Formal Methods in Systems Engineering

Block 3 – WS 2025

Mini Project for Block 3
Due January 16, 2026

Note: The deadline for submitting your solution of the project is January 16, 2026.

1 Running LiquidHaskell

In order to run Liquid Haskell we provide a Docker image that automatically installs all the needed dependencies. As the submissions will be tested using the Docker image, we strongly suggest to also use this method to develop your submission. Instructions on how to install docker on Windows, Linux, and Mac can be found here: `https://docs.docker.com/get-started/get-docker/`.
After you have installed Docker, download the `Dockerfile` from TUWEL[1]. Open a terminal in the directory where the file is located and run

```
docker build . -t fmi
```

This command installs the Haskell GHC compiler, the Z3 SMT-solver and builds Liquid Haskell. The process may take some time but you only need to perform it once.
Once this command is completed, you can check the refinement types in a file `FILE.hs` with Liquid Haskell using the command

```
docker run -v .:/FMI -it --rm fmi liquid FILE.hs
```

Here are some useful links to get started with Liquid Haskell:

- The official documentation website: `https://ucsd-progsys.github.io/liquidhaskell/`
- A tutorial about programming with refinement types: `https://ucsd-progsys.github.io/liquidhaskell-tutorial/`
- An online playground with many example programs: `https://liquidhaskell.goto.ucsd.edu/index.html`

2 Assignment

2.1 Part 1: Factorials

The factorial of a natural number n, denoted by $n!$ is recursively defined by

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise.} \end{cases}$$

Implement the factorial function using LiquidHaskell by completing the definition of `factorial`, shown below. Note that the syntax `/ [n]` at the end of line 2 instructs Liquid Haskell to prove termination by using the function argument `n` as a termination metric. Furthermore, verify that for every natural number `n` it holds that `factorial n ⩾ n` by replacing `true` in line 2 with a suitable expression.

```
1 {-@
2 factorial :: n:Nat -> {i:Int | i > 0 && true} / [n]
3 @-}
4 factorial :: Int -> Int
5 factorial = error "Implement this function"
```

Let $n, k \in \mathbb{N}$ with $k \leqslant n$. The *rising factorial* and *falling factorial*, denoted by $n^{\overline{k}}$ and $n^{\underline{k}}$ respectively, are defined by the following recursions:

$$n^{\overline{k}} = \begin{cases} 1 & \text{if } k = 0 \\ n \cdot (n+1)^{\overline{k-1}} & \text{otherwise} \end{cases} \qquad n^{\underline{k}} = \begin{cases} 1 & \text{if } k = 0 \\ n \cdot (n-1)^{\underline{k-1}} & \text{otherwise} \end{cases}$$

Implement `rising_factorial` and `falling_factorial` in Haskell such that `rising_factorial` n k computes $n^{\overline{k}}$ and `falling_factorial` n k computes $n^{\underline{k}}$. The Haskell type of both functions needs to be `Int → Int → Int`.
Use refinement types to specify the following properties about `rising_factorial` (and likewise about `falling_factorial`):

[1] On Windows, the file may be automatically saved as `Dockerfile.txt`. To fix this, execute `rename Dockerfile.txt Dockerfile` in a command prompt

- Both arguments of `rising_factorial` have to be greater or equal to zero.
- `rising_factorial n k` can only be called if `k ⩽ n`.
- The result of `rising_factorial n k` is greater than zero.
- The result of `rising_factorial n k` is greater or equal than `k`.

You can use the provided docker image to test your solution in two steps:

- First, running the command `docker run -v .:/FMI -it --rm fmi factorial_test` in the directory where the file `Factorial.hs` is saved, you can check if your implementation calculates the correct values. This command does not run Liquid Haskell yet, but only tests the Haskell code. The outout for correct implementations looks like:

```
Testing factorial...
Success!

Testing rising_factorial...
Success!

Testing falling_factorial...
Success!
```

- Second, as described in Section 1 you can use the command

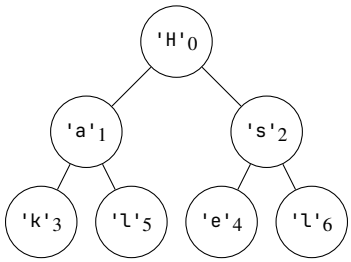
```
docker run -v .:/FMI -it --rm fmi liquid Factorial.hs
```

  to verify the specified refinement types using Liquid Haskell.

2.2 Part 2: Functional Arrays

Arrays can be implemented in functional languages using so called *Braun trees*. The file `BraunTree.hs` contains an implementation for which you are required to add refinement types.

A Braun tree is a binary tree such that the number of nodes in the left subtree is either equal to the number of nodes in its right subtree or contains exactly one more node. Hence, every Braun tree has logarithmic height, which guarantees fast search and updates. For example, consider the character array `['H', 'a', 's', 'k', 'e', 'l', 'l']` whose corresponding Braun tree is depicted on the right-hand side The indices of the elements are only shown in the figure for illustration and are not actually stored in the tree. Since the position for each element in a Braun tree is determined by the binary representation of its index, every element can be accessed in logarithmic time.



Use refinement types to implement the following tasks:

- In Haskell, binary trees are represented by an algebraic data type

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

  Complete the definition of the measure `nodes` which should count the number of nodes in a binary tree. For instance, `nodes (Node 1 (Node 2 Empty Empty) Empty)` is equal to `2`. The measure `nodes` is used in the measure `braun` such that `braun t` is true if and only if `t` is a Braun tree.

- The function `createBraunTree :: [a] → Tree a` creates a Braun tree from a list of values[2]. Add refinement type annotations to `createBraunTree` specifying that the resulting tree satisfies the predicate `braun` and the number of nodes in the tree is equal to the length of the given list[3].

[2] In Haskell, `[a]` denotes lists with values of type `a`.

[3] In refinement type annotations, the function `len` computes the length of a list.

- Replacement of a value at a given index is implemented by the `update` function. Add refinement type annotations to specify the following properties:
  - `update` may only be called on Braun trees.
  - For every call `update t i v`, `i` must be a natural number that is less than the number of nodes in `t`.
  - The result of `update t i v` is a Braun tree whose number of nodes is equal to the ones of `t`.
- The function `pop :: Tree a → Tree a` returns the Braun tree where the element at index 0 was removed. As all positive indices need to be decremented by one, some restructuring is needed. This is done by the function `merge :: Tree a → Tree a → Tree a` that is used to merge the left and right subtrees together. Add refinement type annotations to specify the following properties:
  - `merge` takes as input two Braun trees and returns a Braun tree whose number of nodes is the sum of the number of nodes of the input trees.
  - In an expression `merge t1 t2`, it either holds that `nodes t1 = nodes t2` or `nodes t1 = nodes t2 + 1`.
  - `pop` may only be called on Braun trees with a positive number of nodes.
  - The number of nodes in the tree resulting from `pop` is one less than the the number of nodes in the input tree.
- Implement a function `push :: Tree a → a → Tree a` that adds a new element at index 0 to a Braun tree while incrementing the indices of the existing elements by one. Consider the result of `push t x`. If `t` is the empty tree, the result is a tree whose root has the value `x` and the left and right subtrees are empty. Otherwise, `t` is of the form `Node y l r`. The value at the root of the resulting tree is `x` and the right subtree is `l`. Finally, the left subtree is constructed by adding `y` as a new element at index 0 to the tree `r`.

  After you have completed the implementation in Haskell, verify the following properties using refinement types:
  - `push` only takes Braun trees as input.
  - The number of nodes inside the tree `push t x` is one more than the number of nodes of `t`.

3 Submission

Submit a zip-file named `<matrnr>.zip` containing the files `Factorial.hs` as well as `BraunTree.hs` to TUWEL before January 16, 2026.