Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

Verified Functional Data Structures: Priority Queues in Liquid Haskell

Master's Thesis

by

MohammadMehran Shahidi

September 3, 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 3.9.2025

MohammadMehran Shahidi

Abstract

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

Contents

1. Introduction

1.1. Motivation

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

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

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

1.2. Problem Statement

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

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

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

1.3. Goals and Contributions

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

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

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

1.4. Structure of the Thesis

The rest of this thesis is structured as follows:

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

2. Background and Related Work

2.1. Functional Data Structures

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

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

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

2.2. Program Verification Techniques

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

2.3. Related Work

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

3. Priority Queue Implementations

3.1. Specification of Priority Queue Interface

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

Typical operations include:

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

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

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

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

Listing 3.1: *Leftist Heap Implementation in Haskell*

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

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

3.2. Leftist Heap Implementation

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

- **Heap Property** – For every node, the stored key is less than or equal to the keys of its children. This ensures that the minimum element is always found at the root.
- **Leftist Property** – For every node, the rank (also called the *right spine length*, i.e., the length of the rightmost path from the node in question to an empty node) of the left child is greater than or equal to that of the right child. This property ensures that the right spine of the heap is kept as short as possible, which in turn guarantees logarithmic time complexity for merging operations[Oka98].

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 the right spine is kept short by the leftist property and at most is 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 the chapter 5, we will verify that these implementations satisfy the priority queue interface and maintain the leftist heap invariants.

4. `LiquidHaskell` Overview

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

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

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

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

4.1. Type Refinement

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

Consider the following function:

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

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

Of course, we could use Haskell's `Maybe` type to indicate that the function returns `Nothing` for out-of-bounds indices. However, this merely shifts the handling of invalid inputs to the caller, who must remember to check for the `Nothing`.

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

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

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 $\{v : \mathrm{Int} \mid v \geq 0\}$ describes all non-negative integers. 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:

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

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

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

As an illustration, consider the invalid binding:

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

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

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

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

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

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

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

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

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

4.1.1. Function Contracts

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

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

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

4.1.2. Refined Data Types

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

```
data Slist a = Slist { size :: Int, elems :: [a] }

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

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

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

4.1.3. Lifting Functions to the Refinement Logic

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

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

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

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

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

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

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

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

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

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

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

Reflection is another useful feature that allows the user to define a function in the refinement logic, providing the SMT solver with the function's behavior [Vaz+18]. This has the advantage of allowing the user to lift in the logic functions with more than one argument, but the verification is no more automated [Vaz24]. Additionally, with the use of a library of combinators provided by `LiquidHaskell`, we can leverage the existing programming constructs (e.g. pattern-matching and recursion) to prove the correctness of the program and use the principle of programs-as-proofs. (known as Curry-Howard isomorphism)[Vaz+18; Wad15].

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

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

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

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

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

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

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

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

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

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

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

4.1.4. 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. For example, it defines the following proof combinators:

```
type Proof = ()

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

data QED = QED

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

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

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

`Proof` is a type alias for the unit type `()`, representing the result of a completed proof. The `(***)` function takes a value of type `a` and a value of type `QED`, returning `Proof` (i.e., `()`), and is used to mark the end of a proof. The `(===)` function proves equality, taking `x:a` and `y:{a | y == x}`, returning a value with refinement `{v:a | v == x && v == y}`.

Both `(?)` and `withProof` return their first argument, but differ in Liquid Haskell's equational proofing:

- `(?)`: With type `a<pa> -> b<pb> -> a<pa>`, it preserves the refinement `pa` of the input, making it ideal for maintaining properties across proof steps.
- `withProof`: With type `x:a -> b -> {v:a | v = x}`, it asserts output equality to the input (`v = x`), suited for establishing equalities to chain with `(===)` in equational reasoning.

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

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

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

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

4.2. Strengths and Limitations

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

4.3. Related Work

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

5. Verification in Liquid Haskell

5.1. Encoding Invariants

How structural and behavioral invariants are expressed in refinement types.

5.2. Use of Measures and Predicates

Defining custom properties over data.

5.3. Example Proofs

Walkthroughs of insert and deleteMin correctness.

5.4. Dealing with Termination and Recursion

How Liquid Haskell checks termination.

5.5. Challenges and Workarounds

What was hard to prove and how you solved it.

List of Figures

A. My Code

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

B. My Ideas

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

Bibliography

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

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

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

[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).

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

[Vaz+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.

[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).