pairing heaps, or skew heaps, comparing their verification complexity and examining how different structural invariants interact with LiquidHaskell's refinement logic.

- **Performance and amortized analysis.** While this thesis focused on functional correctness, extending the verification to cover amortized complexity bounds and performance properties would provide stronger guarantees about the practical efficiency of verified implementations.

6.4 Concluding Remarks

This thesis showed that LiquidHaskell offers a compelling compromise between expressiveness and practicality: it enables formal verification *within* the host programming language, avoiding the steep learning curve and reimplementation overhead of traditional proof assistants. While the current ecosystem exhibits technical limitations—especially around reflection, typeclass abstraction, and error feedback—its lightweight nature makes it uniquely suited for embedding correctness reasoning into everyday functional programming practice.

The successful verification of both leftist heaps and binomial heaps demonstrates that LiquidHaskell can handle data structures of varying complexity: from the relatively straightforward tree-based leftist heaps to the intricate multi-layered compositional structure of binomial heaps with their bit-level arithmetic and multiple auxiliary operations. This range of examples illustrates the trade-offs between structural simplicity and verification complexity, showing that refinement types can effectively reason about both simple recursive structures and sophisticated compositional architectures.

Ultimately, the experience gained from verifying these priority queues demonstrates that refinement types can bridge the gap between formal methods and real software engineering, turning correctness from a post-hoc assurance into a first-class component of functional design.

List of Figures

Acknowledgements

I would like to express my gratitude to the developers and maintainers of `LiquidHaskell`, whose research and documentation provided the foundation for this work.

I also acknowledge the use of AI-assisted tools during the preparation of this thesis. GitHub Copilot was used for minor code auto-completion and formatting suggestions. Large Language Models such as ChatGPT and Gemini were consulted to help improve grammar, refine phrasing, and summarize background literature for clarity. All technical contributions, research decisions, and verification work remain my sole responsibility.

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.

[Hin99] Ralf Hinze. "Explaining Binomial Heaps." In: *J. Funct. Program.* 9 (July 30, 1999), pp. 93–104. DOI: 10.1017/S0956796899003317.

[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.

[JSV20a] Ranjit Jhala, Eric Seidel, and Niki Vazou. *Programming With Refinement Types*. en. July 2020. URL: https://ucsd-progsys.github.io/liquidhaskell/ (visited on 09/16/2025).

[JSV20b] 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.

[SS90] Jörg-Rüdiger Sack and Thomas Strothotte. "A Characterization of Heaps and Its Applications". In: *Information and Computation* 86.1 (May 1, 1990), pp. 69–86. ISSN: 0890-5401. DOI: 10.1016/0890-5401(90)90026-E. URL: https://www.sciencedirect.com/science/article/pii/089054019090026E (visited on 11/03/2025).

[Tea24] Agda Development Team. *What is Agda?* 2024. URL: https://agda.readthedocs.io/en/latest/getting-started/what-is-agda.html.

[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).