# Refinement types in Haskell: Exercise Sheet 1

April 9, 2025

## Exercise 1: Vectors

In the second lecture we introduced the following type of sized vectors:

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

data Vector a = V { size :: Int, elems :: [a] }
{-@ data Vector a = V { size :: Nat, elems :: {xs : [a] | len xs = size} } @-}
```

We also introduced the idea of parametrising predicate synonyms over variables by introducing them in uppercase such as in `predicate IsNotDivisibleBy M N`. In Liquid Haskell, type synonyms and datatype refinements can similarly be parametrised (or *indexed*) over term variables. Consider the following alternative presentation of sized vectors:

```
{-@ data Vec a N = {xs : [a] | len xs = N} @-}
```

To apply `Vec a` to a particular natural number in a refinement type definition we can write `Vec a n`. For example:

```
{-@ fromVector :: v:Vector a -> Vec a {size v} @-}
fromVector = elems
```

**Part 1.** Try to define a function from `Vec a N` to `Vector a`, what problem arises?

**Part 2.** Give a definition of non-empty sized vectors first using `Vector a` and then `Vec a N`.

**Part 3.** Define the following functions on both `Vector` and `Vec`: concatenation, zip, dot product, concatMap.

**Part 4.** In Liquid Haskell we can parametrise refinement type definitions over more than a single variable. With this in mind, define a type of sized matrices (note that there are analogous approaches to both the `Vec` and `Vector` definitions).

## Exercise 2: Exercise 3: Balanced Binary Search Trees

In the second lecture, we defined a type of binary search trees as follows:

```
data Tree a
    = Leaf
    | Node { root  :: a
           , left  :: Tree a
           , right :: Tree a
           }

{-@
data Tree a
    = Leaf
    | Node { root  :: a
           , left  :: Tree {v:a | v < root}
           , right :: Tree {v:a | v > root}
           }
@-}
```

**Part 1.** Define the insertion operation on binary search trees from the lecture and then define the union operation for two trees.

**Part 2.** In lecture 3, we will learn more about measures in Liquid Haskell, and the following is an example on trees:

```
{-@ measure depth @-}
depth :: Tree a -> Int
depth Leaf = 0
depth (Node _ m n _)
    | dm > dn   = dm
    | otherwise = dn
    where
    dm = depth m
    dn = depth n
```

By adding this measure to our program, we can now include the depth function in predicates and Liquid Haskell can reason about it. Using depth, define a type of *balanced* binary search trees, whereby the depth of either branch can be at most one greater than the other.